CS 3370 – C++: Assignment 2

A CS 3370 Programming Assignment 2

A-1 Managing Shared Memory for Game NPCs

A-1-1 Background

Thousands of Non-Player Characters (NPCs) may be created and destroyed rapidly during a video game. Each heap allocation has overhead, and frequent allocations can cause memory fragmentation. In this assignment, we will pre-allocate a large block of memory and manage it ourselves - a **memory pool**. You'll implement a memory pool for managing NPCs in a game engine, learning how to override C++'s memory management operators and implement efficient allocation strategies.

A-1-2 Learning Objectives

- Understand custom memory management in C++
- Override `operator new` and `operator delete`
- Implement a memory pool using modern C++ features
- Use `std::span` for safe memory views (C++20)
- Practice RAII principles

A-2 Architectural Details

You will create a class representing a non-player character in a game. You will also create a shared memory pool class called, simply, *Pool*. The NPC class will have an initialized inline static member which is the Pool class (for the shared memory pool). Thus all instances of the NPC class will use this shared pool instance.

B Part 1: The NPC Class

Create a class called `NPC` (Non-Player Character) with the attributes below. Note: if you keep your strings under 16 characters, you are probably safe as they wil be constant size due to how string objects are allocated and stored. If you have longer strings you might want to use character arrays to prevent the size of the NPC object from being variable.

i.e. if you have a `string A;`, `A.size()` does not equal `sizeof(A)`.

```
#pragma once
//npc.h decl

class NPC {

    string _name;       // e.g., "Goblin_247"
    string _type;       // e.g., "merchant", "enemy",
     ↪ "quest_giver"
    int _x, _y;              // Grid location on the map
    int _health;            // Health points
};
```

Note: this class exists for the purposes of the assignment. It does not need mutators and accessors or other common features of C++ classes. You should declare and define the following members described below:

B-1 Requirements

B-1-1 Constructor

```
NPC(const string& name, const string& type, int x, int y, int
 ↪  health = 100)
```

B-1-2 Output operator

```
//this is for ease of displaying the contents of an NPC object.
friend std::ostream& operator<<(std::ostream& os, const NPC& npc)
// Should output: "NPC[name='Goblin_247', type='enemy',
 ↪  pos=(5,10), hp=100]"
```

B-1-3 Custom memory management (to be connected to the Pool)

Implement these overrides by using member functions from the pool as described in the Pool implementation details below.

```
static void* operator new(std::size_t size);
static void operator delete(void* ptr) noexcept;
```

B-1-4 Static pool member (using C++17 inline static)

```
//allows initialization of static object as we discussed in
 ↪  class
private:
    inline static Pool pool{100, true};  // 100 NPCs, tracing
     ↪  enabled
```

B-1-5 Profiling method

```
static void profile() { pool.profile(); }
```

C Part 2: The Pool Class

Implement a memory pool class for NPCs. This uses a vector of pointers to NPC objects to manage the pool.

It also uses C++ `std::span<T>`. This C++ 20 feature lets you manage a pointer to an object and its size with some helper methods. So it's sort of like: `struct T * ptr; std::size_t size;`

C-1 NPCPool.h Structure

```
#pragma once

#include <cstddef>
#include <vector>
#include <memory>
#include <span>       // C++20
#include <iostream>
#include <format>     // C++20 (python style formatting of
 ↪  strings) optional to use

class Pool {
private:
    // Memory storage
    std::byte* _memory_block; //pointer to array of bytes ('new'
     ↪  size of array)
    std::size_t _block_size; //how we divide up the pool (size
     ↪  of NPC in this case)

    // Free slot tracking using vector
    std::vector<std::size_t> free_indices;  // Indices of free
     ↪  slots

    // Statistics
    std::size_t live_npcs = 0; //how many npcs exist in the pool
    bool trace_enabled; //if true, dump lots of extra info to
     ↪  stdout
```

```
    // get memory address for an index
    // (nodiscard means throw if the output of this member is
    ↪ being ignored)
    [[nodiscard]] void* get_slot(std::size_t index) {
        // TODO: Return address of slot at given index
    }

    // Get a safe view of the memory
    //  this is optional, but recommended
    [[nodiscard]] std::span<std::byte> get_memory_span() {
        // TODO: Return a span covering the entire memory block
    }

public:
    Pool(std::size_t count, bool trace = true);
    ~Pool();

    [[nodiscard]] void* allocate();
    void deallocate(void* ptr);
    void profile() const;
};
```

C-2 Key Implementation Points

C-2-1 Constructor

- Allocate the pool with `count * sizeof(NPC)` bytes
- Initialize `free_indices` with all indices (0 to count-1)
- If tracing, print initialization message

C-2-2 allocate()

- Pop an index from `free_indices`
- Return the corresponding memory address using `get_slot()` This memory address will be what is stored in the pointer.
- If tracing, print allocation message

- Throw `std::bad_alloc` if no slots available
- increment the live npcs member.

C-2-3 deallocate()

- Calculate index from pointer address
- Push index back to `free_indices`
- If tracing, print deallocation message
- decrement live npcs member.

C-2-4 get_slot() - Use index-based calculation

```
//the pointer of the memory block, offset by the index times
 ↪  the _block_size.
```

C-2-5 get_memory_span() - Return safe view

Optional but possibly helpful: return a span of the entire block of memory

```

```

C-2-6 profile()

- Print number of live vs free NPCs
- Print indices of free slots

C-3 Sample Trace Output

Initializing NPCPool with 100 slots (NPC size: 64 bytes)
Live NPCs: 0, Free slots: 100

Allocating NPC at slot 99 (address: 0x7fff5000)
Allocating NPC at slot 98 (address: 0x7fff4fc0)
Deallocating NPC at slot 99 (address: 0x7fff5000)

Live NPCs: 1, Free slots: 99
Free slots: 99, 97, 96, 95, 94...

D Main Driver Program

Feel free to modify this, but this is the minimum requirements–your code must at least work with this *main* function (profile optional but recommended).

```cpp
#include <iostream>
#include <vector>
#include "NPC.h"

int main() {
    // Create several NPCs
    std::vector<NPC*> npcs;

    // Spawn a wave of enemies
    for (int i = 0; i < 10; ++i) {
        npcs.push_back(new NPC(
            "Goblin_" + std::to_string(i),
            "enemy",
            i * 10, i * 10,  // Position
            50 + i * 5       // Health
        ));
    }

    // Show one NPC
    std::cout << *npcs[5] << std::endl;

    // Delete some NPCs (simulating death)
    delete npcs[3];
    delete npcs[5];
    npcs[3] = nullptr;
    npcs[5] = nullptr;

    // Profile the pool
```

```
    NPC::profile();

    // Spawn replacements
    npcs[3] = new NPC("Dragon_Boss", "boss", 50, 50, 500);
    npcs[5] = new NPC("Shopkeeper", "merchant", 25, 25, 100);

    // Clean up all NPCs
    for (auto* npc : npcs) {
        delete npc;
    }

    // Final profile
    NPC::profile();

    return 0;
}
```

E Bonus Questions (10 points each)

E-1 Bonus 1: Template Pool Class

Convert Pool to a template class `Pool<T>` that can work with any type:

- Template the class appropriately
- Ensure proper alignment using `alignof(T)`
- Update NPC to use `Pool<NPC>` instead of just the non-templated Pool
- Test with at least one other class (e.g., `StockOrder`) (make it up, but keep it no more complex than NPC).

E-2 Bonus 2: Factory Pattern

Add a factory method to NPC that ensures all NPCs are heap-allocated:

- Make the constructor private
- Add: `static NPC* spawn(const std::string& name, const std::string& typ`
- Prevent stack allocation completely
- Update main.cpp to use the factory method

F Grading Rubric

F-1 Core Requirements

- If your project doesn't build in OnlineGDB and you provide no detailed explanation, I will return it to you to fix.
- **NPC Class Implementation** (20 points)
  - Proper data members and constructor (5)
  - Output operator (5)
  - operator new/delete overrides (10)
- **Pool Implementation** (40 points)
  - Proper memory allocation (15)
  - Vector-based free list management (10)
  - Index-based slot calculation (5)
  - std::span usage (5)
  - Tracing functionality (5)

- **Memory Management** (20 points)
  - No memory leaks (10)
  - Proper RAII in destructor (5)
  - Exception safety with std::bad_alloc (5)

F-2 Code Quality (20 points)

- Clear, readable code with appropriate comments
- Use of modern C++ features (C++20/23)
- No unnecessary copying or dynamic allocations

G Submission Requirements

Submit an OnlineGDB link with your project that builds and runs AND submit the files below.
Submit the following files:

- `NPC.h` and `NPC.cpp`
- `Pool.h` and `Pool.cpp`
- `main.cpp`

Ensure your code compiles with C++20:

H Hints and Tips

1. **Start Simple**: Get basic allocation working before adding tracing
2. **Test Incrementally**: Test allocate() before implementing deallocate()
3. **Use Debugger**: Set breakpoints to watch memory allocation
4. **Vector Efficiency**: `free_indices.pop_back()` is O(1) - use it for allocation