OOP大作业实验报告

郭永军 3200102126

1. 实验目的

本次实验的主要内容为实现一个memory pool，并以此为基础实现一个自己的allocator。在std中本来是有默认的`std::allocator`的，但是在每一次申请内存的时候由于STL容器用它默认的allocator的话需要不断调用内存管理函数，对于系统的开销是非常大的，特别是在管理小块的内存的时候，会极大地增加系统的开销。因此基于这个事实，我们选择自己实现一个allocator， 这个allocator是基于一个memory pool， 首先提前申请一大块内存，然后再申请小内存的时候直接从这个内存池里面拿内存而不需要再一次向系统申请内存，这极大地减小了申请小内存的开销。

2. 实验方案

2.1 数据结构

2.1.1 memory pool

```
//union freelist_node, when it is in free_list it is a pointer. Otherwise it is an
allocated data.
union Slot_ {
    union Slot_* next;//free_list_pointer
    char client_data[1];
};

static Slot_* free_list[FREELIST_NUM]; //the whole list
static char* start; // free list start
static char* end; //free list end
static std::size_t heap_size;//total size of the list
```

对于memory pool来说，它主要由两部分构成，一部分是已经交付给STL容器使用的内存，而另一部分则是还未被分配的空间。free_list就是未被分配的这一部分的空间。其中值得注意的相关空间的利用，对于union slot_来说，当它位于free_list中的时候，他是作为一个指针指向下一块free的空间，而当它是被分配之后，他就是一个有意义的数据空间，可以用来存放数据，再allocate和deallocate的过程中，需要注意这两个角色之间的转换。

2.1.2 Allocator

```
typedef T value_type;
typedef value_type* pointer;
typedef const value_type* const_pointer;
typedef value_type& reference;
typedef const value_type& const_reference;
typedef std::size_t size_type;
typedef ptrdiff_t difference_type;
typedef std::true_type propagate_on_container_move_assignment;
```

```
typedef std::true_type is_always_equal;
template <typename U>
struct rebind
{
    typedef Allocator<U> Other;
};

pointer address(reference _Val) const _NOEXCEPT
const_pointer address(const_reference _Val) const _NOEXCEPT
void deallocate(pointer _Ptr, size_type _Count)
_DECLSPEC_ALLOCATOR pointer allocate(size_type _Count)
template<class _Uty> void destroy(_Uty *_Ptr)
template<class _Objty, class _Types>
void construct(_Objty *_Ptr, _Types&&... _Args)
```

这一部分是Allocator类的标准定义，规定了一系列标准的接口， 具体可以在std::Allocator中找到详细的说明。在本实验中只实现了一部分的主要功能，包括几种构造函数，address, allocate 以及 deallocate等。Allocator类主要是调用memory_pool来实现内存的分配及管理。

2.2 函数实现（主要为allocate和deallocate）

2.2.1 allocate

```
static void* allocate(std::size_t n) {
        if (n > MAX_BYTES) {
            return malloc(n);//如果申请的空间已经大于最大的空间了，则必然不可能够，因此直接申请这一块大内存
        }
        Slot_** now_fl = free_list + freelist_index(n);
        Slot_* res = *now_fl;
        if (res == nullptr) {//如果剩余的内存不够，则需要继续申请内存
            return fill(round(n));
        }
        *now_fl = res->next;//如果剩余的内存足够，则直接返回这一块内存并更新当前的free_list
        return res;
    }
```

allocate的逻辑是，先判断将要申请的空间是否大于一个固定的空间，如果大于这个空间，我们则认为他是一块大内存，因此直接申请这一块内存。否则再判断当前的容量是否足以提供这样一块内存，如果可以的话则直接提供，如果不行的话则需要首先对容量进行扩充，然后再将返回这一块内存。这一部分功能通过fill来实现，再fill内部，首先为free_list申请到n bytes的空间。然后通过free_list的链表将目前的剩下的block连接成为一个新的free_list然后更相关的参数

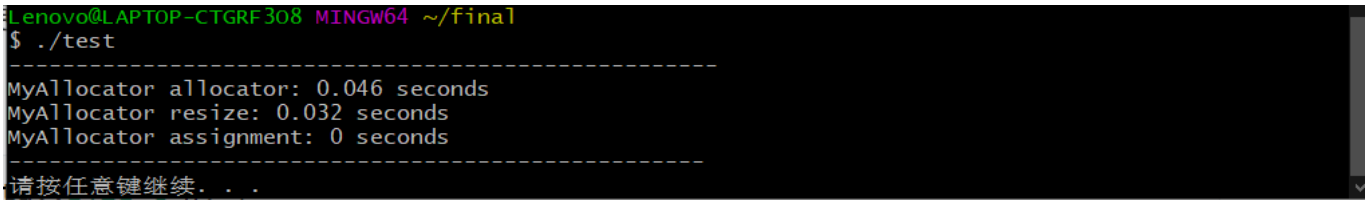
2.2.2 deallocate

```
static void deallocate(void* p, std::size_t n) {
        if (n > MAX_BYTES) {
```

```
            free(p);//if the requested memory is greater than MAX_BYTES, call free
  directly
            return;
        }
        //else put in free list
        Slot_* h = (Slot_*)p;
        Slot_** now_fl = free_list + freelist_index(n);
        h->next = *now_fl;
        *now_fl = h;
    }
```

相对来说deallocate较为简单，如果是大于一定字节的内存，则直接free，否则将这一部分内存连接到当前的free_list之后。

3. 实验结果



由结果可知，相关的操作均正确。但是对于性能的优化相对于原本的allocator只有小部分的提升，这可能是后面实验需要继续突破的部分。相关的代码可以在压缩包中的源文件中查看。