Amir Zuabi - 212606222

Accelerated Parallel Computing - 046279

HW0

8/6/2024

Smart Pointers:

Question 1: [10 points] Describe at least one pro and one con of smart pointers, compared with raw pointers.

One pro of smart pointers is that we can use them to manage pointers and memory better. Because when we exit the smart pointer's scope, it destructs itself and the object it is pointing at (unless it is a shared pointer and the reference counter isn't 0).

One con of smart pointers is overhead and shared pointer loops. Unique pointers don't have much overhead, but shared pointers have non negligent overhead as discussed in our lecture. In addition to that, shared pointers can pose problems when encountering cyclic pointing. If 2 shared pointers point at each other, and each of them has some else pointing at them, it is impossible to erase/destroy them, since their reference count isn't 0.

Question 2: [20 points] Create a simple program that shows a use-case whose performance (execution time) with raw pointers is better than with smart pointers.

The code:

```cpp
int main() {
    const int amount = 1000000;

    // Measure time with raw pointers
    auto start = std::chrono::steady_clock::now();
    {
        std::vector<Enemy*> enemies;
        for (size_t i = 0; i < amount; ++i) {
            enemies.push_back(new Enemy());
        }

        // we clear the vecotr for the next use
        for (Enemy* itr : enemies) {
            delete itr;
        }
        enemies.clear();
    }
    auto end = std::chrono::steady_clock::now();
    std::cout << "Time with raw pointers: " << std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count() << " ms\n";

    // ================================================================================================================

    // Measure time with smart pointers
    start = std::chrono::steady_clock::now();
    {
        std::vector<std::unique_ptr<Enemy>> enemies;
        for (int i = 0; i < amount; ++i) {
            enemies.push_back(std::make_unique<Enemy>());
        }
        // no need to delete since smart pointers destruct on leaving this scope
    }

    end = std::chrono::steady_clock::now();
    std::cout << "Time with smart pointers: " << std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count() << " ms\n";

    return 0;
}
```

The output:

```
amzu@OBSIDIAN:~/amzu_dev/HW0$ g++ -g q3.cpp -o q3.exe
amzu@OBSIDIAN:~/amzu_dev/HW0$ ./q3.exe
Time with raw pointers: 54 ms
Time with smart pointers: 171 ms
amzu@OBSIDIAN:~/amzu_dev/HW0$
```

This program just creates a bunch of pointers, then deletes them. But even in something this simple, after a large amount of pointers we can see that it takes almost x4 time to run with smart pointers compared to raw pointers.

**should've been named q2 but ok*

Functors, function pointers and lambda functions:

Question 3: [5 points] Describe at least one case where using functors may be preferable (have an advantage) over 1) lambda functions and 2) over function pointers.

1)
Unlike lambda functions, Functors can maintain states between function calls, unlike lambdas, which without global variables and without them getting over complicated can't really maintain a state in a concise manner.

2)
Similar to point 1), functors are better for applications where we need to encapsulate and manage the function's/operation's state.

Question 4: [5 points] Describe at least one case where using functors is less preferable (has no clear advantage or clear disadvantage) over lambda functions and function pointers.

1)
For simple one-off operations, that don't require states or maintenance, there's not much difference between functors and lambda functions, except for cleaner syntax.

2)
For operation that do not require states or encapsulations, there no clear advantage over functors or regular functions, except maybe comfort and simpler syntax.