Object-oriented Programming

Assignment 007:
STL allocator + memory pool

Author: 刘仁钦

Date: 2024/12/25

Table of Contents

1. Overview

This project implements a custom allocator to replace the default allocator `std::Allocator<T>`. And the allocator uses memory pool to speed up the dynamic allocation of a large number of small blocks and to reduce memory fragmentation. This allocator can used with some STL containers like `std::vector` and `std::map`.

2. File Structure

```
.
├── build
├── CMakeLists.txt
├── README.md
├── script
│   ├── compile.sh
│   ├── package.sh
│   └── test.sh
├── src
│   └── MemoryPoolAllocator.tpp
└── test
    └── tests.cpp
```

- **`src/`**: Contains the implementation file.
- **`test/`**: Contains test code.
- **`script/`**: Contains auxiliary scripts for packaging and running tests.

3. Environment Requirements

- CMake 3.5.0 or higher
- A C++ compiler support C++11 standards

4. Compile and Test

Run `script/compile.sh` for compilation. Run `script/test.sh` for test.

5. Class Design

The class `MemoryPoolAllocator` is designed to replace the default allocator `std::Allocator<T>`. It uses a memory pool to speed up the dynamic allocation of a large number of small blocks and reduce memory fragmentation. The class provides the following member functions:

1. Constructor and Destructor

- **`MemoryPoolAllocator()`** Initializes an empty free block list for the allocator.

- **`~MemoryPoolAllocator()`** Frees all remaining memory blocks in the free block list, ensuring no memory leaks.

2. Address Functions

- **`pointer address(reference _Val) const noexcept`** Returns the address of a given reference `_Val`.
- **`const_pointer address(const_reference _Val) const noexcept`** Returns the address of a constant reference `_Val`.

3. Memory Management

- **`void deallocate(pointer address, size_type count)`** Deallocates a block of memory and adds it to the free block list for reuse.
- **`pointer allocate(size_type count)`** Allocates a block of memory large enough to hold `count` elements. It reuses a free block if available; otherwise, it creates a new block.

4. Object Construction and Destruction

- **`template<class U> void destroy(U* ptr)`** Destroys an object at the given pointer `ptr` by calling its destructor.
- **`template<class Obj, class... Args> void construct(Obj* ptr, Args&&... args)`** Constructs an object of type `Obj` at the given pointer `ptr` using the provided arguments `args`.

5. Comparison Operators

- **`bool operator==(const MemoryPoolAllocator& other) const`** Compares two allocators for equality. Returns `true` only if they are the same instance.
- **`bool operator!=(const MemoryPoolAllocator& other) const`** Compares two allocators for inequality. Returns `true` if they are different instances.

6. Private Block Structure

The allocator manages memory using a linked list of `Block` objects, defined as follows:

- **Members**:
  - `size_t size`: The size of the block (number of elements).
  - `value_type* data`: Pointer to the block's data.
  - `Block* next`: Pointer to the next block in the list.

- **Constructors**:
  - `Block()`: Creates an empty block with `size = 0` and `data = nullptr`.
  - `Block(size_type size)`: Allocates a block of memory large enough to hold `size` elements.
- **Destructor**: Frees the allocated memory for the block.
- **`Block* split(size_type size)`**: Splits a large block into two smaller blocks, keeping the first part of the requested size and returning a pointer to the remaining block.

The private field `Block* free_blocks` is the head of a list of blocks that are available for reuse. As is shown in the figure below:

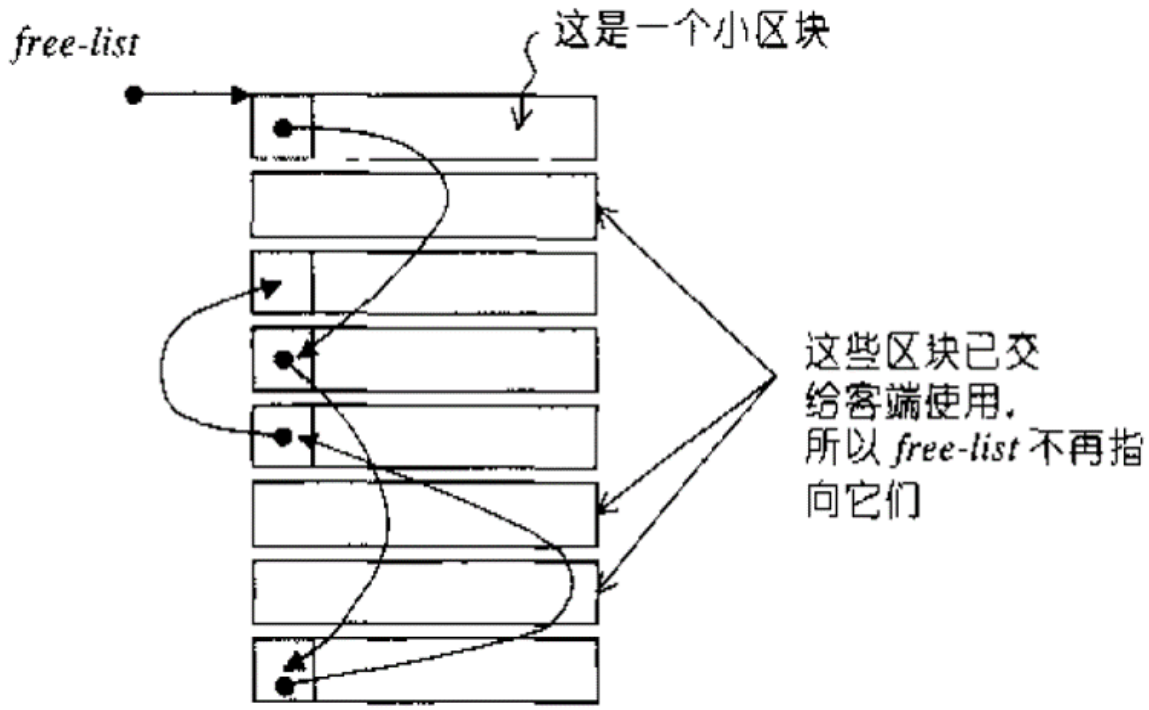

Figure 1: Free list structure

6. Code Implementation

Here is the code inplementation of this assignment, in the class `MemoryPoolAllocator` in the file `src/MemoryPoolAllocator.tpp`

```
#ifndef MEMORYPOOLALLOCATOR_TPP
#define MEMORYPOOLALLOCATOR_TPP

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

```cpp
#include <cstdlib>
#include <limits>

/**
 * @class MemoryPoolAllocator
 * @brief A custom memory allocator that uses a memory pool for efficient
allocation and deallocation.
 *
 * This allocator manages memory in blocks and reuses free blocks to minimize
overhead.
 * It provides basic functionalities required by STL allocators, such as
allocate, deallocate,
 * construct, and destroy.
 *
 * @tparam T The type of object to allocate.
 */
template<class T>
class MemoryPoolAllocator {
public:
    /**
     * @brief Minimum block size for allocation.
     */
#define MIN_BLOCK_SIZE 1024

    using value_type = T;
    using size_type = size_t;
    using difference_type = ptrdiff_t;
    using pointer = value_type*;
    using const_pointer = const value_type*;
    using reference = value_type&;
    using const_reference = const value_type&;

    /**
     * @brief Default constructor. Initializes an empty free block list.
     */
    MemoryPoolAllocator() {
        free_blocks = nullptr;
    }


    /**
     * @brief Destructor. Frees all remaining blocks in the free list.
     */
    ~MemoryPoolAllocator() {
```

```
    while (free_blocks != nullptr) {
        Block* temp = free_blocks;
        free_blocks = free_blocks->next;
        delete temp;
    }
}

/**
 * @brief Returns the address of a reference.
 *
 * @param _Val The reference to the object.
 * @return The pointer to the object.
 */
pointer address(reference _Val) const noexcept {
    return &(_Val);
}

/**
 * @brief Returns the address of a constant reference.
 *
 * @param _Val The constant reference to the object.
 * @return The pointer to the object.
 */
const_pointer address(const_reference _Val) const noexcept {
    return &(_Val);
}

/**
 * @brief Deallocates a block of memory and returns it to the free list.
 *
 * @param address The pointer to the memory block.
 * @param count The number of elements in the block.
 */
void deallocate(pointer address, size_type count) {
    Block* block = new Block();
    block->data = address;
    block->size = count;
    block->next = free_blocks;
    free_blocks = block;
}

/**
 * @brief Allocates a block of memory. Reuses a free block if available.
 *
```

```
     * @param count The number of elements to allocate.
     * @return A pointer to the allocated memory block.
     */
    pointer allocate(size_type count) {
        if (count < MIN_BLOCK_SIZE) {
            count = MIN_BLOCK_SIZE;
        }

        // Find the first free block that is big enough
        Block** current = &free_blocks;
        while (*current != nullptr) {
            Block* block = *current;
            if (block->size >= count) {
                // Split the block if it is too large
                if (block->size > count) {
                    *current = block->split(count);
                }
                // Remove the block from the free list
                *current = block->next;
                return block->data;
            }
            current = &block->next;
        }

        // If no suitable block is found, allocate a new one
        Block* new_block = new Block(count);
        return new_block->data;
    }

    /**
     * @brief Destroys an object at a given pointer by invoking its
destructor.
     *
     * @tparam U The type of the object.
     * @param ptr The pointer to the object to destroy.
     */
    template<class U>
    void destroy(U* ptr) {
        if (ptr != nullptr) {
            ptr->~U();
        }
    }

    /**
```

```
     * @brief Constructs an object at a given pointer with the provided
arguments.
     *
     * @tparam Obj The type of the object to construct.
     * @tparam Args The types of arguments for the constructor.
     * @param ptr The pointer to the memory location.
     * @param args The arguments to pass to the constructor.
     */
    template<class Obj, class... Args>
    void construct(Obj* ptr, Args&&... args) {
        new (ptr) Obj(std::forward<Args>(args)...);
    }

    /**
     * @brief Compares two allocators for equality. Always returns false for
different instances.
     *
     * @param other Another allocator to compare.
     * @return True if the allocators are the same instance, false otherwise.
     */
    bool operator==(const MemoryPoolAllocator& other) const {
        return this == &other;
    }

    /**
     * @brief Compares two allocators for inequality.
     *
     * @param other Another allocator to compare.
     * @return True if the allocators are different instances, false
otherwise.
     */
    bool operator!=(const MemoryPoolAllocator& other) const {
        return this != &other;
    }

private:
    /**
     * @struct Block
     * @brief Represents a memory block in the allocator.
     */
    struct Block {
        size_t size;        ///< Size of the block (number of elements).
        value_type* data;   ///< Pointer to the block's data.
        Block* next;        ///< Pointer to the next block in the list.
```

```
        /**
         * @brief Default constructor. Creates an empty block.
         */
        Block() {
            this->size = 0;
            this->data = nullptr;
            this->next = nullptr;
        }

        /**
         * @brief Constructor. Allocates a block of the specified size.
         *
         * @param size The number of elements to allocate.
         */
        Block(size_type size) {
            this->size = size;
            this->data = static_cast<value_type*>(std::malloc(size *
sizeof(value_type)));
            this->next = nullptr;
        }

        /**
         * @brief Destructor. Frees the allocated memory.
         */
        ~Block() {
            std::free(data);
        }

        /**
         * @brief Splits the block into two blocks if it is larger than the
requested size.
         *
         * @param size The size of the first block after splitting.
         * @return A pointer to the remaining part of the block.
         */
        Block* split(size_type size) {
            Block* new_block = new Block();
            new_block->size = this->size - size;
            new_block->data = &this->data[size];
            new_block->next = this->next;
            this->size = size;
            this->next = new_block;
            return this;
```

```
        }
    };

    Block* free_blocks; ///< Pointer to the head of the free block list.
};

#endif // MEMORYPOOLALLOCATOR_CPP
```

7. Test Cases and Results

I use the provided test using `std::vector`, and designed my extra test using `std::map` as a bonus.

Output of test program `build/tests`:

```
Testing vector creation
Testing vector resize
Testing vector assignment
correct assignment in vecints: 3164
correct assignment in vecpts: 5605
Testing vector destruction
Testing map creation and insertion
Testing map assignment
correct assignment in mapints: 608
correct assignment in mappts: 3902
Testing map destruction
Testing complete
```

And the screenshot of the test program output is below:

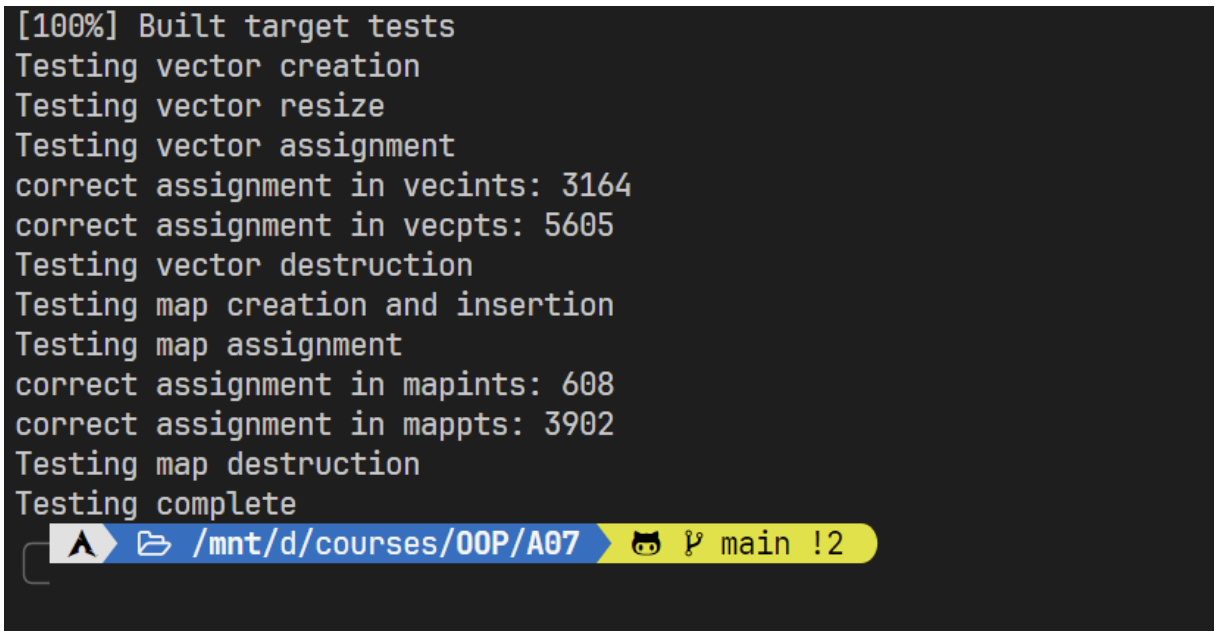

Figure 2: Test program output

Index	STL Container	Operation	Status
1	`std::vector`	creation	passed
2	`std::vector`	resize	passed
3	`std::vector`	assignment	passed
4	`std::vector`	destruction	passed
5	`std::map`	creation and insertion	passed
6	`std::map`	assignment	passed
7	`std::map`	destruction	passed

8. Test Code

The test code is in the file `test/tests.cpp`, and is as follows:

```
#include <iostream>
#include <random>
#include <vector>
#include <list>
#include <map>
#include <deque>
#include "MemoryPoolAllocator.tpp"

// include header of your allocator here
template<class T>
using MyAllocator = MemoryPoolAllocator<T>;

using Point2D = std::pair<int, int>;

const int TestSize = 10000;
const int PickSize = 1000;

int main()
{
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(1, TestSize);

    std::cout << "Testing vector creation" << std::endl;
    using IntVec = std::vector<int, MyAllocator<int>>;
    std::vector<IntVec, MyAllocator<IntVec>> vecints(TestSize);
    for (int i = 0; i < TestSize; i++)
        vecints[i].resize(dis(gen));
```

```
    using PointVec = std::vector<Point2D, MyAllocator<Point2D>>;
    std::vector<PointVec, MyAllocator<PointVec>> vecpts(TestSize);
    for (int i = 0; i < TestSize; i++)
        vecpts[i].resize(dis(gen));

    std::cout << "Testing vector resize" << std::endl;
    for (int i = 0; i < PickSize; i++) {
        int idx = dis(gen) - 1;
        int size = dis(gen);
        vecints[idx].resize(size);
        vecpts[idx].resize(size);
    }

    std::cout << "Testing vector assignment" << std::endl;
    {
        int val = 10;
        int idx1 = dis(gen) - 1;
        int idx2 = vecints[idx1].size() / 2;
        vecints[idx1][idx2] = val;
        if (vecints[idx1][idx2] == val)
            std::cout << "correct assignment in vecints: " << idx1 <<
std::endl;
        else
            std::cout << "incorrect assignment in vecints: " << idx1 <<
std::endl;
    }
    {
        Point2D val(11, 15);
        int idx1 = dis(gen) - 1;
        int idx2 = vecpts[idx1].size() / 2;
        vecpts[idx1][idx2] = val;
        if (vecpts[idx1][idx2] == val)
            std::cout << "correct assignment in vecpts: " << idx1 <<
std::endl;
        else
            std::cout << "incorrect assignment in vecpts: " << idx1 <<
std::endl;
    }

    std::cout << "Testing vector destruction" << std::endl;
    vecints.clear();
    vecpts.clear();

    std::cout << "Testing map creation and insertion" << std::endl;
```

```
    using IntMap = std::map<int, int, std::less<int>,
MyAllocator<std::pair<const int, int>>>;
    std::vector<IntMap, MyAllocator<IntMap>> mapints(TestSize);
    using PointMap = std::map<int, Point2D, std::less<int>,
MyAllocator<std::pair<const int, Point2D>>>;
    std::vector<PointMap, MyAllocator<PointMap>> mappts(TestSize);
    for (int i = 0; i < PickSize; i++) {
        int idx = dis(gen) - 1;
        mapints[idx].insert({ i, dis(gen) });
        mappts[idx].insert({ i, {dis(gen), dis(gen)} });
    }

    std::cout << "Testing map assignment" << std::endl;
    {
        int val = 10;
        int idx1 = dis(gen) - 1;
        int idx2 = mapints[idx1].size() / 2;
        mapints[idx1][idx2] = val;
        if (mapints[idx1][idx2] == val)
            std::cout << "correct assignment in mapints: " << idx1 <<
std::endl;
        else
            std::cout << "incorrect assignment in mapints: " << idx1 <<
std::endl;
    }
    {
        Point2D val(24, 67656);
        int idx1 = dis(gen) - 1;
        int idx2 = mappts[idx1].size() / 2;
        mappts[idx1][idx2] = val;
        if (mappts[idx1][idx2] == val)
            std::cout << "correct assignment in mappts: " << idx1 <<
std::endl;
        else
            std::cout << "incorrect assignment in mappts: " << idx1 <<
std::endl;
    }

    std::cout << "Testing map destruction" << std::endl;
    mapints.clear();
    mappts.clear();

    std::cout << "Testing complete" << std::endl;
```

```
    return 0;
}
```