

Smart Pointers

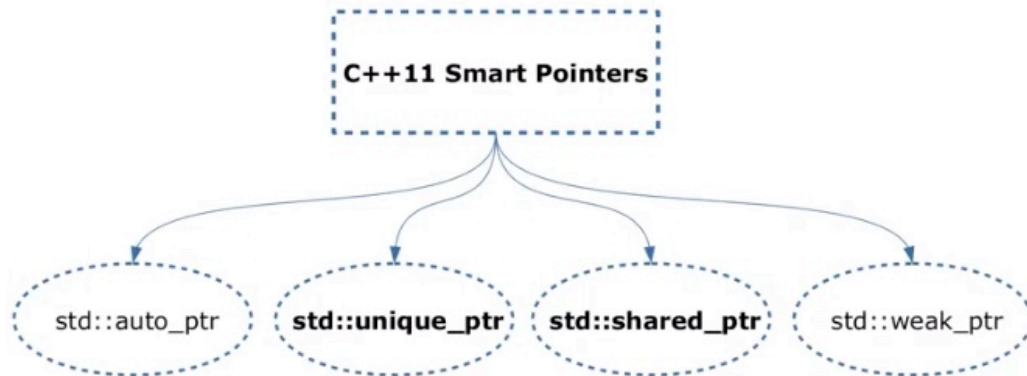
Raw pointers are hard to love

- Its declaration doesn't indicate whether it points to a single object or to an array.
- Its declaration reveals nothing about whether you should destroy what it points to when you are done using it, if the pointers owns the thing it points to.
- It may not be possible to determine whether to use the single-form "delete" or the array form "delete []". If you use wrong form, results are undefined.
- There is typically no way to tell if the pointer dangles. Dangling pointer arise when object are destroyed while pointers still point to them.

Smart Pointers

- Smart pointers wrap a raw pointer into a class and manage its lifetime (RAII).
- Smart pointers are all about ownership.
- Always use smart pointers when the pointer should own heap memory.
- Only use them with heap memory!
- Still use raw pointers for non-owning pointers and simple addressing storing.
- `#include <memory>` to use smart pointers.

C++11 smart pointers **types**



We will focus on 2 types of smart pointers:

- `std::unique_ptr`
- `std::shared_ptr`

Smart Pointers manage memory!

- Smart pointers apart from memory allocation behave exactly as raw pointers
 - Can be set to `nullptr`
 - Use `*ptr` to dereference `ptr`
 - Use `ptr->` to access methods
 - Smart pointers are polymorphic
- Additional functions of smart pointers
 - `ptr.get()` returns a raw pointer that the smart pointer manages
 - `ptr.reset(raw_ptr)` stops using currently managed pointer, freeing its memory if needed, sets `ptr` to `raw_ptr`

std::unique_ptr example

- Create an `unique_ptr` to a type `Vehicle`

```
1 std::unique_ptr<Vehicle> vehicle_1 =  
2 std::make_unique<Bus>(20, 10, "Volkswagen", "LPM_");  
3  
4 std::unique_ptr<Vehicle> vehicle_2 =  
5 std::make_unique<Car>(4, 60, "Ford", "Sony");
```

- Now you can have fun as we had with `raw pointers`

```
1 // vehicle_x is a pointer, so we can use it as it is  
2 vehicle_1->Print();  
3 vehicle_2->Print();
```

45

std::unique_ptr example

- `unique_ptr` are **unique**: This means that we can move stuff but **not** copy:

```
1 vehicle_2 = std::move(vehicle_1);
```

- Address of the pointers **before** the move:

```
1 cout << "vehicle_1 = " << vehicle_1.get() << endl;  
2 cout << "vehicle_2 = " << vehicle_2.get() << endl;
```

```
1 vehicle_1 = 0x56330247ce70  
2 vehicle_2 = 0x56330247cec0
```

- Address of the pointers **after** the move:

```
1 vehicle_2 = 0x56330247ce70  
2 vehicle_1 = 0
```

46

Unique pointer (`std::unique_ptr`)

- Constructor of a unique pointer takes **ownership** of a provided raw pointer
- **No runtime overhead** over a raw pointer
- Syntax for a unique pointer to type `Type`:

```
1 #include <memory>
2 // Using default constructor Type();
3 auto p = std::unique_ptr<Type>(new Type);
4 // Using constructor Type(<params>);
5 auto p = std::unique_ptr<Type>(new Type(<params>));
```

- From C++14 on:

```
1 // Forwards <params> to constructor of unique_ptr
2 auto p = std::make_unique<Type>(<params>);
```

http://en.cppreference.com/w/cpp/memory/unique_ptr

53

What makes it “unique”?

- Unique pointer has no copy constructor
- Cannot be copied, can be moved
- Guarantees that memory is always owned by a single `std::unique_ptr`
- A non-null `std::unique_ptr` always owns what it points to
- Moving a `std::unique_ptr` transfers ownership from the source pointer to the destination pointer (the source pointer is set to `nullptr`)

Shared Pointer (`std::shared_ptr`)

- What if we want to use the same pointer for different resources?
- An object accessed via `std::shared_ptr` has its lifetime managed by those pointers through shared ownership
- No specific `std::shared_ptr` owns the object
- When the last `std::shared_ptr` pointing to an object stops pointing there, that `std::shared_ptr` destroys the object it points to
- Constructed just like `unique_ptr`
- Can be copied
- Stores a usage pointer and a raw pointer
 - Increases usage pointer when copied
 - Decreases usage pointer when destructed
- Frees memory when counter reaches 0

- Can be initialised from a `unique_ptr`

```

1 #include <memory>
2 // Using default constructor Type();
3 auto p = std::shared_ptr<Type>(new Type);
4 auto p = std::make_shared<Type>();
5
6 // Using constructor Type(<params>);
7 auto p = std::shared_ptr<Type>(new Type(<params>));
8 auto p = std::make_shared<Type>(<params>);

```

The screenshot shows a C++ IDE with a file named `shared_ptr_example.cpp`. The code defines a `MyClass` with a constructor and destructor, and a `main` function that demonstrates `std::make_shared` and `use_count`. The terminal output shows the program's execution, including the pointer count increasing from 1 to 2 when a new shared pointer is created from an existing one, and then returning to 1 when the first pointer goes out of scope.

```

18 using std::endl;
17
16 class MyClass {
15 public:
14     MyClass() { cout << "I'm alive!\n"; }
13     ~MyClass() { cout << "I'm dead... :(\n"; }
12 };
11
10 int main() {
9     auto a_ptr = std::make_shared<MyClass>();
8     cout << "pointer count : " << a_ptr.use_count() << endl;
7     {
6         auto b_ptr = a_ptr;
5         cout << "pointer count : " << a_ptr.use_count() << endl;
4     }
3     cout << "Back to main scope\n";
2     cout << "pointer count : " << a_ptr.use_count() << endl;
1     return 0;
27 }

```

Terminal Output:

```

/tmp/shared_ptr c++ --std=c++17 shared_ptr_example.cpp
/tmp/shared_ptr ./a.out
I'm alive!
pointer count :1
pointer count :2
Back to main scope
pointer count :1
I'm dead... :(
/tmp/shared_ptr

```

When to use what?

- Use smart pointers when the pointer must manage memory
- By default use `unique_ptr`
- If multiple objects must share ownership over something, use a `shared_ptr` to it
- Think of any free standing `new` or `delete` as of a memory leak or a dangling pointer:
 - Don't use `delete`
 - Allocate memory with `make_unique`, `make_shared`
 - Only use `new` in smart pointer constructor if cannot use the functions above

Typical beginner error

```
1 int main() {
2     // Allocate a variable in the stack
3     int a = 42;
4
5     // Create a pointer to that part of the memory
6     int* ptr_to_a = &a;
7
8     // Know stuff about pointers eh?
9     auto a_unique_ptr = std::unique_ptr<int>(ptr_to_a);
10
11     // Same happens with std::shared_ptr.
12     auto a_shared_ptr = std::shared_ptr<int>(ptr_to_a);
13
14     std::cout << "Program terminated correctly!!!\n";
15     return 0;
16 }
```

Typical beginner error



```
1 int* ptr_to_a = &a;
2
3 // Know stuff about pointers eh?
4 auto a_unique_ptr = std::unique_ptr<int>(ptr_to_a);
5
6 // Same happens with std::shared_ptr.
7 auto a_shared_ptr = std::shared_ptr<int>(ptr_to_a);
```

```
1 Program terminated correctly!!!
2 munmap_chunk(): invalid pointer
3 [1] 4455 abort (core dumped) ./wrong_unique
```

- Create a **smart pointer** from a **pointer** to a stack-managed variable
- The variable ends up being owned both by the **smart pointer** and the stack and gets deleted twice → **Error!**

Polymorphism example using smart pointers

```
1 #include <memory>
2 #include <vector>
3 using std::make_unique;
4 using std::unique_ptr;
5 using std::vector;
6
7 int main() {
8     vector<unique_ptr<Rectangle>> shapes;
9     shapes.emplace_back(make_unique<Rectangle>(10, 15));
10    shapes.emplace_back(make_unique<Square>(10));
11
12    for (const auto &shape : shapes) {
13        shape->Print();
14    }
15
16    return 0;
17 }
```