CodInClub

Powered By BridgeLabz

Wk5 /S1/ Lecture # : DSOOPS-20

Raw and Smart Pointers

C++ Arrays and Pointers

When working with memory and pointers in C++, one really common way we store lots of values of the same type is an array.
Think of an array like a row of lockers, all lined up: each locker (element) holds a value, and all the lockers are right next to each other in memory.
For example:

```
int numbers[5] = {10, 20, 30, 40, 50};
```

This creates 5 integers in a row.
Here's the interesting part for pointers:
The name of the array (`numbers`) actually acts a lot like a pointer to the first element!
So if you do this:

```
int* p = numbers;
```

Now `p` points to the first number in the array (10).
Arrays and pointers are closely related in C++. If you want to process a whole array using a pointer, you can do things like:

```
for (int i = 0; i < 5; ++i) {
    std::cout << *(p + i) << " ";
}
```

This prints each value in the array, using pointer arithmetic—moving the pointer along the row of elements.

What Are Smart Pointers?

- Definition:
  - A smart pointer is a special pointer that manages memory automatically. When you're done with the memory, it cleans up for you.
  - "Smart" means it deletes the memory automatically when it's not needed, helping your program avoid leaks.

Why Use Smart Pointers?

- Problems with Raw Pointers:
  - You must remember to use `delete` or `delete[]`. If you forget, memory leaks happen.
  - Can accidentally delete memory twice or mix up pointer ownership.
- How Smart Pointers Help:
  - Memory is deleted automatically when it's not needed.
  - Fewer mistakes, safer and more reliable code.

Types of Smart Pointers (Overview)

(Only cover `unique_ptr` and `shared_ptr` in code examples.)

- `unique_ptr`:
  Acts like a one-person owner. Only one smart pointer owns the memory. When it goes away, memory is freed automatically.
- `shared_ptr`:
  Acts like a group with counting. Many smart pointers can own the same memory. Memory is freed only when the *last* one disappears.

Using Smart Pointers in Code

Here's how to use them with simple data types (like `int` or `double`):

`unique_ptr` Example

```
#include <iostream>
#include <memory>
```

```
int main() {
    std::unique_ptr<int> p1(new int(42));
    std::cout << "Number: " << *p1 << std::endl;
    // No delete needed-done automatically!
    return 0;
}
```

Key Point: You can't copy a `unique_ptr`; only one thing "owns" the pointer. You can move it, though.

`shared_ptr` Example

```
#include <iostream>
#include <memory>

int main() {
    std::shared_ptr<int> p1(new int(99));
    std::shared_ptr<int> p2 = p1; // Both share the memory!
    std::cout << "p1: " << *p1 << ", p2: " << *p2 << std::endl;
    std::cout << "Count: " << p1.use_count() << std::endl;
    return 0;
}
```

Key Point: Memory is freed only when all `shared_ptr`s disappear.

Comparing Raw vs Smart Pointers

	Raw Pointer	**Smart Pointer**
Needs manual delete?	Yes	No (automatic)
Can cause memory leaks?	Yes	Much less likely
Ownership model	You decide, it's manual	Automatic (unique/shared)
C++ header needed?	None	`<memory>`

Practice Problems and Activities

A. Understanding Code Output

1. What will this print? Why is there no delete?

```
#include <memory>
int main() {
    std::unique_ptr<int> x(new int(12));
    return 0;
}
```

2. How many owners after these lines? What happens at the end?

```
std::shared_ptr<int> a(new int(100));
std::shared_ptr<int> b = a;
std::shared_ptr<int> c = b;
std::cout << a.use_count() << std::endl;
```

B. Complete the Code

1. Fill in the blanks to safely store a pointer to a double using a smart pointer, print it, and avoid leaks:

```
#include <iostream>
#include <memory>
int main() {
    // Line A: Create a smart pointer to a double with value 5.5
    ------------
    std::cout << *ptr << std::endl;
    // No delete needed
    return 0;
}
```

2. Make a `shared_ptr` to an int, copy it once, then print the number of owners.

C. Short Concept Questions

- What happens if you forget to use delete with a raw pointer?
- Can two `unique_ptr<int>` point to the same memory? Why or why not?
- Why might you want to use `shared_ptr` instead of `unique_ptr`?

D. Debugging Task (Practice Memory Safety)

Rewrite code that uses raw pointers (with missing delete) using smart pointers to make it safe.

E. Challenge (Optional)

Write code that creates a triangle of shared pointers (all point to the same number), then explain when the memory is freed.