Smart Pointers

terça-feira, 22 de outubro de 2024 17:07

Types of Smart Pointers

The C++ Standard Library provides three main types of smart pointers:

1. `std::unique_ptr`: Represents unique ownership of a dynamically allocated object.
2. `std::shared_ptr`: Represents shared ownership, where multiple smart pointers can point to the same object.
3. `std::weak_ptr`: A weak reference to an object that is managed by a `std::shared_ptr`, used to prevent circular references.

When to Use Each Smart Pointer

- **Use** `std::unique_ptr` when you need exclusive ownership of an object and no other entity should share that ownership. It's lightweight and efficient but cannot be copied.
- **Use** `std::shared_ptr` when you need shared ownership, such as when multiple objects or functions need access to the same object. It's more expensive due to reference counting.
- **Use** `std::weak_ptr` when you need to break cyclic dependencies in graphs of objects managed by `std::shared_ptr`, or when you want a non-owning reference.

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

- **Behavior**: A `std::unique_ptr` owns a dynamically allocated object exclusively. No two `std::unique_ptr` instances can own the same object at the same time.
- **Key Features**:
  - Automatically deletes the object when the `unique_ptr` goes out of scope or is explicitly reset.
  - Cannot be copied, but can be **moved** to transfer ownership to another `unique_ptr`.

Example:

cpp Copy code

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

class BTreeNode {
public:
    int value;
    BTreeNode(int val) : value(val) {
        std::cout << "Node created with value " << value << "\n";
```

```
    BTreeNode(int val) : value(val) {
        std::cout << "Node created with value " << value << "\n";
    }
    ~BTreeNode() {
        std::cout << "Node destroyed\n";
    }
};

int main() {
    std::unique_ptr<BTreeNode> rootNode = std::make_unique<BTreeNode>(10);  // Create a un
    std::cout << "Root node value: " << rootNode->value << std::endl;

    // Transferring ownership
    std::unique_ptr<BTreeNode> newOwner = std::move(rootNode);  // rootNode is now empty
    if (rootNode == nullptr) {
        std::cout << "rootNode is now null after ownership transfer\n";
    }
    std::cout << "New owner node value: " << newOwner->value << std::endl;

    return 0;  // Automatically cleans up the BTreeNode
}
```

Key Points:

- `std::make_unique<T>(args...)` is used to allocate a `T` object and return a `std::unique_ptr<T>`.
- **Ownership Transfer**: You can't copy a `std::unique_ptr`, but you can move it using `std::move()`.

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

- **Behavior**: A `std::shared_ptr` allows multiple smart pointers to share ownership of a dynamically allocated object. The object is destroyed when the last `std::shared_ptr` pointing to it is destroyed or reset.
- **Key Features**:
  - Internally uses a reference counter to keep track of how many `shared_ptr` instances point to the object.
  - Supports both copy and move semantics.

Example:

cpp Copy code

cpp

Copy code

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

class BTreeNode {
public:
    int value;
    BTreeNode(int val) : value(val) {
        std::cout << "Node created with value " << value << "\n";
    }
    ~BTreeNode() {
        std::cout << "Node destroyed\n";
    }
};

void displayNode(std::shared_ptr<BTreeNode> node) {
    std::cout << "Displaying node value: " << node->value << std::endl;
}

int main() {
    // Create a shared_ptr to a BTreeNode
    std::shared_ptr<BTreeNode> rootNode = std::make_shared<BTreeNode>(10);

    // Sharing ownership with another shared_ptr
    std::shared_ptr<BTreeNode> anotherNode = rootNode;
    std::cout << "Root node value: " << rootNode->value << std::endl;

    // Call a function that takes a shared_ptr by value (increases reference count)
    displayNode(anotherNode);

    std::cout << "Reference count after sharing: " << rootNode.use_count() << "\n";

    // When all shared_ptrs go out of scope, the object is destroyed automatically
    return 0;
}
```

Key Points:

- `std::make_shared<T>(args...)` is used to allocate a `T` object and return a `std::shared_ptr<T>`.
- **Reference Counting**: The object is destroyed only when the reference count (use count) reaches zero. You can use `use_count()` to check how many `shared_ptr` instances point to the object.
- **Copying and Sharing**: You can copy `std::shared_ptr` freely, and each copy increases the reference count.

3. `std::weak_ptr` (Weak Reference)

- **Behavior**: A `std::weak_ptr` is used in conjunction with `std::shared_ptr` to prevent cyclic references (e.g., two objects that reference each other via `std::shared_ptr`, causing a memory leak).
- **Key Features**:
  - A `std::weak_ptr` does not contribute to the reference count of a `std::shared_ptr`.
  - You can check if the object is still alive (not deleted) by using `lock()`.

Example:

cpp

Copy code

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

class BTreeNode {
public:
    int value;
    BTreeNode(int val) : value(val) {
        std::cout << "Node created with value " << value << "\n";
    }
    ~BTreeNode() {
        std::cout << "Node destroyed\n";
    }
};

int main() {
    std::shared_ptr<BTreeNode> rootNode = std::make_shared<BTreeNode>(10);
    std::weak_ptr<BTreeNode> weakNode = rootNode;  // weak_ptr does not increase reference

    std::cout << "Reference count: " << rootNode.use_count() << "\n";

    if (auto lockedNode = weakNode.lock()) {  // lock() gives a shared_ptr if the object i
        std::cout << "Locked node value: " << lockedNode->value << std::endl;
    } else {
        std::cout << "Node is no longer alive.\n";
    }

    rootNode.reset();  // Resetting the shared_ptr, destroying the object

    if (auto lockedNode = weakNode.lock()) {
        std::cout << "Locked node value: " << lockedNode->value << std::endl;
    } else {
        std::cout << "Node is no longer alive.\n";
    }
```

```
    } else {
        std::cout << "Node is no longer alive.\n";
    }

    return 0;
}
```

Key Points:

- A `weak_ptr` is a weak reference to an object managed by `std::shared_ptr`. It does not increase the reference count.
- **Cyclic References**: Use `std::weak_ptr` to avoid cyclic references, which can cause memory leaks with `std::shared_ptr`.
- **Locking**: Use `lock()` to obtain a `std::shared_ptr` from a `weak_ptr`, but only if the object still exists.