Notes: Smart Pointers

Smart pointers are objects that **automatically manage dynamic memory**, preventing memory leaks and dangling pointers. They are part of the C++ Standard Library (`<memory>` header) and follow **RAII (Resource Acquisition Is Initialization)** principles.

---

1. Types of Smart Pointers

Smart Pointer	Ownership	Use Case
`std::unique_ptr`	**Exclusive ownership**	Single owner, cannot be copied
`std::shared_ptr`	**Shared ownership**	Multiple owners, reference-counted
`std::weak_ptr`	**Non-owning reference**	Prevents circular references with `shared_ptr`

---

2. `std::unique_ptr` (Exclusive Ownership)

- Only **one** `unique_ptr` can own the memory.
- Automatically deletes memory when it goes out of scope.
- **Cannot be copied**, but can be moved (`std::move`).

Example:

```
#include <memory>
#include <iostream>
using namespace std;

int main() {
      unique_ptr<int> ptr(new int(10));  // Owns the memory
      cout << *ptr << endl;  // Output: 10

   // ptr2 takes ownership (ptr becomes nullptr)
   unique_ptr<int> ptr2 = move(ptr);

   if (!ptr) {
      cout << "ptr is now null" << endl;
```

```
    }
        // Memory automatically freed
}
```

Output:

```
10
ptr is now null
```

---

3. `std::shared_ptr` (Shared Ownership)

- **Multiple** `shared_ptr` instances can own the same memory.
- Uses **reference counting** to track ownership.
- Memory is freed when the **last** **`shared_ptr`** **is destroyed**.

Example:

```
#include <memory>
#include <iostream>
using namespace std;

int main() {
        shared_ptr<int> ptr1(new int(20));
        shared_ptr<int> ptr2 = ptr1;  // Both share ownership

        cout << *ptr1 << " " << *ptr2 << endl;  // 20 20
        cout << "Use count: " << ptr1.use_count() << endl;  // 2

    ptr1.reset();  // ptr1 releases ownership (use_count decreases)
        cout << "Use count after reset: " << ptr2.use_count() << endl;  // 1

        // Memory freed when ptr2 goes out of scope
}
```

Output:

```
20 20
Use count: 2
Use count after reset: 1
```

---

4. `std::weak_ptr` (Non-Owning Reference)

- **Does not increase reference count** (unlike `shared_ptr`).
- Used to **break circular references** (e.g., in graphs, linked lists).
- Must be converted to `shared_ptr` to access data (`lock()`).

Example:

```
#include <memory>
#include <iostream>
using namespace std;

int main() {
    shared_ptr<int> sharedPtr(new int(30));
    weak_ptr<int> weakPtr = sharedPtr;  // Does not own memory

  if (auto tempPtr = weakPtr.lock()) {  // Converts to shared_ptr
        cout << *tempPtr << std::endl;  // 30
  } else {
        cout << "Memory already freed!" << endl;
  }

  sharedPtr.reset();  // Memory freed
  if (weakPtr.expired()) {
    xout << "Weak pointer is expired" << endl;
  }

  return 0;
}
```

Output:

```
30
Weak pointer is expired
```

5. When to Use Which Smart Pointer?

Scenario	Recommended Smart Pointer
Single owner	`unique_ptr`
Shared ownership	`shared_ptr`
Observing without ownership	`weak_ptr`
C-style arrays	`unique_ptr<int[]>`

Example with Arrays:

```
std::unique_ptr<int[]> arr(new int[5]{1, 2, 3, 4, 5});
std::cout << arr[2];  // 3 (No need for delete[])
```

6. Key Benefits of Smart Pointers

✅ **Automatic memory management** (no `delete` needed).

✅ **Prevents memory leaks**.

✅ **Avoids dangling pointers**.

✅ **Thread-safe** (for `shared_ptr`).

7. Common Pitfalls

❌ **Circular references** (solved with `weak_ptr`).

❌ **Mixing raw pointers with smart pointers** (can cause double-free).

❌ **Using `get()` to manually delete memory** (defeats the purpose).

Example of a Circular Reference:

```
struct Node {
    shared_ptr<Node> next;
};
```

```
shared_ptr<Node> node1(new Node);
shared_ptr<Node> node2(new Node);
node1→next = node2;
node2→next = node1;  // Memory leak! (use weak_ptr instead)
```

Happy Coding!