OOP project: STL allocator + memory pool

作者

鲍奕帆 3180103499

朱亦陈

内存池结构设计

基本思想

本实验采用动态内存池实现。向系统申请一块较大的内存，一般为1024的整数倍。再根据需要，不断的从这一块内存划分出小的chunk(chunk的大小根据分配的内存大小可变)。由于存在内存的分配与回收，因此我们需要用双向链表将空闲的chunk连接起来，以便在后续使用时之间从对应的chunk分配。我们称这一双向链表为free list。free list的头部是最大的chunk，我们每次分配时都从free list的头部划分进行分配，如果需要分配的内存空间超过了对应的大小，我们就从系统中继续申请大块空间，继续进行分配，同时对空闲的chunk利用双向链表进行连接，但block之间并不需要连接。

Chunk结构

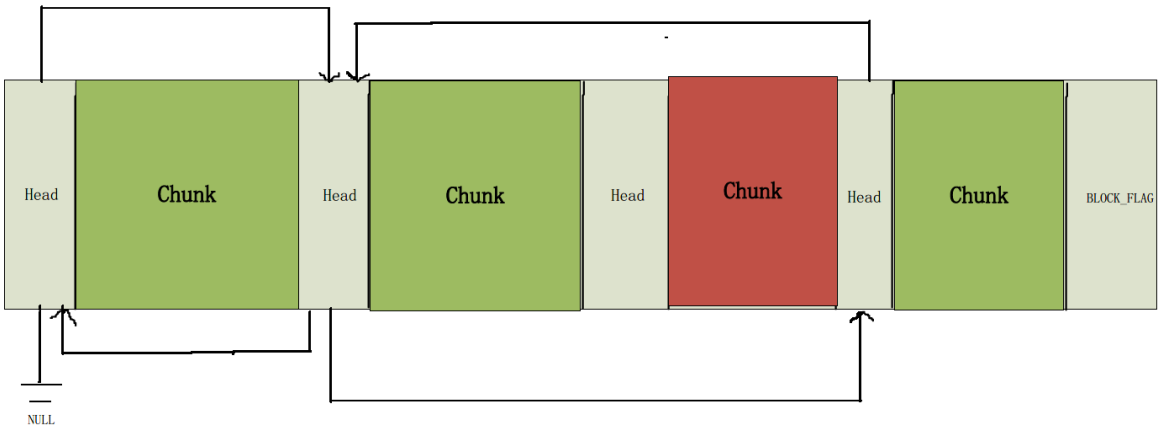
```
struct Chunk
{
    unsigned char isFree;   // If this chunk is free
    Chunk* preChunk;        // Previous chunk in the same block
    size_t size;            // Storage size of a chunk(excluding the metadata
header,i.e the struct Chunk it self)
    // Doubly free list node
    Chunk* preNode;         // Previous node in doubly linked list
    Chunk* nxtNode;         // Next node in doubly linked list
};
```

Chunk结构是分配出去的chunk的元信息(metadata)定义，存储于chunk内存块的头部。同时他也是链表的结点,包含链表的双向指针。

Block 结构

这里的一个block是指的从操作系统申请的一块较大的单元，大小也不是固定的(根据需要分配的空间而定，一般在申请空间较小的情况下为4096)，但必须要是1024的整数倍。

下图为在运行过程中一个block的结构图，其中head即为上面的struct Chunk. Chunk可以通过链表在不同的block之间进行链接。同一个block里面的chunk也链接了自己的前面一块(图中未表示)，图中红色表示已经分配，绿色标识未分配。



相关全局变量

```
// Size of metadata of a chunk
const size_t HEADER_SIZE = sizeof(Chunk);
// BlockSize
// We get memory size integer times of BlockSize
const size_t BLOCK_SIZE = 4096;
// Flag: end of a block
const unsigned char BLOCK_FLAG = 0xEE;
```

HEADER_SIZE 即struct Chunk的大小。BLOCK_SIZE这里设置为4096，表明了我们一次申请的BLOCK至少为4096. BLOCK_FLAG是BLOCK的末尾，数值随意。

MemoryPool类定义

```
class MemoryPool
{
public:
    // Typedefs
    using pointer = void*;
    using size_type = size_t;
    using difference_type = ptrdiff_t;
    using data_type = unsigned char;
    using data_pointer = unsigned char*;
    // Allocate memory of size size
    // Return the pointer
    pointer allocate(size_type size);
    // Deallocate memory pointed by ptr
    void deallocate(data_pointer ptr);
    // Constructor
    MemoryPool();
    // Destructor
    ~MemoryPool();
private:
    // Free list head
    Chunk* header;
    // Allocate block of memory from OS
    // According to the size, this function
    // allocate memory of integer times of BlockSize
    // and return the pointer
    data_pointer allocateMemory(size_type size, size_type& realSize);
    // Insert chunk after preNode in the linked list
    void insertNode(Chunk* chunk, Chunk* preNode);
```

```
    // Remove chunk from free linked list
    void removeNode(Chunk* chunk);
    // Merge chunk with its next chunk
    void mergeChunk(Chunk* chunk);
    // Maintain free list
    void maintainList();
};
```

MemoryPool类，关键就是维护了free list，其成员变量header为free list的表头。allocateMemory即从操作系统申请一个block的内存。allocate和deallocate函数是为Allocator提供的接口，分配出一个chunk的空间以及回收。insertNode与removeNode以及maintainList进行的是freelist的维护。最后的mergeChunk是将一个chunk与其下一个连续的chunk合并(如果该chunk为free的时候)

MemoryPool重要成员函数算法解析(语言描述形式)

MemoryPool::~MemoryPool()

1. Delete向量记录所有的preChunk为NULL的Chunk，这表明该Chunk为Block的首地址
2. free掉所有的向量

MemoryPool::data_pointer MemoryPool::allocateMemory(size_type size, size_type& totalSize)

1. 计算需要向操作系统申请的空间，包括分配的大小，BLOCK_FLAG，HEADER_SIZE，最后向上round到BlockSize的整数倍
2. 设置BLOCK_FLAG
3. 返回相应的地址

MemoryPool::pointer MemoryPool::allocate(size_type size) (IMPORTANT)

1. 如果空间不够，或者为初始化block，则首先调用allocateMemory初始化，此时的chunk即为一整块block，设置相关head信息，并插入free list
2. 对free list的第一个chunk，我们将其进行分割。一部分是需要分配的chunk，剩下的部分增加head信息，成为一个新的free chunk，并将其插入free list，将原来的删除。同时修改相关信息与链表指针
3. 维护free list，尽量使最大的chunk在头部。

void MemoryPool::deallocate(data_pointer ptr)

1. 记录要回收内存区域的chunk，以及其前后的chunk
2. 根据其前后的free信息，回收的chunk与前后进行合并，在free list中删除原来的结点，插入合并后的结点。
3. 维护free list

Allocator接口设计

```
template <class T>
class Allocator
{
public:

    // Typedefs
    using value_type = T;
```

```
    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 data_type = unsigned char;
    using data_pointer = unsigned char*;

    template<typename U>
    struct rebind { typedef Allocator<U> other; };

    // Default constructor
    Allocator() = default;
    ~Allocator() = default;
    template <class U>
    Allocator(const Allocator<U>&) noexcept {};

    // Address information
    pointer address(reference r) { return &r; }
    const_pointer address(const_reference r) { return &r; }

    // Max size
    size_type max_size() const {
        return std::numeric_limits<size_type>::max() / sizeof(T);
    }

    // void construct(pointer p, const T& t) { new(p) T(t); }
    // void destroy(pointer p) { p->~T(); }

    // == and != operators
    template<class U>
    bool operator==(const Allocator<U>&) const noexcept{return true;};
    template<class U>
    bool operator!=(const Allocator<U>&) const noexcept{return false;};

    // Allocate memory, return the pointer if success
    pointer allocate(size_type n, const_pointer hint = 0){
        return reinterpret_cast<pointer>(mempool.allocate(sizeof(T) * n));
    };

    // Deallocate the memory
    void deallocate(pointer p, size_type n){
        mempool.deallocate(reinterpret_cast<data_pointer>(p));
    };
private:
    static MemoryPool mempool;  // MemoryPool
};
template <typename T> MemoryPool Allocator<T>::mempool;
```

Allocator 与系统定义的基本一致。当然事实上，他本身的实现本身也不是唯一的，不同的编译器是不一样的，此处是按照对应的功能进行设计。

Performance

以下性能测试首先是根据pinti上的框架完成，但测试后发现性能并不特别理想，我怀疑是编译器本身就对std::allocator有优化，因此自己实现的allocator能有比较接近甚至有时更优的性能就是比较好的。

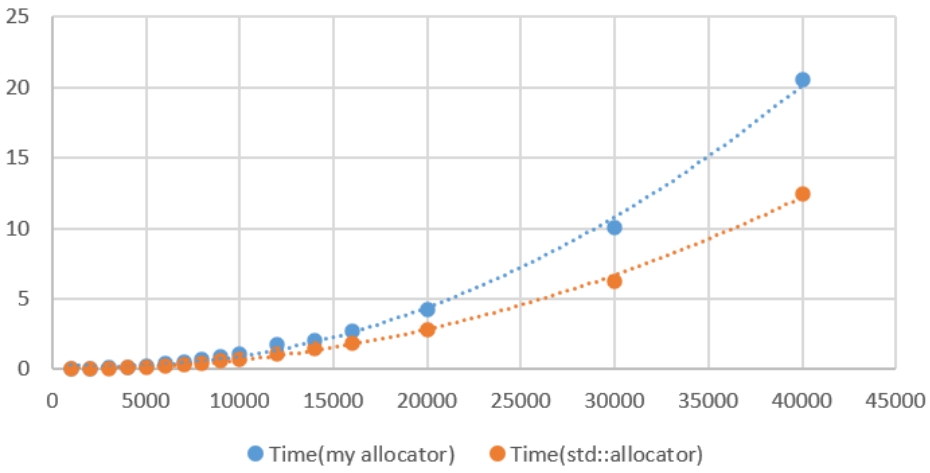
// vector creation

```
// Sample code
int clk = clock();
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));
```

My Allocator

Size	Time(my allocator)	Time(std::allocator)
1000	0.011	0.008
2000	0.043	0.031
3000	0.098	0.067
4000	0.176	0.121
5000	0.272	0.175
6000	0.399	0.250
7000	0.526	0.334
8000	0.692	0.435
9000	0.867	0.559
10000	1.064	0.740

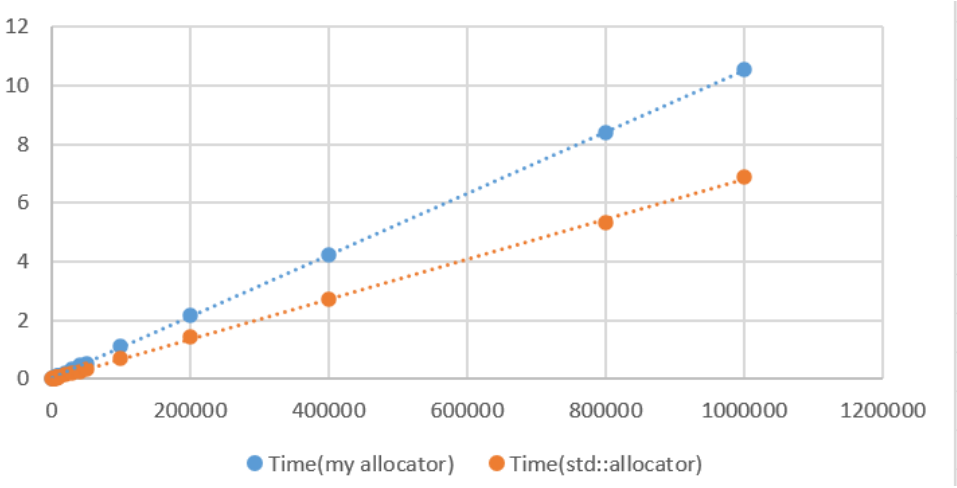
Size	Time(my allocator)	Time(std::allocator)
12000	1.734	1.115
14000	2.080	1.433
16000	2.707	1.838
20000	4.232	2.795
30000	10.015	6.270
40000	20.545	12.422



// vector resize

```
//sample code
 using IntVec = std::vector<int, MyAllocator<int>>;
 std::vector<IntVec, MyAllocator<IntVec>> vecints(TestSize);
 using PointVec = std::vector<Point2D, MyAllocator<Point2D>>;
 std::vector<PointVec, MyAllocator<PointVec>> vecpts(TestSize);
 int clk = clock();
 for (int i = 0; i < PickSize; i++)
 {
   int idx = dis(gen) - 1;
   int size = dis(gen);
   vecints[idx].resize(size);
   vecpts[idx].resize(size);
 }
```

Size	Time(my allocator)	Time(std::allocator)
1000	0.011	0.009
2000	0.023	0.014
3000	0.034	0.021
4000	0.044	0.027
5000	0.055	0.035
6000	0.064	0.041
7000	0.076	0.046
8000	0.087	0.054
9000	0.095	0.060
10000	0.107	0.066
20000	0.211	0.135
30000	0.319	0.196
40000	0.481	0.260
50000	0.513	0.329
100000	1.123	0.717
200000	2.157	1.450
400000	4.245	2.724
800000	8.382	5.321
1000000	10.574	6.877



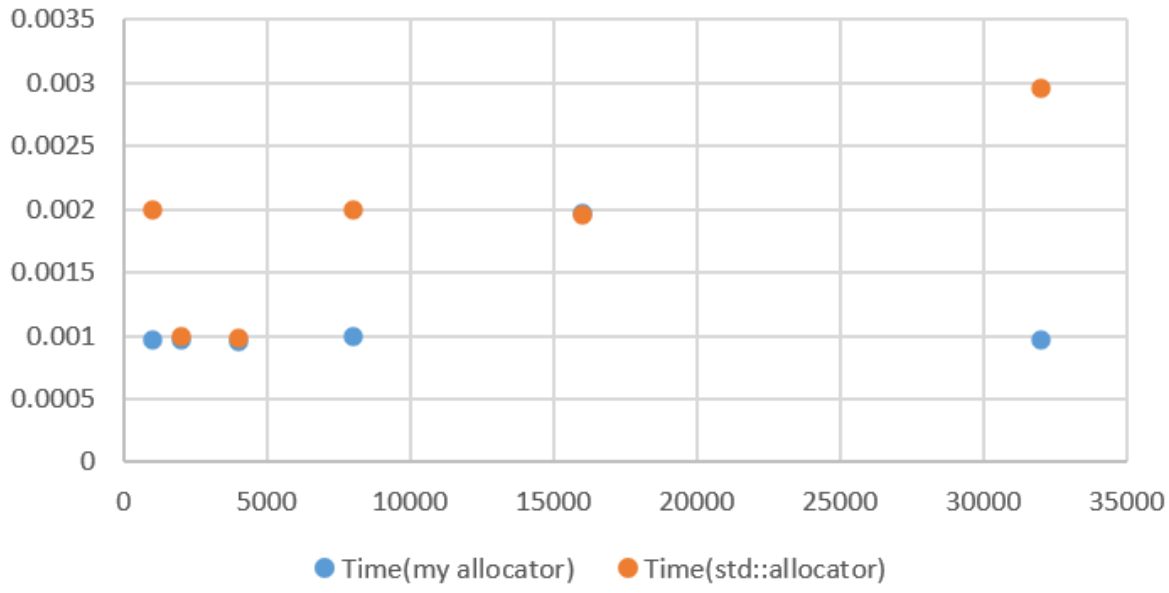
// vector element assignment

这个时间比较短，几乎看不出差距，毕竟也没涉及内存分配。

因此采用高精度计时器计时。当然，主要还是大致看一下结果

Size	Time(my allocator)	Time(std::allocator)
1000	0.000966	0.0019945
2000	0.0009603	0.0009967
4000	0.000958	0.0009773
8000	0.000999	0.0019962
16000	0.0019638	0.0019581
32000	0.0009648	0.0029514

时间类似

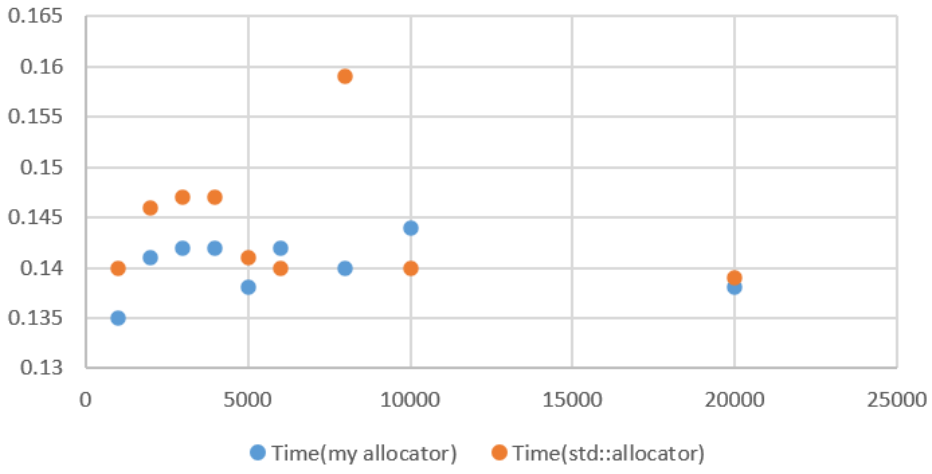


// vector release

// 开启 -o2 优化

```
    // sample code
    // vector release
    int clk = clock();
    for (int i = 0; i < TestSize; i++)
    {
        std::vector<int,MyAllocator<int>>().swap(vecints[i]);
    }
```

Size	Time(my allocator)	Time(std::allocator)
1000	0.135	0.140
2000	0.141	0.146
3000	0.142	0.147
4000	0.142	0.147
5000	0.138	0.141
6000	0.142	0.140
8000	0.140	0.159
10000	0.144	0.140
20000	0.138	0.139



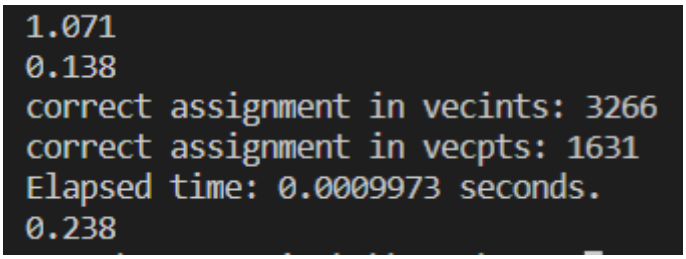
Test code

```
#include <iostream>
#include <random>
#include <vector>
#include <ctime>
#include <chrono>
#include "Allocator.h"

template<class T>
using MyAllocator = Allocator<T>;
using Point2D = std::pair<int, int>;
```

```cpp
const int TestSize = 10000;     // !!!Change size here!!!
const int PickSize = 1000;

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

  //vector creation
  int clk = clock();
  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));
  printf("%.3f\n", 1.0 * (clock() - clk) / CLOCKS_PER_SEC); // PRINT TIME


  // vector resize
  clk = clock();
  for (int i = 0; i < PickSize; i++)
  {
    int idx = dis(gen) - 1;
    int size = dis(gen);
    vecints[idx].resize(size);
    vecpts[idx].resize(size);
  }
  printf("%.3f\n", 1.0 * (clock() - clk) / CLOCKS_PER_SEC);  // PRINT TIME

  auto start = std::chrono::high_resolution_clock::now();
  // vector element assignment
  {
    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;
  }
```

```
  auto end = std::chrono::high_resolution_clock::now();
  std::chrono::duration<double> elapsed = end - start;
  //PRINT TIME
  std::cout << "Elapsed time: " << elapsed.count() << " seconds." << std::endl;

  //vector release
  clk = clock();
  for (int i = 0; i < TestSize; i++)
  {
    std::vector<int,MyAllocator<int>>().swap(vecints[i]);
  }
  printf("%.3f\n", 1.0 * (clock() - clk) / CLOCKS_PER_SEC);  //PRINT TIME

  return 0;
}
```

Sample Output of test program



总结

可以看出，自己实现的allocator在vector creation以及vector resize上性能不如std::allocator, 但与标准库比较接近(同一数量级)，这是因为我们在分配的时候进行了chunk的合并，可能导致了一定的时间开销。对于vector element assignment，我们自己实现的allocator性能总体上远远大大于标准库的性能(可能存在一定的误差)。而对于release操作，我们的释放速度也是总体比较快的，这是因为我们chunk的合并使得内存fragmentation减少，提高了释放的速度。

最后还需要支出的是，自己的机器配置不是很好，难以进行更大规模的验证，我认为，更大规模的验证才能看出二者的差异，否则结果都是比较类似的，难以体现自己实现的内存池的性能的优越性。

Reference

[1 eapach/wof_alloc Wheel-of-Fortune Memory Allocator]

[2 mtrebi/memory-allocators]