# ECE 449/590 – OOP and Machine Learning
# Lecture 23 Smart Pointers I

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

November 14, 2022

# Outline

# Outline

# Worry-Free Memory Management

- For many applications, it is possible one cannot decide the exact place where objects should be deleted.
  - It is now programmers' responsibility to delete them correctly.
- That's impossible for the majority of the programmers – garbage collection (GC) is a must
  - Programmers decide when objects should be created on the heap.
  - The compiler/runtime library decide when they should be deleted.
- Typical garbage collection algorithms
  - Reachability analysis: reclaim memory when necessary
  - Reference counting: delete an object immediately when it's no longer in use

# Smart Pointers

- Smart pointers: class types that can be used as pointers but are smarter than built-in pointers.
  - They will delete an object when it is no longer in use.
  - Provide GC to C++ programs
- `std::unique_ptr`: keep a non-copyable pointer to an object
  - It is straight-forward to determine when the object should be deleted – in its dtor.
- `std::shared_ptr`: allow to share a pointer to an object
  - Reference counting is used to determine when the object should be deleted.

# Outline

Smart Pointers

Reference Counting and shared_ptr

Examples

# Reference Counting

- For each object created on the heap, we need to maintain a number indicating how many pointers points to it.
  - The number is called the reference count, or the count.
- The object should be deleted when the count becomes $0$.
- Where should we store the count?
  - The count should be per object, and have the same lifetime as the object.
  - We may ask each object to hold a count, but that's not a good solution.
  - An intuitive solution: create the count on the heap and delete it when the object is deleted.
  - An efficient solution: `make_shared`, ask the count to hold the object so that a single piece of memory is allocated from heap.
- How should we update the count correctly?
  - Manually when the pointer is copied/discarded? Too tedious and error prone
  - Solution: wrap the pointer in a class so the count can be updated automatically

# The shared_ptr Class

```
template<class T>
class shared_ptr {
    T *p_;
    int *count_;
}; // class shared_ptr<T>
```

- Let's define `shared_ptr` as a class template so it can be used for any type of pointers.
- `count_` will maintain a pointer to the reference count, created on the heap.

# Class Invariants

```
template<class T>
class shared_ptr {
    T *p_;
    int *count_;
}; // class shared_ptr<T>
```

- Either both `p_` and `count_` are `nullptr`,
- Or there are `*count_` number of `shared_ptr<T>` objects holding the pointers pointing to `*p_`.
- Other pointers pointing to `*p_` are not counted since one should access the object only through `shared_ptr<T>`.

# Default Ctor

```
template<class T>
class shared_ptr {
    T *p_;
    int *count_;
public:
    shared_ptr() : p_(nullptr), count_(nullptr) {}
}; // class shared_ptr<T>
```

- For default ctor, we can simply set the pointers to `nullptr`.

# Dtor

```
template<class T>
class shared_ptr {
    T *p_;
    int *count_;
public:
    ...
    ~shared_ptr() {
        if (count_ != nullptr) {
            --*count_;
            if (*count_ == 0) {
                delete p_;
                delete count_;
            }
        }
    }
}; // class shared_ptr<T>
```

- We should decrease the count by `1` in dtor since there is one less `shared_ptr<T>` object holding the pointers.
- The object and the count should be deleted when the count drops to `0`.

# Ctor

```
template<class T>
class shared_ptr {
    T *p_;
    int *count_;
public:
    ...
    shared_ptr(T *p) {
        try {
            count_ = new int(1); // initialize the count to 1
            p_ = p;
        }
        catch (...) {
            delete p;
            throw;
        }
    }
}; // class shared_ptr<T>
```

- If we fail to create the count on the heap, we should also `delete` the pointer.
    - The user of the class `shared_ptr<T>` should not be bothered by such issue – as if the creation of the object is failed.

# Implicit Construction

```
void some_function(const shared_ptr<int> &p);

void another_function() {
    int i = 0;
    some_function(&i);
}
```

- Since a ctor of `shared_ptr<T>` would take a pointer to `T` as the argument, the above code is valid as the construction is done implicitly.
- But it's wrong – the object is not created on the heap.
- Although we cannot prohibit someone to construct a `shared_ptr<T>` object from a pointer to an object not created on the heap, we should highlight the creation of the `shared_ptr<T>` objects by prohibiting such implicit construction.

# explicit Construction

```
template<class T>
class shared_ptr {
    T *p_;
    int *count_;
public:
    ...
    explicit shared_ptr(T *p) {
        ...
    }
}; // class shared_ptr<T>
```

- The `explicit` keyword indicates the construction through the ctor must be explicit.
- Note that for ctors with at least two parameters w/o default arguments, construction must be explicit and it is not necessary to use `explicit`.

# shared_ptr and new

```
void some_function(const shared_ptr<int> &p);

void another_function() {
    int i = 0;
    // some_function(&i);                          // won't compile
    some_function(shared_ptr<int>(&i));            // wrong
    some_function(shared_ptr<int>(new int(5)); // correct
}
```

- Programmers can quickly identify errors with highlighted constructions.
  - If a `shared_ptr<T>` object is constructed and there is no `new`, then something could be wrong.
- It is probably a good idea to use `make_shared` instead of constructing `shared_ptr` from pointer.

# Copy Ctor

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

- For a built-in pointer, making a copy means to copy the address of the object instead of to copy the object itself.
- Same rule applies here – we make a copy of the pointer to the object, and a copy of the pointer to the count.
  - That's why we call `shared_ptr<T>` smart pointer.
  - Need to increase count by `1`
- Note that the parameter of the copy ctor should use the exact type `shared_ptr<T>` instead of `shared_ptr`.

# Assignment

```
template<class T>
class shared_ptr {
    T *p_;
    int *count_;
public:
    ...
    shared_ptr<T> &operator=(shared_ptr<T> rhs_copy) {
        swap(rhs_copy);
        return *this;
    }
}; // class shared_ptr<T>
```

- For exception safety, let's use copy-and-swap.
- Note that both the return type and the parameter should use the exact type `shared_ptr<T>`.

# Swap

```
template<class T>
class shared_ptr {
    T *p_;
    int *count_;
public:
    ...
    void swap(shared_ptr<T> &sp) {
        std::swap(p_, sp.p_);
        std::swap(count_, sp.count_);
    }
}; // class shared_ptr<T>
```

- It's straight-forward: just swap the members.

# Pointer Operations

```
void some_function() {
    shared_ptr<std::string> p(new std::string);

    *p = "string"; // operator*
    p->size();     // operator->
}
```

- We need to overload `*` and `->` such that a smart pointer can be used like a built-in pointer.
- Pointer arithmetics are not supported since it points to an object but not an array of object.

# Dereference

```
template<class T>
class shared_ptr {
    T *p_;
    int *count_;
public:
    ...
    T &operator*() const {
        assert(p_ != nullptr);
        return *p_;
    }
}; // class shared_ptr<T>
```

- `*` should return a reference to the object on heap.
- `*` won't change the pointer and thus can be a `const` member function.

# Arrow

```
template<class T>
class shared_ptr {
    T *p_;
    int *count_;
public:
    ...
    T *operator->() const {
        assert(p_ != nullptr);
        return p_;
    }
}; // class shared_ptr<T>
```

- C++ allows to overload `->` but not `.`.
  - Unlike other operators, when `->` is overloaded, the compiler would interpret `p->size()` as `p.operator->()->size()`.
- `->` should return a pointer to the object on heap.

# Accessor

```
template<class T>
class shared_ptr {
    T *p_;
    int *count_;
public:
    ...
    T *get() const {
        return p_;
    }
}; // class shared_ptr<T>
```

- In case the pointer is needed, users can access it through `get()`.
- It should be used with care.
  - Note that one can always access the pointer by taking the address of the reference returned by `*` so `get()` doesn't expose more information of the smart pointer.

# Outline

# Example I

```
typedef shared_ptr<std::string> str_ptr;

str_ptr create_string(std::string s) {
    return str_ptr(new std::string(s));
}
str_ptr some_function() {
    str_ptr sp_first = create_string("first");
    str_ptr sp_second(new std::string("second"));
    str_ptr sp;

    sp = sp_first;
    sp_first = sp_second;

    return sp;
}
```

- What objects are created on the heap and when they are deleted?

# Example I: Step 1

```
str_ptr create_string(std::string s) {
    return str_ptr(new std::string(s));
}
str_ptr some_function() {
    str_ptr sp_first = create_string("first");
    ...
}
```

- Assume copy ctor is not optimized away by the compiler.
- Let's use `(str, count)` to represent the object and its reference count on the heap.
- Inside `create_string`, a temporary `str_ptr` object is first created pointing to `("first", 1)`.
- It is then copied to the return value: the pair becomes `("first", 2)`.
- When the function returns, the temporary `str_ptr` object is destroyed: the pair becomes `("first", 1)`.

# Example I: Step 2

```
str_ptr some_function() {
    str_ptr sp_first = create_string("first");
    str_ptr sp_second(new std::string("second"));
    str_ptr sp;
    ...
}
```

- The return value of `create_string` is copied to `sp_first`: the pair becomes `("first", 2)`.
- A new pair `("second", 1)` is created and pointed by `sp_second`.
- `sp` holds `nullptr` members.

# Example I: Step 3

```
str_ptr some_function() {
    str_ptr sp_first = create_string("first");
    str_ptr sp_second(new std::string("second"));
    str_ptr sp;

    sp = sp_first;
    sp_first = sp_second;

    ...
}
```

- `sp_first` is assigned to `sp`: both of them point to the pair `("first", 3)`
  - The return value of `create_string` is not destroyed yet – that's the third smart pointer pointing to the pair.
- `sp_second` is assigned to `sp_first`: both of them point to the pair `("second", 2)`
  - The reference count to `"first"` is decreased by `1` to `2`.
  - `sp` and the return value of `create_string` point to the pair `("first", 2)`.

# Example I: Step 4

```
str_ptr some_function() {
    str_ptr sp_first = create_string("first");
    str_ptr sp_second(new std::string("second"));
    str_ptr sp;

    sp = sp_first;
    sp_first = sp_second;

    return sp;
}
```

- `sp` is copied to the return value of `some_function`: the pair they point to becomes `("first", 3)`.
- What happens when `some_function` returns?
  - Local variables are destroyed in the reverse order of construction.

# Example I: Step 5

```
str_ptr some_function() {
    str_ptr sp_first = create_string("first");
    str_ptr sp_second(new std::string("second"));
    str_ptr sp;
    ...
    return sp;
}
```

- Before `some_function` returns,
  - `("first", 3)`: `sp`, return value of `some_function`, return value of `create_string`
  - `("second", 2)`: `sp_first`, `sp_second`
- `sp` is destroyed: `("first", 2)`.
- `sp_second` is destroyed: `("second", 1)`
- `sp_first` is destroyed: `("second", 0)`
  - The pair is then deleted from the heap.
- Return value of `create_string` is destroyed: `("first", 1)`
- Finally, we only have the pair `("first", 1)`, which is pointed by the return value of `some_function`.

# Example II

```
struct B;
struct A {
    shared_ptr<B> b_of_A;
}; // struct A
struct B {
    shared_ptr<A> a_of_B;
}; // struct B
void some_function() {
    shared_ptr<A> pa(new A);
    shared_ptr<B> pb(new B);

    pa->b_of_A = pb;
    pb->a_of_B = pa;
}
```

- Any thing wrong?

# Example II: Cycles Lead to Memory Leakage

```
struct B;
struct A {
    shared_ptr<B> b_of_A;
}; // struct A
struct B {
    shared_ptr<A> a_of_B;
}; // struct B
void some_function() {
    shared_ptr<A> pa(new A); // (A, 1)
    shared_ptr<B> pb(new B); // (A, 1), (B, 1)

    pa->b_of_A = pb;         // (A, 1), (B, 2)
    pb->a_of_B = pa;         // (A, 2), (B, 2)

    // pb is destroyed:         (A, 2), (B, 1)
    // pa is destroyed:         (A, 1), (B, 1)
}
```

- Memory leakage!
- Limitation of reference counting based smart pointers: at runtime, if the smart pointers form a cycle, resources won't be released properly.

# Summary and Advice

- `std::shared_ptr` provides reference counting based GC and resource management to C++ programs.
  - It helps to improve the quality of application and should be used whenever you cannot determine the exact lifetime of an object.
  - However, programmers should ensure no cycle may form at runtime to prevent resource leakage.