OOP Lab7 Report

1 实验概述

1.1 实验目的

- 编写一个分配器，兼容 `std::allocate` 的相关接口，使其可以被 `std::vector` 等容器使用
- 优化分配器的性能

1.2 实验环境

- 操作系统：macOS 15.3
- 编译器：Apple clang version 16.0.0
- 编译参数：`-g -O0`
- CPU：Apple M4

2 实验过程

2.1 类模板定义

首先我们需要给出我们的类模板定义。根据 cppreference 上对 allocator 的描述，我们的分配器的类模板 `mAllocator` 对外暴露的类型与接口函数如下代码所示：

```
template<class _Tp, size_t BlockSize = 4096>
class mAllocator {
public:
    using _Not_user_specialized = void;
    using value_type = _Tp;
    using pointer = value_type*;
    using const_pointer = const value_type*;
    using reference = value_type&;
    using const_reference = const value_type&;
    using size_type = std::size_t;
    using difference_type = std::ptrdiff_t;
    using propagate_on_container_move_assignment = std::true_type;
    using is_always_equal = std::true_type;

    template <class _Up> struct rebind {
        using other = mAllocator<_Up, BlockSize>;
    };

    mAllocator() noexcept;
    mAllocator(const mAllocator& mallocator) noexcept;
    template<class _Up> mAllocator(const mAllocator<_Up>& other) noexcept;
    ~mAllocator();

    pointer address(reference x) const noexcept;
    const_pointer address(const_reference x) const noexcept;
    pointer allocate(size_type n, const _Not_user_specialized* hint = 0);
    void deallocate(pointer p, size_type n);
    size_type max_size() const noexcept;
```

```
29
30      template<class _Up, class... Args>
31      void construct(_Up* p, Args&&... args);
32
33      template<class _Up>
34      void destroy(_Up* p);
35  }
```

模板参数 `_Tp` 和 `BlockSize` 分别指定了分配器实例针对的数据类型，以及分配器在底层一次申请的内存大小。

为了使我们的分配器可以被 `std::vector` 等容器使用，我们需要实现以下接口函数：

- `mAllocator() noexcept`：构造函数
- `mAllocator(const mAllocator& mallocator) noexcept`：拷贝构造函数
  - 拷贝构造函数无需任何操作，因为分配器是全局性的，内部变量均为 `static` 类型
- `~mAllocator()`：析构函数
- `pointer address(reference x) const noexcept`：获取引用 `x` 的地址
- `pointer allocate(size_type n, const _Not_user_specialized* hint = 0)`：为 n 个类型为 `_Tp` 的数据分配空间，但不进行构造操作
- `void deallocate(pointer p, size_type n)`：释放 `n` 个类型为 `_Tp` 的数据的空间，释放前不进行析构操作
- `template<class _Up, class... Args> void construct(_Up* p, Args&&... args)`：为 `p` 处的数据进行 `_Up` 类型的构造，参数为 `args`
- `template<class _Up> void destroy(_Up* p)`：为 `p` 处的数据进行 `_Up` 类型的析构

2.2 分配操作的实现

在调用 `allocate` 进行内存分配时，我们可以通过 `n * sizeof(_Tp)` 算出申请的总内存空间。对于不同大小的内存申请，我们采取不同的策略：

- 对于总申请空间大于 128 Byte 的情况，我们直接调用 `malloc` 进行空间分配，并返回 `malloc` 返回的指针。
- 对于总申请空间小于 128 Byte 的情况，我们根据申请空间大小从对应的 `free_slot_list` 中获得可用的 `slot`
  - 若 `free_slot_list` 为空，则从内存池中分配一段连续空间，接入 `free_slot_list`
  - 若内存池空间不足，则调用 `malloc`，以 `BlockSize` 为单位进行空间分配，使用新申请的空间作为新的内存池

下面我们来具体实现以上策略：

2.2.1 数据结构

对于总申请空间小于 128 Byte 的情况，我们使用 slot 作为分配的单元。slot 的结构如下所示：

```
union slot_t {
    slot_t* next;
    _Tp data[0];
};
```

其中，next 用于 free_slot_list 中链表的构建，data 用于存储实际的数据。我们使用联合体来表示 slot，是因为一个 slot 要么在空闲链表中，要么已经被分配，因此 next 和 data 不会被同时使用，所以我们可以将他们存储在同一内存地址。

为了方便管理，我们将 slot 的大小限定为不大于 128 的范围内的 8 的倍数：

```
constexpr static int SLOT_SIZE_NUM = 16;
constexpr static int SLOT_SIZE_AVAILABLE[SLOT_SIZE_NUM] = {
    8, 16, 24, 32, 40, 48, 56, 64,
    72, 80, 88, 96, 104, 112, 120, 128
};
```

这样我们便可以直接通过申请空间 size 来确定其所属的 free_slot_list，其映射关系在函数 get_slot_index 中定义：

```
int get_slot_index(size_t size) {
    return size > 128 ? -1 : (size - 1) >> 3;
}
```

在此基础上，我们可以定义 mAllocator 需要的数据结构：

```
static slot_t* free_slot_list[SLOT_SIZE_NUM];   // 空闲链表组
static uint8_t* memory_pool_start;              // 内存池的起始地址
static uint8_t* memory_pool_end;                // 内存池的结束地址
```

2.3 分配操作

我们先实现 allocate 函数的上层逻辑，即根据申请空间的大小选择不同的分配方式：

```
template<typename _Tp, size_t BlockSize>
typename mAllocator<_Tp, BlockSize>::pointer mAllocator<_Tp, BlockSize>::allocate(size_type n, const
_Not_user_specialized* hint) {
    int alloc_size = n * sizeof(value_type);
    int index = get_slot_index(alloc_size);
    if (index == -1) {
        return reinterpret_cast<pointer>(MallocAllocator::allocate(alloc_size));
    } else {
        if (free_slot_list[index] == nullptr) {
            extend_free_slot_list(alloc_size);
        }
        slot_t* slot = free_slot_list[index];
        free_slot_list[index] = slot->next;
        return slot->data;
    }
```

```
}
```

我们先根据 n 和 _Tp 计算所需的空间大小，然后调用 get_slot_index 得到其对应的 free_slot_list 的编号。若编号为 -1，说明大于 128 Byte，直接调用 MallocAllocator::allocate 进行分配（实际上是 malloc 的简单封装）；若编号不为 -1，则优先从 free_slot_list[index] 中获取可用 slot，若没有，则需要调用 extend_free_slot_list 延长空闲槽链表。

接着我们来看 extend_free_slot_list：这个函数需要从内存池中获取一定大小的空间，并插入到空闲槽链表中。鉴于这个函数被调用的次数可能很多，我们采用一个 prefetch 的策略：每次调用 extend_free_slot_list 时，向空闲槽链表中插入 DEFAULT_ALLOC_SLOT 个大小与申请空间大小相同的 slot，以避免该函数被频繁调用。

```
template<class _Tp, size_t BlockSize>
void* mAllocator<_Tp, BlockSize>::extend_free_slot_list(size_type size) {
    int count = DEFAULT_ALLOC_SLOT;
    int slot_index = get_slot_index(size);
    int alloc_size = SLOT_SIZE_AVAILABLE[slot_index];
    if (slot_index == -1) {
        return nullptr;
    }

    slot_t* slot = reinterpret_cast<slot_t*>(alloc_slot_mempool(alloc_size, count));
    for (int i = 0; i < count; i++) {
        slot->next = free_slot_list[slot_index];
        free_slot_list[slot_index] = slot;
        slot = reinterpret_cast<slot_t*>(reinterpret_cast<uint8_t*>(slot) + alloc_size);
    }
    return free_slot_list[slot_index];
}
```

extend_free_slot_list 需要向内存池申请空间，而这个操作由 alloc_slot_mempool 函数实现：该函数会先通过 memory_pool_start 和 memory_pool_end 计算出内存池的剩余大小，与所需的空间大小进行比较。若足够，则直接从内存池中分配空间，并更新 memory_pool_start；若不足，则需要申请一块新的内存池。

```
template<class _Tp, size_t BlockSize>
void* mAllocator<_Tp, BlockSize>::alloc_slot_mempool(size_type size, int count) {
    size_type alloc_size = size * count;
    if (memory_pool_start + alloc_size > memory_pool_end) {
        inflate_mempool(alloc_size);
    }

    slot_t* slot = reinterpret_cast<slot_t*>(memory_pool_start);
    memory_pool_start = reinterpret_cast<uint8_t*>(memory_pool_start) + alloc_size;
    return slot;
}
```

申请新的内存池的操作由 inflate_mempool 函数实现。该函数直接调用 malloc 申请对应大小的内存作为内存池，申请的空间大小为模板参数 BlockSize 的倍数，最后更新 memory_pool_start 和 memory_pool_end。

```cpp
template<class _Tp, size_t BlockSize>
void mAllocator<_Tp, BlockSize>::inflate_mempool(size_type size) {
    size_type alloc_size = (size + BLOCK_SIZE - 1) / BLOCK_SIZE * BLOCK_SIZE;

    memory_pool_start = static_cast<uint8_t*>(malloc(alloc_size));
    if (!memory_pool_start) {
        memory_pool_end = nullptr;
        throw std::bad_alloc();
    }

    memory_pool_end = memory_pool_start + alloc_size;
}
```

借助上述函数，我们便实现了 `allocate` 操作。对于 `deallocate`，我们同样需要根据空间大小进行不同的操作：

- 如果待释放的空间大于 128 Byte，则通过 `MallocAllocator::deallocate` 进行释放（实际上是 `free` 的简单封装）
- 如果待释放的空间不大于 128 Byte，则将这块空间所属的 `slot` 加入对应的 `free_slot_list` 中，实现空间的回收

```cpp
template<typename _Tp, size_t BlockSize>
void mAllocator<_Tp, BlockSize>::deallocate(pointer p, size_type n) {
    if (p == nullptr) {
        throw std::bad_alloc();
    }

    int free_size = n * sizeof(value_type);
    int index = get_slot_index(free_size);
    if (index == -1) {
        MallocAllocator::deallocate(static_cast<void*>(p));
    } else {
        slot_t* slot = reinterpret_cast<slot_t*>(p);
        slot->next = free_slot_list[index];
        free_slot_list[index] = slot;
    }
}
```

除了 `allocate` 和 `deallocate` 之外，其他的函数实现较为简单，此处略。

3 实验结果

3.1 时间测量

我们将 c++ 的 `chrono` 库中的计时器封装到一个类 `Timer` 中，其在构造时开始计时，在析构时停止计时并输出计时结果：

```cpp
class Timer {
private:
    std::chrono::high_resolution_clock::time_point start_time;
    std::chrono::high_resolution_clock::time_point end_time;
    std::chrono::duration<double, std::milli> duration;

```

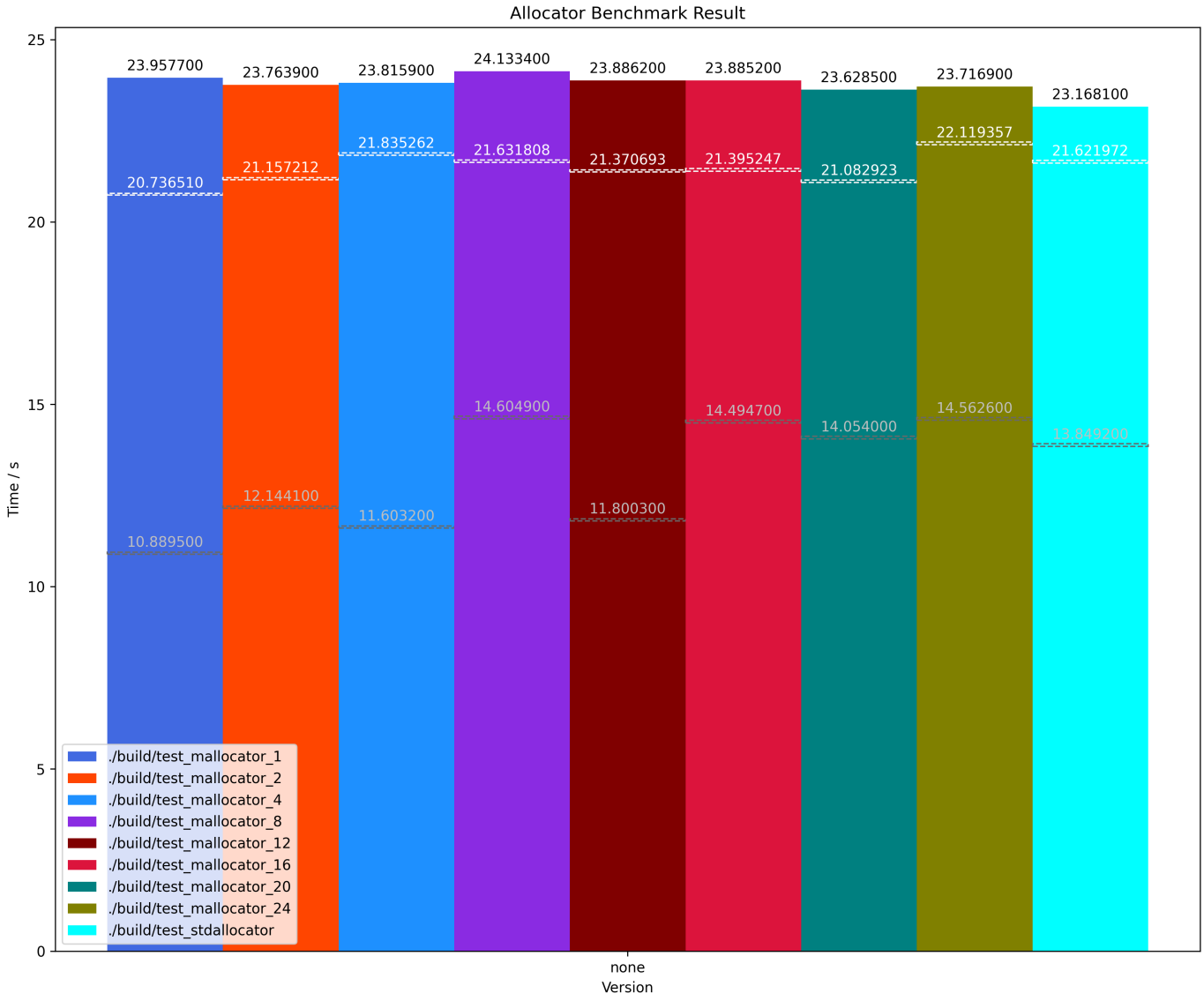
```
public:
    Timer() {
        start_time = std::chrono::high_resolution_clock::now();
    }
    ~Timer() {
        end_time = std::chrono::high_resolution_clock::now();
        duration = end_time - start_time;
        std::cout << "Time elapsed: " << duration.count() << " ms" << std::endl;
    }
};
```

我们在测试代码 `test_allocator.cpp` 的 `testVec = new vecWrapper*[TESTSIZE];` 语句之后实例化了一个 `Timer` 开始计时，在程序结束时 `Timer` 自动析构并输出程序耗时。

3.2 测试

我们编写了自动化测试脚本 `autobench.py`，其可以多次运行测试程序，自动获取程序运行的最长时间、平均时间和最短时间，并绘制结果图。

我们的实现中，`extend_free_slot_list` 函数中的预取参数 `DEFAULT_ALLOC_SLOT` 可以通过宏定义进行调整。我们选取了 $1, 2, 4, 8, 12, 16, 20, 24$ 这八组数据进行测试。重复运行 40 次，测试结果如下图所示：



P.S. 灰色线表示最短时间，白色线代表平均时间，实心部分代表最长时间

从图上可以看出，我们可以看到 `DEFAULT_ALLOC_SLOT` 取值为 1, 2, 12, 16, 20 时，其速度均超过了使用标准库 `std::allocator` 的版本。其中平均耗时最短为 20.736410 ms，相对标准库的加速比为 1.0427。最快耗时最短为 10.889500 ms，相对标准库的加速比为 1.2718。