ECE 449/590 – OOP and Machine Learning
Lecture 24 Smart Pointers II

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

November 16, 2022

Outline

Outline

Issues with Our Smart Pointer Implementation

Abstraction by Runtime Polymorphism

Handle Incomplete Types and Derived Types

Beyond Memory Management

Pointers and Incomplete Types

```
class A; // forward declaration

bool operation_one(A *p);
void operation_two(A *p);

void combined_operation(A *p) {
    if (operation_one(p)) {
        operation_two(p);
    }
}
```

- Opaque pointer
  - Especially for C where OO is not supported directly, e.g. `FILE *`
  - You can implement `combined_operation` without knowing anything about `A`.
- Opaque pointer is supported via incomplete types and forward declarations.

Smart Pointers and Incomplete Types

```
class A; // forward declaration

bool operation_one(shared_ptr<A> p);
void operation_two(shared_ptr<A> p);

void combined_operation(shared_ptr<A> p) {
    if (operation_one(p)) {
        operation_two(p);
    }
}
```

- What if we replace `A *` with `shared_ptr<A>` as implemented in the last lecture?
- The code won't compile.
  - When `combined_operation` returns, the dtor of `shared_ptr<A>` will be called.
  - Since the dtor of `shared_ptr<A>` is a member of a class template, it is instantiated here.
  - It need to access the dtor of `A` for `delete`, but that's not available since `A` is incomplete.

Pointers and Derived Types

```
class derived : public base {
    ...
}; // class derived

bool base_operation(base *p);

void some_function(derived *p) {
    if (base_operation(p)) {
        // do something here
    }
}
```

- Implicit conversion works here to convert a `derived` pointer to a `base` pointer.

Smart Pointers and Derived Types I

```
template<class T>
class shared_ptr {
    T *p_;
    int *count_;
public:
    ...
    template <class U>
    shared_ptr(const shared_ptr<U> &sp):
        p_(sp.p_),   // ensure U is T or derived from T
        count_(sp.count_) {
        ++*count_;
    }
    ...
}; // class shared_ptr<T>
```

- We can enable the implicit conversion among smart pointers by introducing a template ctor to replace the copy ctor.
- Note that if `U` is not `T` or is not actually derived from `T` (except `T` is void), then `p_(sp.p_)` won't compile.

Smart Pointers and Derived Types II

```
bool base_operation(shared_ptr<base> p);

void some_function(shared_ptr<derived> p) {
    if (base_operation(p)) {
        // do something here
    }
}
```

- In such case, there is a chance the `derived` object will be deleted through the `base` pointer.
- If the dtor of `base` is not `virtual`, memory leakage may happen.
- C++ does allow a base class to have a dtor not being `virtual`.

Pointers and Smart Pointers

- Our implementation of `shared_ptr` is not quite like built-in pointers
  - Need a complete type when used
  - Cannot handle derived objects with base pointers
- As the issues are both with the `delete` in dtor, can we modify the implementation to address them?
- In addition, can we use the same idea to manage resources not on heap?
  - Store a pointer to a local object (with care) in `shared_ptr`
  - Use `shared_ptr` to manage resources like `FILE *`

Outline

Issues with Our Smart Pointer Implementation

Abstraction by Runtime Polymorphism

Handle Incomplete Types and Derived Types

Beyond Memory Management

delete and Abstraction

```
~shared_ptr() {
    if (count_ != nullptr) {
        --*count_;
        if (*count_ == 0) {
            delete p_;        // <=========== here is the problem
            delete count_;
        }
    }
}
```

- All issues happen when we attempt to `delete` a pointer in the dtor of `shared_ptr`.
- We need to make sure we are deleting the pointer with the correct type.
  - Or don't `delete` at all if the resource is not on heap
- We need an abstraction of deletion to hide how the resource should be deleted/released.

Abstraction by Runtime Polymorphism

```
class ref_count {
    ref_count(const ref_count &) = delete;
    ref_count &operator=(const ref_count &) = delete;

    virtual void destroy() = 0;
    int count_;
public:
    void inc_ref();
    void dec_ref();
protected:
    ref_count() : count_(1) {}
    virtual ~ref_count() {}
}; // class ref_count
```

- The abstraction could be defined through the pure `virtual` function `destroy`.
- Follow the template method pattern, the reference count is managed in the same class.
  - The template methods are `inc_ref` and `dec_ref`.

Implement inc_ref and dec_ref

```
class ref_count {
    ...
    void inc_ref() {
        ++count_;
    }
    void dec_ref() {
        --count_;
        if (count_ == 0) {
            destroy();
            delete this;
        }
    }
    ...
}; // class ref_count
```

- The `virtual` dtor ensures derived objects of `ref_count` will be deleted correctly through `ref_count` pointers.
- These two functions can be further tweaked for multi-threaded programs/ multi-core platforms.

Updated shared_ptr Class

```
template<class T>
class shared_ptr {
    T *p_;
    ref_count *count_;
public:
    ...
    template <class U>
    shared_ptr(const shared_ptr<U> &sp) : p_(sp.p_), count_(sp.count_) {
        count_->inc_ref();
    }
    ~shared_ptr() {
        if (count_ != nullptr)
            count_->dec_ref();
    }
    ...
}; // class shared_ptr<T>
```

- The data member for reference count, the template ctor, and the dtor should be updated.
- All other members except `shared_ptr(T *p)` remain the same.

Outline

Resources on Heap

```
template <class T>
class ref_count_heap : public ref_count {
    T *p_;
    virtual void destroy() {
        delete p_;
    }
public:
    ref_count_heap(T *p) : p_(p) {}
}; // class ref_count_heap<T>
```

- For resources on heap, the `ref_count_heap<T>` class is designed.
  - It should hold a pointer to the object of type `T` on heap.
  - The `destroy` function implements how the object should be deleted.
- We do need more storage for a `ref_count_heap<T>` object than a simple `int`.
  - There is always trade-offs in your design. Here we prefer usability since the storage overhead is acceptable.

Updated Ctor

```
template<class T>
class shared_ptr {
    T *p_;
    ref_count *count_;
public:
    ...
    explicit shared_ptr(T *p) {
        try {
            count_ = new ref_count_heap<T>(p);
            p_ = p;
        }
        catch (...) {
            delete p;
            throw;
        }
    }
    ...
}; // class shared_ptr<T>
```

- Note that we use the base pointer `count_` to store a derived object created by `new` on heap.

Construction with a Complete Type

```
template<class T>
class shared_ptr {
    ...
    explicit shared_ptr(T *p) {
        ...
            count_ = new ref_count_heap<T>(p);
        ...
            delete p;
        ...
    }
    ...
}; // class shared_ptr<T>

shared_ptr<A> create_object() {
    return shared_ptr<A>(new A);
}
```

- `shared_ptr(T *p)` is instantiated when called.
- At that point, the type `T` (`A`) should be complete since we expect `*p` to be created on the heap in the same statement.
  - `delete p` should compile without a problem.
  - `ref_count_heap<T>` is instantiated at the line `count_=`...

Instantiation of ref_count_heap<T>

```
count_ = new ref_count_heap<T>(p);
```

- For a class template, the data members are instantiated when an object of the class is referred to; an member function is instantiated when they are referred to.
- For the above line, clearly it would lead to the instantiation of the data members and the ctor.
- The dtor and `destroy` will be referred within the ctor as they are virtual.
  - So they are also instantiated here.
- As the type `T` (`A`) is complete here, `destroy` will compile correctly.

Handle Incomplete Types

```
class A; // forward declaration

bool operation_one(shared_ptr<A> p);
void operation_two(shared_ptr<A> p);

void combined_operation(shared_ptr<A> p) {
    if (operation_one(p)) {
        operation_two(p);
    }
}
```

- The dtor of `shared_ptr<A>` may eventually call the `virtual` function `destroy` of `ref_count` to release the object.
  - The compiler don't need to instantiate `ref_count_heap<A>::destroy` here and don't need to know the complete type of `A` here.
  - Is `count_` pointing to a `ref_count_heap<A>` object?
- No more compiling error!

Handle Derived Types

```
template<class T>
class shared_ptr {
    ...
    template <class U>
    shared_ptr(const shared_ptr<U> &sp) : p_(sp.p_), count_(sp.count_) {
        count_->inc_ref();
    }
}; // class shared_ptr<T>
```

- `count_` will point to an object that knows the exact type of `p_`.
- We'll always release the object pointed by `p_` through a pointer of its actual type.
- No memory leakage!

Assignment and Derived Types

```
void some_function() {
    shared_ptr<derived> sp_derived(new derived);
    shared_ptr<base> sp_base;

    sp_base = sp_derived; // will it work?
}
```

- Do we need to implement a template `operator=` to support the assignment?
- `shared_ptr<base>::operator=` asks for an argument of type `const shared_ptr<base> &`.
- The compiler will use the template ctor to construct a `shared_ptr<base>` object from `sp_derived`.
- It compiles and works correctly with the current implementation.

What if ...

```
shared_ptr<base> create_derived_object() {
    return shared_ptr<base>(new derived);
}
```

- The pointer to the `derived` object will be converted to a `base` pointer implicitly.
  - Memory leakage may happen.
- I know you can implement the function as

  ```
  shared_ptr<base> create_derived_object() {
      return shared_ptr<derived>(new derived);
  }
  ```

  - But what if someone forgot to do so?

Another Template Ctor

```
template<class T>
class shared_ptr {
    ...
    template <class U>
    explicit shared_ptr(U *p) {
        try {
            count_ = new ref_count_heap<U>(p);
            p_ = p;     // ensure U is T or derived from T
        }
        catch (...) {
            delete p;   // another place that need the exact type of p
            throw;
        }
    }
    ...
}; // class shared_ptr<T>
```

- By extending the ctor `shared_ptr(T *p)` to a template, we can obtain the exact argument type for construction.
- The type information is then used to create the correct `ref_count_heap` object on heap.

Outline

Issues with Our Smart Pointer Implementation

Abstraction by Runtime Polymorphism

Handle Incomplete Types and Derived Types

Beyond Memory Management

Deleters

```
template <class T, class D>
class ref_count_deleter : public ref_count {
    T *p_;
    D deleter_;
    virtual void destroy() {
        deleter_(p_);
    }
public:
    ref_count_deleter(T *p, D d) : p_(p), deleter_(d) {}
}; // class ref_count_deleter<D>
```

- We can further generalize `delete` by a deleter.
  - For now, let's say a deleter is a function that releases a kind of resources not created on heap.
- We can derive the class `ref_count_deleter<T,D>` from `ref_count` where `D` is the type of the deleter function.
- As we will see later, type argument deduction will be leveraged to generate the type `D` and we don't need to specify it explicitly.

Template Ctor and Deleters

```
template<class T>
class shared_ptr {
    ...
    template <class U, class D>
    shared_ptr(U *p, D d) {
        try {
            count_ = new ref_count_deleter<U, D>(p, d);
            p_ = p;
        }
        catch (...) {
            d(p);       // if fails, we release p using the deleter d
            throw;
        }
    }
    ...
}; // class shared_ptr<T>
```

- Similar to `template <class U> shared_ptr(U *p)`, we can obtain the exact argument types for construction with deleters by a template ctor.
- The types `U` and `D` are used to create the correct `ref_count_deleter` object on heap.

Example I: Local Objects

```
void nullptr_deleter(void *) {} // do nothing

void some_operation(shared_ptr<A> p);

void some_function() {
    A a;
    shared_ptr<A> sp0(&a, nullptr_deleter);
    some_operation(sp0);

    shared_ptr<A> sp1(new A);
    some_operation(sp1);

    sp1 = sp0;
    sp0 = shared_ptr<A>(new A);
}
```

- With a deleter that does nothing, we can store a pointer to a local object in `shared_ptr`.
  - Be careful for its lifetime.
- It can be use together with the smart pointers with objects created on heap.
  - The derived classes of `ref_count` will make sure all objects are deleted correctly.

Example II: Objects not on Heap

```
void some_function() {
    shared_ptr<FILE> f(fopen("data.txt", "r"), fclose);

    file_operation(f);
}
```

- The file will be closed when it is no longer in use.

Example III: Clean-ups and Exception Safety

```
void some_function() {
    shared_ptr<void> p(malloc(1000), free);

    if (...) {
        ...
        return;
    }
    else if (...)  {
        ...
        return;
    }
    ...
}
```

- Clean-ups are done automatically.
  - Also guarantee exception safety
- It is the easiest way to bridge your C++ code with libraries that provide a C interface.

Summary and Advice

- Polymorphism is the key element in library design.
  - Use runtime polymorphism to hide detailed type information so couplings are minimum
  - Use compile-time polymorphism to extract detailed type information so cases can be handled in the correct way
  - The two can be combined to achieve elegant solutions for many design problems.