STL Allocator + Memory Pool Report

1 Description

内存池是一种常见的内存分配方式，它可以减少内存分配的次数，从而提高程序的运行效率。本次实验要求实现一个内存池，然后将其与 STL 中的 `std::allocator` 进行比较，从而验证内存池的有效性以及性能是否提升。

2 Idea

2.1 Allocator

Allocator 是 STL 中的一个重要组成部分，它负责 STL 容器的内存分配。STL 中的容器在进行内存分配时，都会调用 Allocator 的 `allocate` 函数来分配内存，然后调用 `deallocate` 函数来释放内存。STL Allocator 的实现方式是通过 `new` 和 `delete` 来进行内存分配和释放的，这种方式的缺点是频繁的内存分配和释放会导致内存碎片的产生，从而降低程序的运行效率。

在我们的实现中，我们的 Allocator 沿用了 STL Allocator 的接口，但是内部的实现方式不同。我们的 Allocator 会在内存池中分配内存，从而减少内存碎片的产生，提高程序的运行效率。

2.2 Memory Pool

我们实现 Memory Pool 的方法是在内存池中维护一个链表，链表中的每个节点都是一个内存块，每个内存块的大小都是固定的。当 Allocator 需要分配内存时，我们会将链表指向的首个内存块取出，然后将其分配给 Allocator, 并让链表指向下一个空闲块。当 Allocator 需要释放内存时，我们会将其归还给内存池。

特别的是，每个内存块的大小不完全相同，而是为 16, 32, 48, 64, 80, 96, 112, 128, 144, 160, 176, 192, 208, 224, 240, 256 这样的序列。我们这里基于 fastbin 的思想，以将内存块的大小分为 16 个桶，每个桶的大小都是 16 的倍数，这样的好处是可以减少内存碎片的产生。

3 How Program works

main.cpp 这里仅提供入口，以此调用我们下面提到的两个测试函数。

我们这里有两个测试文件, correctness_test.cpp 和 performance_test.cpp. 其中 main.cpp 是一个简单的测试文件，用于测试 Allocator 的正确性。performance_test.cpp 是一个性能测试文件，用于测试 Allocator 的性能。

运行 correctness_test 时，我们会我们会随机生成 TestSize 个 vector，每个 vector 的大小为 1~TestSize 之间的随机数。随后，我们会随机选择 PickSize 个 vector，对它们进行 resize 操作，resize 的大小也是 1~TestSize 之间的随机数。最后，我们会修改 vector 中的值，并且检查是否修改成功，同时输出修改成功的 vector 的编号。

运行 performance_test 时，我们会运行 `performance_test()` 函数，我们会声明 `list<int, allocator<int>>` 和 `list<int, Allocator<int>>` 两个 list 容器，分别使用 STL 默认分配器和手写的内存池和分配器。随后我们会对两个容器进行 1000 次的 10000 个元素的插入和删除操作，最后输出两个容器的运行时间。通过比较两个容器的时间，测试我们手写的内存池和分配器的性能。

4 Explanation of Source Code

这里我们主要介绍内存池的代码。

- allocate
  - `allocate` 函数用于分配内存，其参数为所需内存的大小。对于给定大小，我们找到其在 free_list 中对应的桶，然后取出这个桶的第一个内存块，将其分配给 Allocator。如果这个桶为空，我们会调用 `refill` 函数，从而将这个桶填满。
    这里如果超过了内存池中的最大内存块大小，我们会直接调用 `malloc` 函数，从系统中分配内存，相当于通过一级分配器分配内存。

```
void *MemoryPool::allocate(size_t n) noexcept
{
    if (n > MAX_BLOCK_SIZE) return malloc(n * siz
eof(char));        // 超过了 block 的最大大小
    auto index = FREELIST_INDEX(n);     // 根据大小
找到对应的 freelist
    auto now_node = free_list[index];
    // 如果为空，说明当前的 freelist 已经满了，需要再分
配。
    if (now_node == nullptr) refill(index);
    // 将 now_list 指向下一个空闲的块。
    now_node = free_list[index];
    free_list[index] = now_node->free_list_next;
    return now_node;
}
```

  - 在 `refill` 函数中，我们会将内存池中的剩余可用内存分配给这个桶（默认一次分配 16 块），将块用 free_list 指向的链表串起来。如果

内存池中的剩余可用内存不足，我们会调用 `chunk_alloc` 函数，从系统中申请更多的内存。

```
void MemoryPool::refill(size_t index)
{
    auto cnt_node = 16;
    auto size = blocksize[index];
    auto free_space = free_ed - free_st; // 计算当前分配的内存池中还剩多少空间
    if (free_space < size)         // 无法分配一个块，需要扩容
        chunk_alloc();
    else if (free_space < cnt_node * size) cnt_node = free_space / size;    // 可以分配，但是不够一个 cnt_node 个块
    for (auto i = 0 ; i < cnt_node; i++) {  // 将内存池中的空间分配给 freelist
        block *tmp = reinterpret_cast<block *>(free_st);
        free_st += size;
        tmp->free_list_next = free_list[index];
        free_list[index] = tmp;
    }
}
```

- 在 `chunk_alloc` 函数中，我们先将内存池中的剩余可用空间分配掉，随后申请一块新的内存池。值得注意的是内存池也是通过链表串起来的，因此每块内存池开头中都有一个指向下一块内存池的指针(即 `POOL`)。随后设置好 `free_st` 和 `free_ed` 指针。

```
void MemoryPool::chunk_alloc()
{
    size_t free_space;
    while (true) {      // 将当前内存池剩下的空间分配给 free_list
        free_space = free_ed - free_st;
        if (free_st == nullptr || free_space <= 0)       // 说明此时还没有分配有效内存
            break;
        // 这里因为我们默认会返回比 size 略大的 block，因此要减一。其实就是在将 size 拆分为若干个不同大小的块。
        auto index = FREELIST_INDEX(free_space) - 1;
        auto tmp = reinterpret_cast<block *>(free
```

```
_st);
        free_st += blocksize[index];
        tmp->free_list_next = free_list[index];
        free_list[index] = tmp;
    }
    auto new_pool = (Pool *)malloc(sizeof(char) *
POOL_SIZE + sizeof(Pool));
    if (new_pool == nullptr) {
        cout << "No memory can be malloc." << end
l;
        return;
    }
    new_pool->next = nullptr;
    if (now_pool != nullptr)
        now_pool->next = new_pool;
    now_pool = new_pool;
    free_st = reinterpret_cast<char *>(new_pool)
+ sizeof(Pool);
    free_ed = reinterpret_cast<char *>(new_pool)
+ sizeof(Pool) + POOL_SIZE;
}
```

- deallocate
  释放内存相对简单。我们只需要将内存块放回对应的 free_list 中即可。如果超过了 block 的最大大小，我们直接调用 `free` 函数释放内存。这是因为超过最大大小后，我们会直接调用 `malloc` 函数分配内存，因此我们也需要调用 `free` 函数释放内存，本质上是交给一级分配器处理。

```
void MemoryPool::deallocate(void *p, size_t n) noexce
pt
{
    if (n > MAX_BLOCK_SIZE) {   // 超过了 block 的最大大
小
        free(p);
        return;
    }
    auto index = FREELIST_INDEX(n);
    auto tmp = reinterpret_cast<block *>(p);
    tmp->free_list_next = free_list[index];
    free_list[index] = tmp;
}
```

- reallocate
  与 deallocate 类似，如果超过了 block 的最大大小那么我们直接调用

`reallocate` 函数。否则我们先将原来的内存块释放，然后重新分配内存。

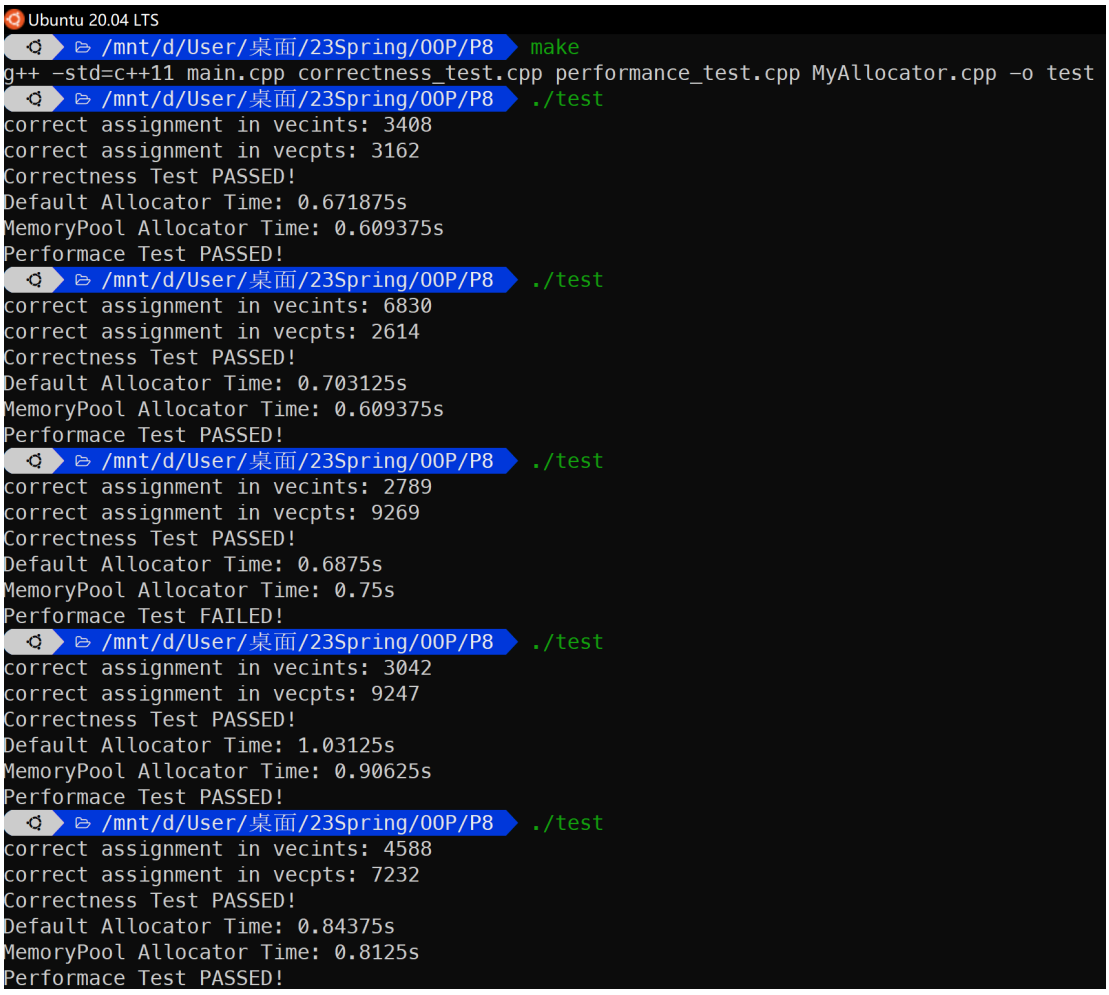
```
void *MemoryPool::reallocate(void *p, size_t old_sz,
size_t new_sz) noexcept
{
    if (old_sz > MAX_BLOCK_SIZE) return realloc(p, ne
w_sz);
    deallocate(p, old_sz);
    return allocate(new_sz);
}
```

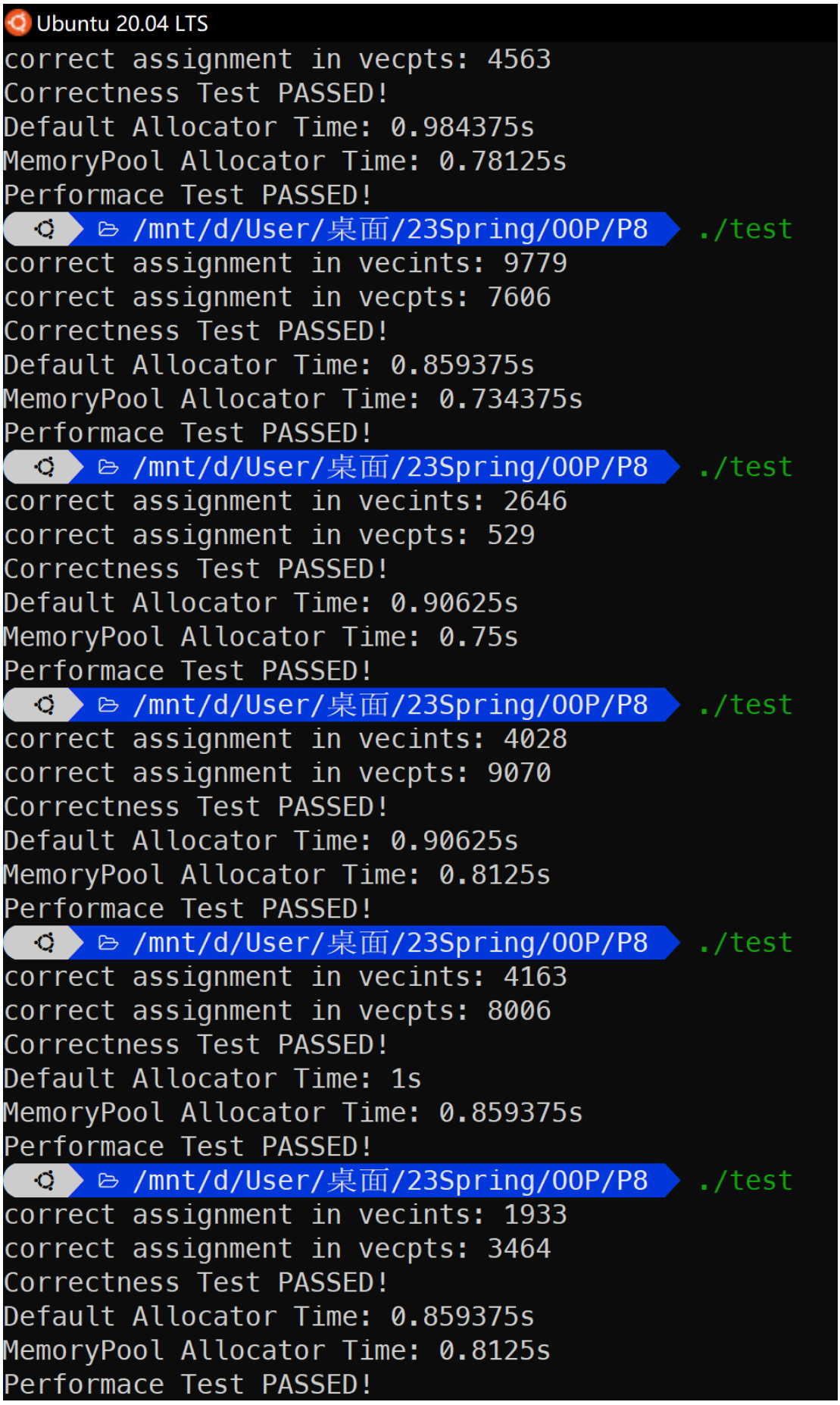
- free_pool
  释放内存池中的所有内存。我们只需要遍历内存池，然后调用 `free` 函数释放内存即可。

```
void MemoryPool::free_pool()
{
    auto tmp = now_pool;
    // 首先释放还没有分配的内存池
    while (tmp != nullptr) {
        auto next = tmp->next;
        free(tmp);
        tmp = next;
    }
    now_pool = nullptr;
    free_st = nullptr;
    free_ed = nullptr;
    // 然后释放 free_list 中的内存
    for (auto i = 0; i < FREELIST_NUM; i++) free_list
[i] = nullptr;
}
```

5 Test Results

测试代码见 correctness_test.cpp 和 performance_test.cpp.


Ubuntu 20.04 LTS
/mnt/d/User/桌面/23Spring/OOP/P8 make
g++ -std=c++11 main.cpp correctness_test.cpp performance_test.cpp MyAllocator.cpp -o test
/mnt/d/User/桌面/23Spring/OOP/P8 ./test
correct assignment in vecints: 3408
correct assignment in vecpts: 3162
Correctness Test PASSED!
Default Allocator Time: 0.671875s
MemoryPool Allocator Time: 0.609375s
Performace Test PASSED!
/mnt/d/User/桌面/23Spring/OOP/P8 ./test
correct assignment in vecints: 6830
correct assignment in vecpts: 2614
Correctness Test PASSED!
Default Allocator Time: 0.703125s
MemoryPool Allocator Time: 0.609375s
Performace Test PASSED!
/mnt/d/User/桌面/23Spring/OOP/P8 ./test
correct assignment in vecints: 2789
correct assignment in vecpts: 9269
Correctness Test PASSED!
Default Allocator Time: 0.6875s
MemoryPool Allocator Time: 0.75s
Performace Test FAILED!
/mnt/d/User/桌面/23Spring/OOP/P8 ./test
correct assignment in vecints: 3042
correct assignment in vecpts: 9247
Correctness Test PASSED!
Default Allocator Time: 1.03125s
MemoryPool Allocator Time: 0.90625s
Performace Test PASSED!
/mnt/d/User/桌面/23Spring/OOP/P8 ./test
correct assignment in vecints: 4588
correct assignment in vecpts: 7232
Correctness Test PASSED!
Default Allocator Time: 0.84375s
MemoryPool Allocator Time: 0.8125s
Performace Test PASSED!


Ubuntu 20.04 LTS
correct assignment in vecpts: 4563
Correctness Test PASSED!
Default Allocator Time: 0.984375s
MemoryPool Allocator Time: 0.78125s
Performace Test PASSED!
/mnt/d/User/桌面/23Spring/OOP/P8 ./test
correct assignment in vecints: 9779
correct assignment in vecpts: 7606
Correctness Test PASSED!
Default Allocator Time: 0.859375s
MemoryPool Allocator Time: 0.734375s
Performace Test PASSED!
/mnt/d/User/桌面/23Spring/OOP/P8 ./test
correct assignment in vecints: 2646
correct assignment in vecpts: 529
Correctness Test PASSED!
Default Allocator Time: 0.90625s
MemoryPool Allocator Time: 0.75s
Performace Test PASSED!
/mnt/d/User/桌面/23Spring/OOP/P8 ./test
correct assignment in vecints: 4028
correct assignment in vecpts: 9070
Correctness Test PASSED!
Default Allocator Time: 0.90625s
MemoryPool Allocator Time: 0.8125s
Performace Test PASSED!
/mnt/d/User/桌面/23Spring/OOP/P8 ./test
correct assignment in vecints: 4163
correct assignment in vecpts: 8006
Correctness Test PASSED!
Default Allocator Time: 1s
MemoryPool Allocator Time: 0.859375s
Performace Test PASSED!
/mnt/d/User/桌面/23Spring/OOP/P8 ./test
correct assignment in vecints: 1933
correct assignment in vecpts: 3464
Correctness Test PASSED!
Default Allocator Time: 0.859375s
MemoryPool Allocator Time: 0.8125s
Performace Test PASSED!

可以看到在 11 次测试中，我们的内存池均通过了正确性测试，且性能测试也仅有一次慢于 STL Allocator 的，这说明我们的内存池的性能是非常优秀的。

6 Discussion

本次的 Project 让我了解了很多关于内存分配的知识，也让我对 C++ 的内存分配机制有了更深的理解。在实现内存池的过程中，我也遇到了一些问题。作为 OOP 的最后一个 Project, 这次的 Project 也让我对 C++ 的面向对象编程有了更深的理解。

时间原因，我们还可以对内存池进行更多的优化，比如内存池的扩容策略、内存池的释放策略等等，这里实现的只是一个简单的内存池。希望以后能更加深入地了解这方面的知识，基于此做进一步的完善。