SMART POINTERS

`unique_ptr<T> and shared_ptr<T>`

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

Motivation for `std::optional<>` (1/6)

- Consider case of a function that may or may not return a value

```
int as_int(std::string const& s) {
  try {
    return std::stoi(s);
  } catch (std::invalid_argument const& ex) {
    std::cout << "invalid argument: " << ex.what() << '\n';
  } catch (std::out_of_range const& ex) {
    std::cout << "out of range: " << ex.what() << '\n';
  }
  // what is to be returned if exception is thrown?
}
```

Whenever we write function that must return a value, but that can also possibly fail, function interface must be changed

Motivation for `std::optional<>` (2/6)

- Use success-indicating return value and "*out*" parameter(s)

```
bool as_int_bool(std::string const& s, int& ri) {
  try {
    ri = std::stoi(s);
    return true;
  } catch (std::invalid_argument const& ex) {
    std::cout << "invalid argument: " << ex.what() << '\n';
    return false;
  } catch (std::out_of_range const& ex) {
    std::cout << "out of range: " << ex.what() << '\n';
    return false;
  }
}
```

Motivation for std::optional<> (3/6)

- Return pointer [smart pointer is better] that can be set to nullptr if there is failure

```
int* as_int_ptr(std::string const& s) {
  try {
    return new int{std::stoi(s)};
  } catch (std::invalid_argument const& ex) {
    std::cout << "invalid_argument: " << ex.what() << '\n';
    return nullptr;
  } catch (std::out_of_range const& ex) {
    std::cout << "out_of_range: " << ex.what() << '\n';
    return nullptr;
  }
}
```

Motivation for std::optional<> (4/6)

- Throw exception upon failure

```
// keep function signature simple by throwing
// an exception in case of failure
int as_int(std::string const& s) {
  return std::stoi(s);
}
```

Motivation for `std::optional<>` (5/6)

- Another example of function that may or may not return a value

```
struct factor_t {
  bool is_prime;
  uint64_t factor;
};

// checks if a number is prime but returns
// the first factor if there is one
factor_t factor(uint64_t n) {
  for (uint64_t i{2}; i <= n/2; ++i) {
    if (0 == n%i) {
      return factor_t{false, i};
    }
  }
  return factor_t{true, n};
}
```

Motivation for `std::optional<>` (6/6)

- Another variation of same theme …

```
// checks if a number is prime but returns
// the first factor if there is one
std::pair<bool, uint64_t> factor(uint64_t n) {
  for (uint64_t i{2}; i <= n/2; ++i) {
    if (0 == n%i) {
      return std::make_pair(false, i);
    }
  }
  return std::make_pair(true, n);
}
```

What Is `std::optional<>`?

- Previous approaches work, are fairly common, but are clumsy
- C++17 provides `std::optional<>`
  - Models *nullable* instance of arbitrary type in type-safe manner
  - Defines object of arbitrary type with additional Boolean member/flag to signal whether value exists
  - More than just structure [with Boolean flag] as data member: if there is no value, no ctor is called for contained type, allowing objects that don't have default ctor to get default state

Using `std::optional<>` (1/4)

- See *optional-asint.cpp* for abilities of `std::optional<>` as optional return type

Using `std::optional<>` (2/4)

- See *optional-name.cpp* for use of `std::optional<>` in optional passing of arguments and/or optional setting of data member

Using std::optional<> (3/4)

- See *optional-extract.cpp* for use of `std::optional<>` in optional passing of arguments

Using `std::optional<>` (4/4)

- See *optional-int.cpp* for use of `std::optional<>` in performing arithmetic with optional integer values

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


```cpp
// 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();
}
```

```cpp
// 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_ptrs

`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};
};
```

Avoid Implicit conversion like this
std::unique_ptr<int> up3(new int).
By using noexcept, this can be avoided

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;
}
```

Move constructor

Move assignment

```
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 …
  - Defining a destructor
  - Resource leaks caused by exceptions thrown during initialization of an object

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

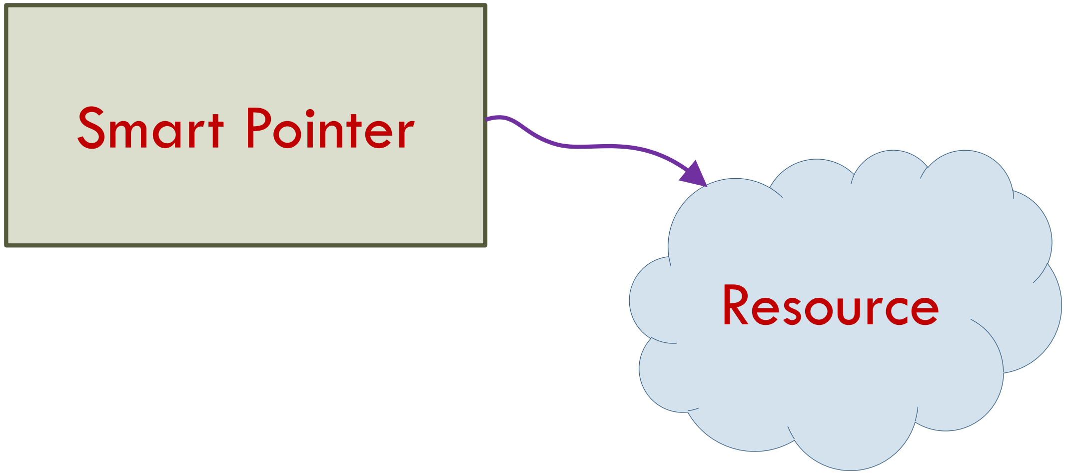
```
void coo(Widget *p);
```

std::shared_ptr<T>

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

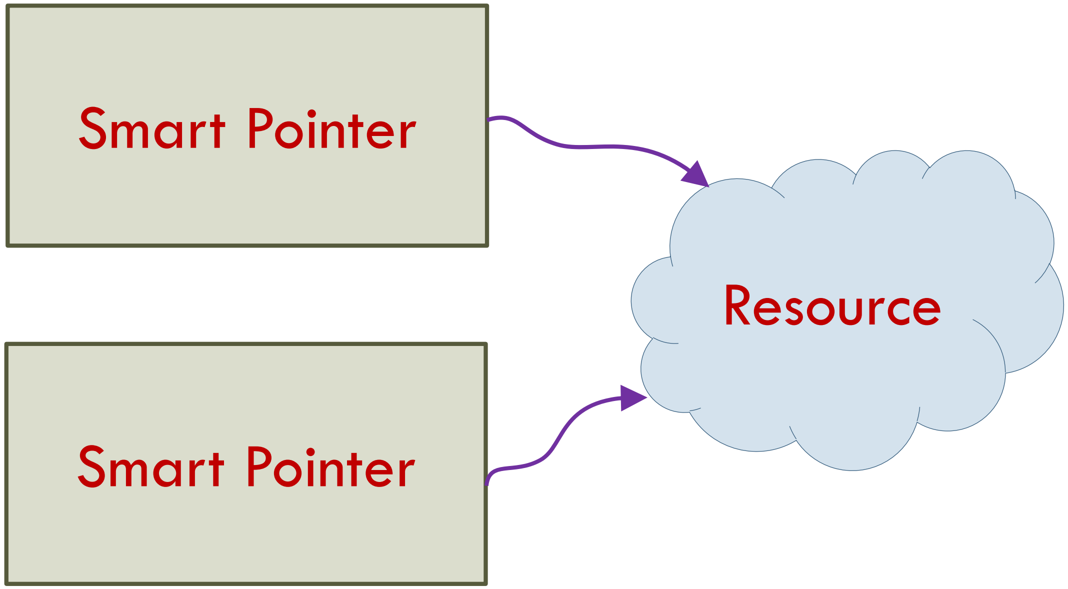
Reference Counting: Idea

- 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



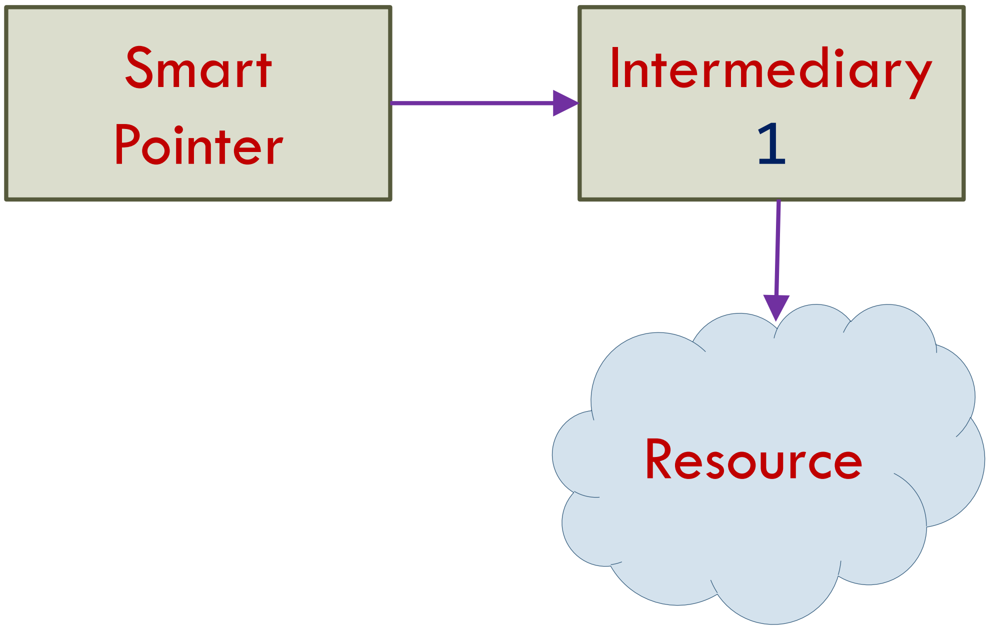
Reference Counting: Idea

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



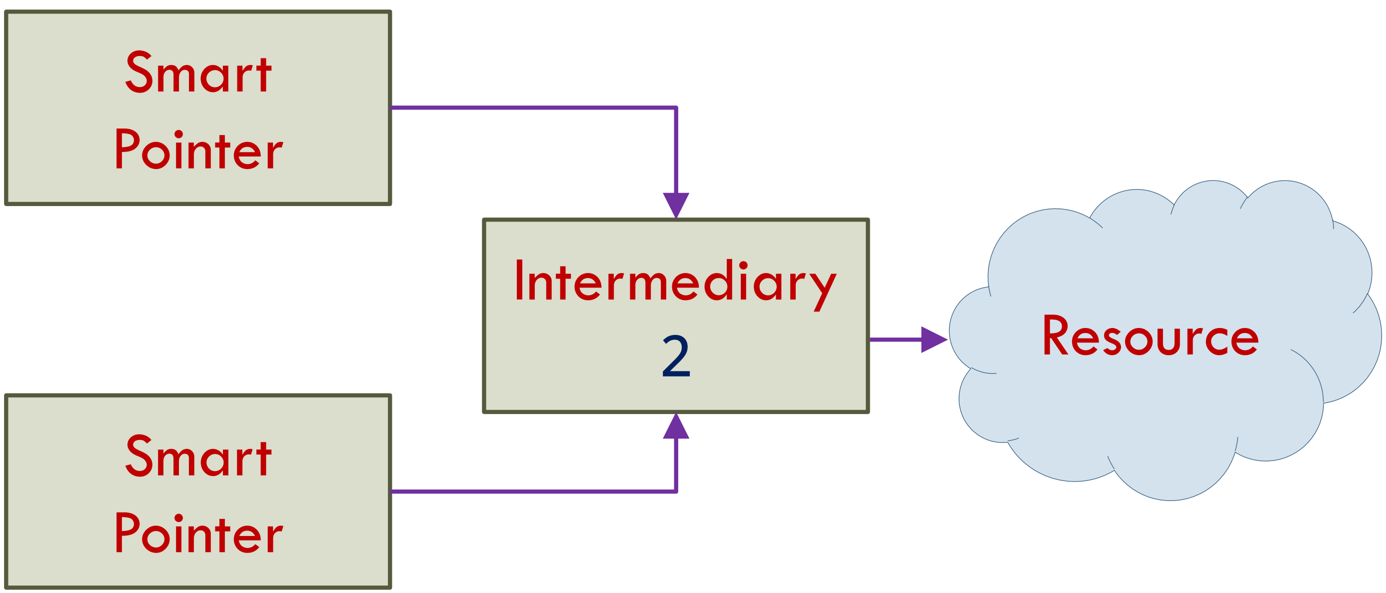
Reference Counting: Idea

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



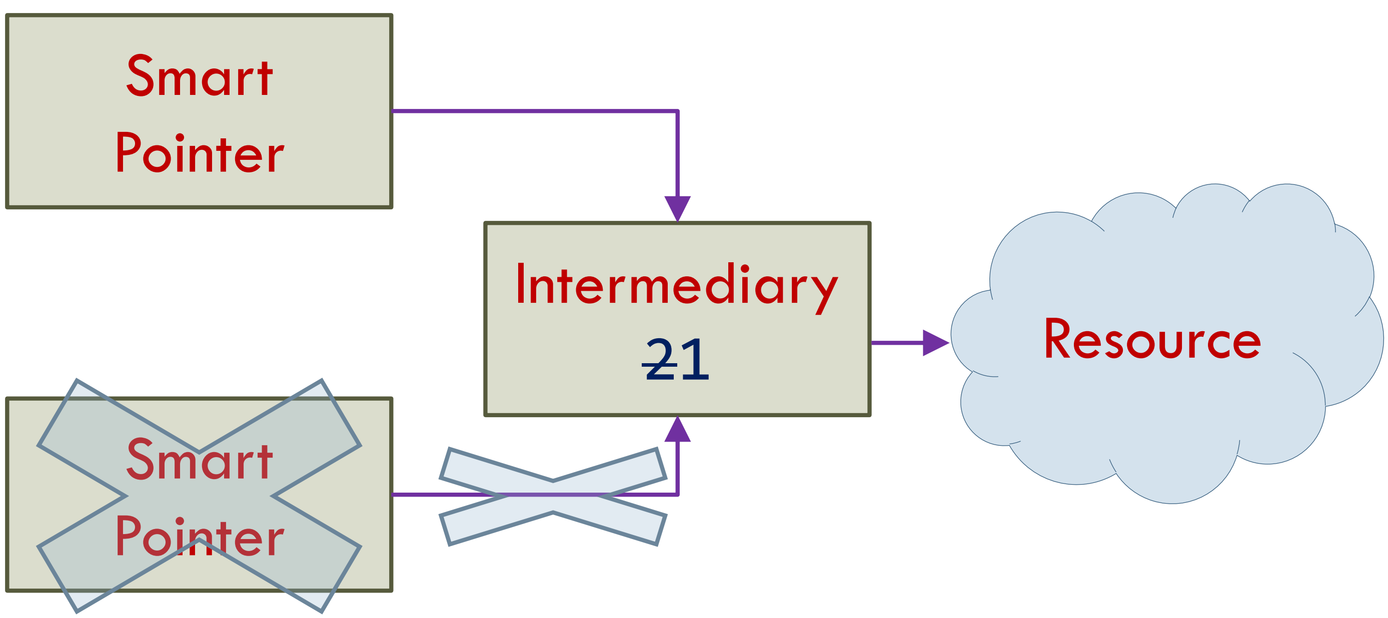
Reference Counting: Idea

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



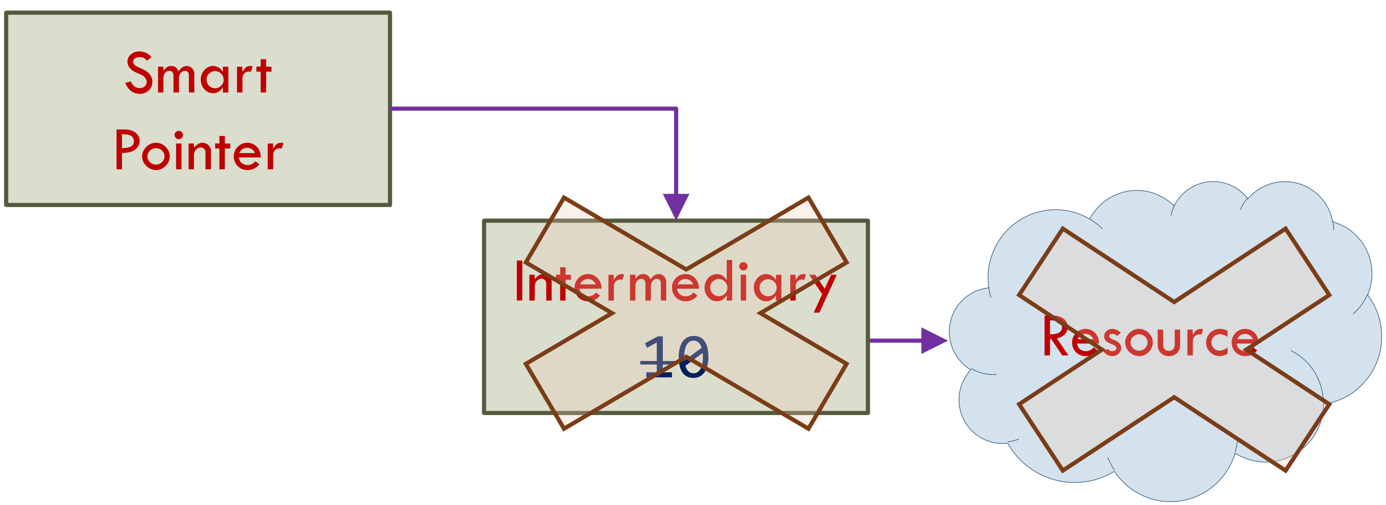
Reference Counting: Idea

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



Reference Counting: Idea

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



Reference Counting: Summary

- 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


```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);
};
```

Using `std::shared_ptr<T>`

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

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

- 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


```
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 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