CS2810 OOAIA

Smart Pointers & Parallel Execution in Modern C++

Smart Pointers

C++11 introduced smart pointers in the `<memory>` header to facilitate automatic and exception-safe memory management. The three standard smart pointers are `std::unique_ptr`, `std::shared_ptr`, and `std::weak_ptr`. Each serves a distinct ownership model and use case.

1. `std::unique_ptr` – Exclusive Ownership

A `std::unique_ptr` is a smart pointer that owns and manages a dynamically allocated object through a pointer and disposes of that object when the `unique_ptr` goes out of scope. It enforces exclusive ownership, meaning that only one `unique_ptr` can own the same resource at any given time.

Key characteristics:

- Cannot be copied; supports move semantics.
- Automatically deletes the owned object when it goes out of scope.
- Lightweight and efficient; no reference counting overhead.

Typical use cases:

- Resource ownership within classes (e.g., composition).
- Ensuring a resource is not shared and cannot be accidentally copied.

2. `std::shared_ptr` – Shared Ownership

A `std::shared_ptr` is a smart pointer that retains shared ownership of an object through a reference count. Multiple `shared_ptr` instances can own the same object. The object is destroyed only when the last `shared_ptr` owning it is destroyed or reset.

Key characteristics:

- Implements reference counting to manage the lifetime of the shared object.
- Supports both copy and move semantics.
- Thread-safe reference counting (but not necessarily thread-safe object access).

Typical use cases:

- Shared ownership across multiple objects or modules.
- Object factories or global/shared configuration instances.

3. `std::weak_ptr` – Non-Owning Reference

A `std::weak_ptr` is a smart pointer that holds a non-owning reference to an object managed by `shared_ptr`. It does not affect the reference count. It is primarily used to prevent cyclic references in data structures such as graphs or parent-child trees.

Key characteristics:

- Must be converted to `shared_ptr` using `.lock()` before access.
- Used to observe an object without affecting its lifetime.
- Avoids memory leaks due to circular references.

Typical use cases:

- Observers that do not own the object.
- Parent pointers in tree-like or graph-like structures where ownership is managed elsewhere.

Parallel Execution

Modern C++ provides support for parallelism in the Standard Template Library (STL) through **execution policies**, introduced in C++17. These enable the standard algorithms to be executed in parallel, allowing developers to utilize multi-core processors more effectively.

The execution policies are provided in the `<execution>` header.

There are three main execution policies:

1. `std::execution::seq`

    - Executes the algorithm sequentially (default behavior).
    - Same as traditional STL algorithm behavior.
2. `std::execution::par`
    - Permits parallel execution of the algorithm using multiple threads.
    - Improves performance on large datasets by leveraging multicore CPUs.
3. `std::execution::par_unseq`
    - Permits parallel and vectorized (SIMD) execution.
    - Allows both parallel execution and instruction-level parallelism.
    - The order of execution is not guaranteed; suitable for algorithms with no data dependencies.

Note: `std::execution::par_unseq` may lead to non-deterministic behavior if data dependencies exist.

Example Code Snippets

1. `std::unique_ptr` for Exclusive Ownership

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

class Logger {
public:
    Logger() { std::cout << "Logger initialized\n"; }
    ~Logger() { std::cout << "Logger destroyed\n"; }
    void log(const std::string& msg) { std::cout << "[Log] " << msg
<< "\n"; }
};

void uniquePtrExample() {
    std::unique_ptr<Logger> logger = std::make_unique<Logger>();
    logger->log("Application started");
}
```

2. `std::shared_ptr` with Multiple Owners

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

class Configuration {
public:
    Configuration() { std::cout << "Config loaded\n"; }
    ~Configuration() { std::cout << "Config unloaded\n"; }
};

void sharedPtrExample() {
    std::shared_ptr<Configuration> config =
std::make_shared<Configuration>();

    auto moduleA = config;
    auto moduleB = config;
```

```
    std::cout << "Use count: " << config.use_count() << "\n"; //
Output: 3
}
```

3. `std::weak_ptr` – Prevent Cyclic Dependencies

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

class Node;

class Parent {
public:
    std::shared_ptr<Node> child;
    ~Parent() { std::cout << "Parent destroyed\n"; }
};

class Node {
public:
    std::weak_ptr<Parent> parent; // avoids cyclic reference
    ~Node() { std::cout << "Node destroyed\n"; }
};

void weakPtrExample() {
    auto parent = std::make_shared<Parent>();
    auto child = std::make_shared<Node>();

    parent->child = child;
    child->parent = parent; // weak_ptr prevents memory leak
}
```

4. Parallel Sorting Using `std::execution::par`

```
#include <vector>
#include <algorithm>
#include <execution>
#include <iostream>
```

```
void parallelSortExample() {
    std::vector<int> data = {7, 3, 1, 9, 4};

    std::sort(std::execution::par, data.begin(), data.end());

    for (int val : data) std::cout << val << " ";
}
```

5. Parallel `for_each` with a Stateless Lambda

```
#include <vector>
#include <execution>
#include <iostream>

void parallelForEachExample() {
    std::vector<int> values = {1, 2, 3, 4, 5};

    std::for_each(std::execution::par, values.begin(), values.end(),
                  [](int x) {
                      std::cout << "Square of " << x << " is " << x *
x << "\n";
                  });
}
```

6. Smart Pointer Management with Parallel Execution

```
#include <vector>
#include <memory>
#include <execution>
#include <iostream>

class Task {
public:
    Task(int id) : id(id) {}
    void process() const {
        std::cout << "Processing task " << id << "\n";
    }
```

```
private:
    int id;
};

void smartParallelExample() {
    std::vector<std::shared_ptr<Task>> tasks;
    for (int i = 0; i < 5; ++i) {
        tasks.push_back(std::make_shared<Task>(i));
    }

    std::for_each(std::execution::par, tasks.begin(), tasks.end(),
                  [](const std::shared_ptr<Task>& task) {
                      task->process();
                  });
}
```

7. Thread-Safe Logging with `shared_ptr`

```
#include <iostream>
#include <memory>
#include <mutex>
#include <thread>
#include <vector>

class Logger {
    std::mutex logMutex;
public:
    void log(const std::string& message) {
        std::lock_guard<std::mutex> guard(logMutex);
        std::cout << "[Thread " << std::this_thread::get_id() << "] "
<< message << "\n";
    }
};

void threadSafeLoggingExample() {
    auto logger = std::make_shared<Logger>();

    auto logTask = [logger](int id) {
```

```
        logger->log("Processing task " + std::to_string(id));
    };

    std::vector<std::thread> threads;
    for (int i = 0; i < 4; ++i) {
        threads.emplace_back(logTask, i);
    }

    for (auto& t : threads) t.join();
}
```

Practice Problems

1. Problem: Implement a class `ResourceManager` that owns a `ResourceManager` object using `std::unique_ptr`. Add a method `useResource()` that calls a method on the `Resource`.

Example:

```
ResourceManager mgr;
mgr.useResource();

Output:
Resource created
Using resource
Resource destroyed
```

2. Problem: Create a `Document` class. Multiple `Editor` objects should share the same `Document` using `std::shared_ptr`. Display the current reference count after creating each `Editor`.

Example:

```
std::shared_ptr<Document> doc = std::make_shared<Document>();
Editor e1(doc), e2(doc), e3(doc);

Output:
Document created
Editor count: 2
Editor count: 3
Editor count: 4
```

3. Problem: Create a `Node` class that contains a `shared_ptr<Node>` for its child and a `weak_ptr<Node>` for its parent. Verify that no memory leak occurs after creating a small tree.

Example:

```
auto parent = std::make_shared<Node>();
auto child = std::make_shared<Node>();
parent->setChild(child);
child->setParent(parent);

Expected Behavior:
No memory leaks or dangling references when parent and child go out
of scope. Use destructors to confirm cleanup.
```

4. Problem: Calculate the sum of squares of the first 100 integers in parallel. Use atomic or mutex to handle shared state safely.

Example:

```
int result = parallelSumSquares(100);
std::cout << result << "\n";

Output:
328350
```

5. Problem: Given a vector of `shared_ptr<Task>`, each with a `.run()` method, process them in parallel using `.run()execution::par`.

Example:

```
std::vector<std::shared_ptr<Task>> tasks = generateTasks(4);
runTasksInParallel(tasks);

Output (order may vary):
Task 0 running
Task 1 running
Task 2 running
Task 3 running
```