Name: Ahmed abdelghany

Lab: Final Exam

The code:

```
#include <iostream>

#include <memory>

#include <chrono>

#include <thread>

#include <cassert>


// TimedSharedPtr template class


/**

 * TimedSharedPtr<T> wraps std::shared_ptr<T> with expiration.

 * get() returns nullptr if current time > expiration.

 */

template<typename T>

class TimedSharedPtr {

private:

    std::shared_ptr<T> sptr;

    std::chrono::steady_clock::time_point expiration;


public:

    TimedSharedPtr() : sptr(nullptr),

        expiration(std::chrono::steady_clock::time_point::min()) {}
```

```
template<class Rep, class Period>
TimedSharedPtr(std::shared_ptr<T> ptr,
            const std::chrono::duration<Rep, Period>& duration)
   : sptr(ptr) {
   expiration = std::chrono::steady_clock::now() +
             std::chrono::duration_cast<std::chrono::steady_clock::duration>(duration);
}

template<class Rep, class Period>
TimedSharedPtr(T* rawPtr,
            const std::chrono::duration<Rep, Period>& duration)
   : sptr(std::shared_ptr<T>(rawPtr)) {
   expiration = std::chrono::steady_clock::now() +
             std::chrono::duration_cast<std::chrono::steady_clock::duration>(duration);
}

TimedSharedPtr(const TimedSharedPtr<T>& other)
   : sptr(other.sptr), expiration(other.expiration) {}

TimedSharedPtr(TimedSharedPtr<T>&& other) noexcept
   : sptr(std::move(other.sptr)), expiration(other.expiration) {
   other.expiration = std::chrono::steady_clock::time_point::min();
}

TimedSharedPtr& operator=(const TimedSharedPtr<T>& other) {
   if (this != &other) {
       sptr = other.sptr;
       expiration = other.expiration;
   }
   return *this;
}

TimedSharedPtr& operator=(TimedSharedPtr<T>&& other) noexcept {
   if (this != &other) {
       sptr = std::move(other.sptr);
       expiration = other.expiration;
       other.expiration = std::chrono::steady_clock::time_point::min();
   }
   return *this;
}

~TimedSharedPtr() = default;

T* get() {
   if (!sptr) return nullptr;
   if (std::chrono::steady_clock::now() >= expiration) return nullptr;
   return sptr.get();
}

T& operator*() const { return *sptr; }
T* operator->() const { return sptr.get(); }
```

```
    explicit operator bool() const {
        return sptr && (std::chrono::steady_clock::now() < expiration);
    }
};

// Test and demonstration
int main() {
    std::cout << "TimedSharedPtr Unit Tests\n";

    TimedSharedPtr<int> ptr1(new int(42), std::chrono::milliseconds(500));
    assert(ptr1.get() != nullptr);
    std::cout << "Test1: Before expiration, get() returns non-null (value = " << *ptr1.get() << ")\n";

    TimedSharedPtr<int> ptr2(new int(100), std::chrono::milliseconds(100));
    std::this_thread::sleep_for(std::chrono::milliseconds(150));
    assert(ptr2.get() == nullptr);
    std::cout << "Test2: After expiration, get() returns nullptr\n";

    TimedSharedPtr<int> ptr3(new int(55), std::chrono::milliseconds(200));
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    assert(ptr3.get() != nullptr);
    std::cout << "Test3: Midway before expiration, get() returns non-null (value = " << *ptr3.get()
<< ")\n";
    std::this_thread::sleep_for(std::chrono::milliseconds(120));
    assert(ptr3.get() == nullptr);
    std::cout << "Test3: Eventually after expiration, get() returns nullptr\n";
    auto sp = std::make_shared<int>(77);

    TimedSharedPtr<int> ptr4a(sp, std::chrono::milliseconds(300));

    TimedSharedPtr<int> ptr4b = ptr4a;

    assert(ptr4a.get() != nullptr && ptr4b.get() != nullptr);

    std::cout << "Test4: Copy constructed ptr2 (shared value = " << *ptr4b.get() << ") before
expiration\n";

    std::this_thread::sleep_for(std::chrono::milliseconds(310));

    assert(ptr4a.get() == nullptr && ptr4b.get() == nullptr);

    std::cout << "Test4: After expiration time, both ptr1 and ptr2 get() return nullptr\n";


    auto shared = std::make_shared<int>(99);

    TimedSharedPtr<int> shortPtr(shared, std::chrono::milliseconds(100));

    TimedSharedPtr<int> longPtr(shared, std::chrono::milliseconds(300));

    assert(shortPtr.get() != nullptr && longPtr.get() != nullptr);

    std::cout << "Test5: Two TimedSharedPtr with different durations initially valid\n";

    std::this_thread::sleep_for(std::chrono::milliseconds(150));

    assert(shortPtr.get() == nullptr && longPtr.get() != nullptr);

    std::cout << "Test5: After 150ms, shorter expired (get()==nullptr), longer still valid\n";

    std::this_thread::sleep_for(std::chrono::milliseconds(200));

    assert(longPtr.get() == nullptr);

    std::cout << "Test5: After 350ms total, longer expired as well (get()==nullptr)\n";


    std::cout << "All tests passed.\n";

    return 0;

}
```

Documentation – TimedSharedPtr<T>

Class Purpose

The TimedSharedPtr<T> class is a smart pointer wrapper around std::shared_ptr<T> that adds **time-based expiration**. It allows you to use shared ownership while also enforcing a validity timeout: after a specified duration, the pointer becomes unusable via get().

Only the get() method checks for expiration; other operations like dereferencing or arrow access (*, ->) do not check expiry—this is by design, as per the assignment.

Core Data Members

std::shared_ptr<T> sptr;

std::chrono::steady_clock::time_point expiration;

- sptr holds the actual managed object.
- expiration stores the time at which get() will begin returning nullptr.

Design Diagram (UML-style)

```
+-----------------------------+
|  TimedSharedPtr<T>          |
|-----------------------------|
| - sptr : shared_ptr<T>      |
| - expiration : time_point   |
|-----------------------------|
| + get() : T*                |
| + operator*() : T&          |
| + operator->() : T*         |
| + operator bool() : bool    |
| + constructor(ptr, duration)|
+----------------------------+
```

Copy & Ownership Behavior

- Copies share the same object **and** expiration timestamp.
- Moving transfers both ownership and expiration.
- Reference counting is managed through shared_ptr, ensuring proper memory deallocation.

How Expiration Works

- When a TimedSharedPtr is constructed, it records now + duration as the expiration point.
- Every call to get() checks the current time vs. the expiration.
- If the current time exceeds expiration, get() returns nullptr.

Thread Safety Note

This class does **not** implement thread synchronization (e.g., for simultaneous get() calls across threads). If used in multi-threaded contexts, external synchronization (mutexes) may be necessary depending on use case.

Github link (https://github.com/Ventapa/Lab-final.git)