CS100 Recitation 10

Ainsley

May 18, 2022

Contents

Contents

What is Smart Pointer

- A Smart Pointer is a wrapper class over a pointer intended to make it more easier and safer to manage memory.
- There are two type of Smart Pointer used to manage object:
  - **Shared Pointer** (`std::shared_ptr`)
  - **Unique Pointer** (`std::unique_ptr`)
- There is another special type of Smart Pointer: **Weak Pointer**(`std::weak_ptr`), which is served as an observer and should not be used alone.

What is Smart Pointer

- A Smart Pointer is a wrapper class over a pointer intended to make it more easier and safer to manage memory.
- There are two type of Smart Pointer used to manage object:
  - **Shared Pointer** (`std::shared_ptr`)
  - **Unique Pointer** (`std::unique_ptr`)
- There is another special type of Smart Pointer: **Weak Pointer**(`std::weak_ptr`), which is served as an observer and should not be used alone.
- Smart Pointers are also template. To use Smart Pointers, include `memory` header file.

Why We Use Smart Pointers

- Dynamic memory management is very error-prone
  - Forgot to `delete` memory: Memory Leak
  - Dereferencing dangling pointer/wild pointer.
  - `delete` the same memory twice.

Why We Use Smart Pointers

- Dynamic memory management is very error-prone
  - Forgot to `delete` memory: Memory Leak
  - Dereferencing dangling pointer/wild pointer.
  - `delete` the same memory twice.

```
int *p = new int(42); // allocate memory
auto q = p;           // p and q point
                      // to the same memory
delete p;             // p and q are both
                      // dangling pointers
p = nullptr;          // Ok. But q is still
                      // dangling pointer
```

Why We Use Smart Pointers

- Dynamic memory management is very error-prone
  - Forgot to `delete` memory: Memory Leak
  - Dereferencing dangling pointer/wild pointer.
  - `delete` the same memory twice.

```
int *p = new int(42); // allocate memory
auto q = p;           // p and q point
                      // to the same memory
delete p;             // p and q are both
                      // dangling pointers
p = nullptr;          // Ok. But q is still
                      // dangling pointer
```

It would be nice if the memory can be automatically deleted when it is no longer needed.

Contents

Shared Pointer Introduction

- The owner of the resource is responsible for freeing the memory that was allocated.
- Shared Pointers have the ability of taking ownership of a pointer and share that ownership.
- Share Pointers provide limited garbage-collection based on reference counting.

Declare a Shared Pointer

```
shared_ptr<string> p1;       // can point to string.
shared_ptr<vector<int>> p2; // can point to vector<int>.
auto p3 = make_shared<double>(3.14);
// use make_shared to create a shared pointer.
auto p4(p3);
// copy construct. point to p1 and share the ownership.
shared_ptr<int> p5(new int(42));
// point to a newly allocated int.
shared_ptr<int> p6 = new int(43);
// Error: converting from raw pointer to shared pointer
// is explicit.
```

Declare a Shared Pointer

```
shared_ptr<string> p1;       // can point to string.
shared_ptr<vector<int>> p2; // can point to vector<int>.
auto p3 = make_shared<double>(3.14);
// use make_shared to create a shared pointer.
auto p4(p3);
// copy construct. point to p1 and share the ownership.
shared_ptr<int> p5(new int(42));
// point to a newly allocated int.
shared_ptr<int> p6 = new int(43);
// Error: converting from raw pointer to shared pointer
// is explicit.
```

The safest way to create a shared pointer is to use `make_shared`.

Reference Counting

- When default constructed, the reference count is 0.
- When constructed with a raw pointer, or through `make_shared`, the reference count is 1.
- The reference count is increased by 1 when copied.
- When the reference count is decreased to 0, the memory is freed.

Reference Counting

```
shared_ptr<double> p1;                    // ref-count: 0
auto p3 = make_shared<double>(3.14); // ref-count: 1
auto p4(p3); // ref-count: 2
p1 = p3;     // ref-count: 3
p3.reset(new double(2.7)); // reset p3 to a new memory.
// p3 have ref-count 1. p1 and p4 have ref-count 2
p4.reset(); // ref-count: 1
p1.reset(); // ref-count: 0. memory is freed.
```

Reference Counting

```
shared_ptr<double> p1;                    // ref-count: 0
auto p3 = make_shared<double>(3.14); // ref-count: 1
auto p4(p3); // ref-count: 2
p1 = p3;     // ref-count: 3
p3.reset(new double(2.7)); // reset p3 to a new memory.
// p3 have ref-count 1. p1 and p4 have ref-count 2
p4.reset(); // ref-count: 1
p1.reset(); // ref-count: 0. memory is freed.
```

- The difference between `reset()` and assignment operator.
- `reset()` can accept an object which can be converted into `shared_ptr` (e.g. raw pointer)
- assignment operator can only accept `shared_ptr`.

Pass a Deleter

- Sometimes we use creater creater and deleter function to manage an object.

Pass a Deleter

- Sometimes we use creater creater and deleter function to manage an object.
- For example, suppose we are using a network library.

```
struct destination; // connect to where.
struct connection;  // the information of the connection.
connection connect(destination *d); // allocate memory
void disconnect(connection c); // free the memory.
void f(destination &d) {
  connection c = connect(&d);
  // if we forget to call disconnect()
  // we will never be able to disconnect c.
}
```

Pass a Deleter

- We would like our shared pointer to automaticall call `disconnect()` for us.

```
void f(destination &d) {
  connection c = connect(&d);
  shared_ptr<connection> p(c, disconnect);
  // p will call disconnect() when it is destroyed.
  // this is also exception-safe
}
```

Shared Pointer Pitfalls

Mixing shared pointers and raw pointers is begging for trouble.

```
void process(shared_ptr<int> ptr) {
  do_something(ptr);
} // ptr was destroyed here.
// If we pass a shared_ptr, the ref-count of ptr
// is at least 2. Life is good.
shared_ptr<int> p(new int(42)); // ref-count 1.
process(p); // in process, the ref-count is 2.
int i = *p; // correct, the ref-count is 1.
```

Shared Pointer Pitfalls

Mixing shared pointers and raw pointers is begging for trouble.

```
void process(shared_ptr<int> ptr) {
  do_something(ptr);
} // ptr was destroyed here.
// Although we can't pass raw pointer directly,
// we can create a temporary shared_ptr.
int *x(new int(1024)); // raw pointer.
process(x); // Error: Converting from raw pointer
            // to shared pointer is explicit.
process(shared_ptr<int>(x));
// correct, but the memeory will be freed.
int j = *x; // UB: x is dangling pointer.
```

Shared Pointer Pitfalls

- `get()` returns the raw pointer underlying the shared pointer. You should be careful with this function.
  - Do not `delete` the pointer return by `get()`
  - Do not use `get()` to initialize or `reset()` another shared pointer.

```
shared_ptr<int> p1(new int(42)); // ref-count 1
int *q = p1.get();
{
  shared_ptr<int> p(q); // ref-count 1
} // p is destroyed and memory is freed.
int* i = *q; // UB: q is dangling pointer.
```

Shared Pointer Pitfalls

- `get()` returns the raw pointer underlying the shared pointer. You should be careful with this function.
  - Do not `delete` the pointer return by `get()`
  - Do not use `get()` to initialize or `reset()` another shared pointer.

```
shared_ptr<int> p1(new int(42)); // ref-count 1
int *q = p1.get();
{
  shared_ptr<int> p(q); // ref-count 1
} // p is destroyed and memory is freed.
int* i = *q; // UB: q is dangling pointer.
```

- Do not use the same raw pointer to initialize two shared pointers.

Contents

What is a Unique Pointer?

- Same as Shared Pointers, Unique Pointers have the ability of taking ownership of a pointer.
- However, Unique Pointer do not share its ownership.
- Only one unique pointer can point to an object at the same time.
- When the unique pointer is destroyed, the object is destroyed.

Declare a Unique Pointer

Unique Pointer can be direct initialized or by make_unique(C++14).

```
unique_ptr<int> p1(new int(42));  // correct.
unique_ptr<int> p2(p1);
// Error: unique_ptr do not support copy construction.
unique_ptr<int> p3; // correct. default-constructed.
p3 = p1; // Error: do not support assignment operator.
unique_ptr<double> p3 = make_unique<double>(3.14);
unique_ptr<int[]> p4 = make_unique<int[]>(10);
// For array types.
```

Pass a Deleter

Just like shared pointers, we can also pass a deleter to unique pointer.

```
void f(destination &d) {
  connection c = connect(&d);
  unique_ptr<connection, decltype(disconnect)*>
    p(&c, disconnect);
}
```

But the type of the function should be determined at compile time.

Change the ownership of a Unique Pointer

- unique pointer have a function called `release()`, which is used to give up the ownership of the pointer.
- `release()` returns the pointer underlying the unique pointer and set the unique pointer to null.

```
unique_ptr<int> p1(new int(42));
auto q = p1.release();
// correct. but we have to delete q manually.
```

Change the ownership of a Unique Pointer

- unique pointer have a function called `release()`, which is used to give up the ownership of the pointer.
- `release()` returns the pointer underlying the unique pointer and set the unique pointer to null.

```
unique_ptr<int> p1(new int(42));
auto q = p1.release();
// correct. but we have to delete q manually.
```

- We can use `reset()` and `release()` together to transfer ownership from one unique pointer to another.

```
unique_ptr<int> p3(new int(42));
unique_ptr<int> p4;
p4.reset(p3.release());
```

Return a unique pointer from a function

How can we return a unique pointer from a function if it does not support copy?

Return a unique pointer from a function

How can we return a unique pointer from a function if it does not support copy?
We will return to this problem after we have learnt move semantics.

Contents

What is Weak Pointer

- Different from shared pointers and unique pointers, weak pointers do not have the ability of taking ownership of a pointer.
- Weak pointer should point to a object belongs to a shared pointer, but it does not contribute to the ref-count of the shared pointer.
- Even if the weak pointer points to the object, the object still can be destroyed.

Observe the Object

- Since weak pointer may point to a object that is already destroyed, we can use `expired()` to check if the object is still alive.
- For the same reason, we should use `lock()` to get a shared pointer to the object.
- Weak pointer is a good way to solve dangling pointer.

Cyclic Dependency

Suppose we have two class:

```
class A {
  shared_ptr<B> p_;
 public:
  A() {  cout << "A()\n"; }
  ~A() { cout << "~A()\n"; }
  void setShared(
    shared_ptr<B>& p) {
    p_ = p;
  }
};
```

```
class B {
  shared_ptr<A> p_;
 public:
  B() { cout << "B()\n"; }
  ~B() { cout << "~B()\n"; }
  void setShared(
    shared_ptr<A>& p) {
    p_ = p;
  }
};
```

Cyclic Dependency

What is the output of the function:

```
void cyclic() {
  shared_ptr<A> a_ptr(new A);
  shared_ptr<B> b_ptr(new B);
  a_ptr->setShared(b_ptr);
  b_ptr->setShared(a_ptr);
}
```

Cyclic Dependency

What is the output of the function:

```
void cyclic() {
  shared_ptr<A> a_ptr(new A);
  shared_ptr<B> b_ptr(new B);
  a_ptr->setShared(b_ptr);
  b_ptr->setShared(a_ptr);
}

// A()
// B()
```

The destructor was not called. There was a memory leak.

Cyclic Dependency

Now we replace the shared pointer with weak pointer.

```
class A {
  weak_ptr<B> p_;
 public:
  A() { cout << "A()\n"; }
  ~A() { cout << "~A()\n"; }
  void setShared(
    weak_ptr<B> p) {
    p_ = p;
  }
};
```

```
class B {
  weak_ptr<A> sP1;
 public:
  B() {  cout << "B()\n"; }
  ~B() { cout << "~B()\n"; }
  void setShared(
    weak_ptr<A> p) {
    p_ = p;
  }
};
```

The problem solved.

Contents

Seperate .cpp files

- As the size of the program grows, it takes more and more time to compile.
- It also difficult to understand the overall structure of the program.

Seperate .cpp files

- As the size of the program grows, it takes more and more time to compile.
- It also difficult to understand the overall structure of the program.
- A good way is to seperate the program into different parts.
- When compiling, we only need to compile the modified parts and then link them together.
- The seperation of the program is also a good way to make the program easier to understand.

Seperate .hpp files

Suppose we have two functions and we want to put them into two different files.

```
// A.cpp
void do_something() {
  do_something_else();
}
// B.cpp
void do_something_else() {
  // do things here...
}
```

But this code will not compile, because in A.cpp, `do_something_else()` is not defined.

Seperate .hpp files

To solve this problem, we can put the declaration into another file, and include it when needed.

```
// A.hpp
void do_something();
// A.cpp
#include "A.hpp"
#include "B.hpp"
void do_something() {
  do_something_else();
}
```

```
// B.hpp
void do_something_else();
// B.cpp
#include "B.hpp"
void do_something_else() {
  // do things here...
}
```

Header Guard

To avoid including the same header file multiple times, we should write header guard in each header file.

```
// In My_Class.hpp
#ifndef MY_CLASS_HPP
#define MY_CLASS_HPP
    // ...
#endif // MY_CLASS_HPP
```

Deal with Templates

What if we want to put the declarations and implementations in different files?

Deal with Templates

What if we want to put the declarations and implementations in different files?

```
template<typename Tp>
class Foo{
 public:
  Foo(Tp data);
  void bar();
 private:
  Tp data_;
}
#include "foo.inc"
```

foo.hpp

```
template<typename Tp>
Foo<Tp>::Foo(Tp data)
  : data_(data) {}

template<Typename Tp>
void Foo<Tp>::bar() {
  std::cout << data_
            << std::endl;
}
```

foo.inc

Include the implementation file at the end of the header file.

Contents

What is Name Space

- A namespace is a declarative region that provides a scope to the identifiers (names of the types, function, variables etc.) inside it.
- `std` is the standard namespace.
- To refer to an identifier in the namespace, use `::`. (e.g. `std::vector`)

Why we need it

- Namespace provide a method for preventing name conflicts in large projects.
- Do not write `using namespace` directly, especially not in a header file. This will reintroduce the name conflict into your code.

The Interface Principle

For a class X, all functions, including free functions, that both

(a) "mention" X, and

(b) are "supplied with" X

are logically part of X, because they form part of the interface of X.

- Putting the functions and the classes into the same namespace is a good way to do the seperation. Just like packages in Python.

Defining a Namespace

```
// declare a namespace
namespace ShanghaiTech {
  class SIST_Student {/* */}
  void debug_programs(
      SIST_Student &);
  class SLST_Student {/* */}
  void write_paper(
      SLST_Student &);
  // ...
}
```

ShanghaiTech.hpp

```
#include "ShanghaiTech.hpp"
// open the namespace
namespace ShanghaiTech {
  /* The definations of the
      classes and functions
      */
}
```

ShanghaiTech.cpp