MyAllocator Report

2020.6.29

唐子豪 3180102086 CS1804

顾核金 3180103732 CS1804

1 背景和项目要求

实现一个带有内存池的 allocator，支持 std::vector。为了能够替换标准模板库的 Allocator，应当提供以下 API：

```
typedef void _Not_user_specialized;
typedef _Ty 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;

size_type max_size() noexcept;
pointer address(reference _Val) noexcept;
const_pointer address(const_reference _Val) 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);
```

Allocator 要用内存池来优化内存分配速度，满足不同大小的内存分配要求，应当在小内存的分配上有效率的提升。

2 开发环境配置

- Visual Studio
- C++17
- Windows 10 及以上/Mac OS 10.15 及以上
- 4 GB 及以上

3 数据结构和重要参数

3.1 重要参数

在实现中，整个内存分配器内共享一个内存池，在类中，我们定义了一些参数来给出整个内存池的信息：

```
const size_t ALIGN = 8;
const size_t ALIGN_BIT = 3;
const size_t MAX_BYTES = 1024;
const size_t N_FREELIST = MAX_BYTES / ALIGN;
```

```
const size_t N_CHUNK = 16;
inline static char* start_pool = NULL;
inline static char* end_pool = NULL;
```

其中 ALIGN 是分配内存的最小单元，也是内存块大小的增长量，为了计算的方便，我们用 ALIGN_BIT 来表示log_2ALIGN.

MAX_BYTES 定义了使用内存池分配的最大内存，本实现中为 1024B，超过此大小将使用原生的 malloc 和 free。

N_FREELIST 是自由链表的最大索引值，N_CHUNK 是一般情况下每次申请内存分配的内存块数量。

start_pool 和 end_pool 是内存池所分配空间的起点和终点。

3.2 自由链表

自由链表记录可分配的内存空间，在本实现中提供了 8B，16B，24B，...，1024B 的节点。每一个节点均记录了该节点在自由链表中的后继信息，即 Next 指针，结构如下：

```
typedef struct node
{
    struct node* Next;
}Obj;
```

在本分配器中，使用指针数组来表示自由链表：

```
inline static Obj* free_list[N_FREELIST] = { NULL };
```

第 0 个自由链表的节点大小为 8B，此后以 ALIGN 参数为差分，进行增长。用参数 N_FREELIST 记录自由链表的总块数。

4 内存池设计

在内存分配的时候，我们只会处理小块的内存请求，利用内存池来分配空间，对于大空间请求，使用 malloc 更为迅速一些，整个内存池分配的流程如下：

1 使用 allocate 函数向内存池请求分配_Count 个 value_type 对象的内存空间，如果需要请求的内存大小大于 1024 bytes，直接使用 malloc.

2 否则 allocate 用 FREELIST_INDEX 函数找到最适合的自由链表.

a) 如果链表不为空，返回该自由链表，将数组中指向该自由链表的指针指向下一个 node.

b) 如果链表为空，意味着内存池为空，所有的自由链表都为空链表，使用 refill 函数填充内存池.

3 用户调用 deallocate 释放内存空间，如果要求释放的内存空间大于 1024bytes，直接调用 free.

4 否则按照其大小用 FREELIST_INDEX 找到合适的自由链表，并将其插入，而不是直接返还给操作系统，以提高运行速度.

主要代码如下：

```
static _DECLSPEC_ALLOCATOR pointer allocate(size_type _Count)
{
    Obj** cur_free_list;
```

```
    Obj* dst;
    int index;
    _Count *= sizeof(value_type);
    if (_Count > MAX_BYTES)
        return (pointer)std::malloc(_Count);
    index = FREELIST_INDEX(_Count);
    cur_free_list = free_list + index;
    dst = *cur_free_list;
    if (!dst)
        return (pointer)refill(ROUND_UP(_Count));
    *cur_free_list = dst->Next;
    return (pointer)dst;
}
```

5 其他函数分析

5.1 内存对齐函数 ROUND_UP

在实际内存分配中，申请的内存往往不是 ALIGN 的幂次，因此我们需要把最适合的块分配给请求，即需要把请求的空间大小放大到 ALIGN 的整数幂次去。比如 ALIGN=8 的话，要将 7 放大至 8，8 不变，10 放大至 16，以此类推。ROUND_UP 函数用位运算实现了这一操作，具体代码如下：

```
static inline size_type ROUND_UP(size_type bytes)
{
    return (bytes + ALIGN - 1) & ~(ALIGN - 1);
}
```

5.2 自由链表索引函数 FREELIST_INDEX

同样的，在寻找自由链表时，我们使用了函数 FREELIST_INDEX 给出合适的索引，在 ALIGN=8 的情况下，0-7 的内存请求均属于 0，8-15 属于 1，以此类推。

```
static inline size_type FREELIST_INDEX(size_type bytes)
{
    return ((bytes + ALIGN - 1) >> ALIGN_BIT) - 1;
}
```

5.3 自由链表补充函数 refill

在自由链表用尽时，我们需要进行补充，这时候用到了 refill 函数，它向内存池调用 chunk_alloc 函数，请求足够的空间，然后添加至 free_list 当中，一般来说，将会分配 N_CHUNK 个节点，在内存池不够的情况下，分配节点的个数将会减少。

```
static void* refill(size_type _Count)
{
    // Default to allocate N_CHUNK objects.
    int n_obj = N_CHUNK;
    char* chunk = chunk_alloc(_Count, n_obj);
    Obj** cur_free_list;
    Obj* dst;
    Obj* cur_obj, * next_obj;
    int i;

    // If chunk_alloc returns only one object, return.
    if (n_obj == 1) return chunk;
    cur_free_list = free_list + FREELIST_INDEX(_Count);

    dst = (Obj*)chunk;
    *cur_free_list = next_obj = (Obj*)(chunk + _Count);
    for (i = 1; i < n_obj; i++)
    {
        // Build the link.
    }

    return dst;
}
```

5.4 内存块分配函数 chunk_alloc

受到 refill 函数的调用，返回足够的内存空间。

1 如果内存池中空间足够，就直接分配，然后把指向内存池中空间起始位置的 start_pool 移到新的位置.

2 如果内存池不够一个 chunk 的空间但可以分配几个 nodes，直接返回给用户，依然要移动 start_pool.

3 如果连一个 node 都没有，再次向操作系统请求分配内存.

a) 分配成功，再次用 chunk_alloc 函数.

b) 分配失败，抛出异常.

```
static char* chunk_alloc(size_type size, int& n_obj) {
    char* dst;
    size_type total_bytes = size * n_obj;
    size_type bytes_left = end_pool - start_pool;
    if (bytes_left >= total_bytes) {
        return it;
    } else if (bytes_left >= size) {
        return space as much as we can;
    } else {
```

```
            spend the memory pool;
            add them to free_list;
            malloc new space;
            if (success) {
                reuse chunk_alloc;
            } else
                throw "Cannot allocate enough space!";
        }
        return NULL;
    }
```

6 测试与分析

6.1 测试方案

在此内存分配器设计中，我们使用了很多参数，因此需要对参数进行测试来达到最优的性能。测试考虑了以下 3 个参数:

MAX_BYTES:

- 允许在分配器中分配的最大块大小。如果所需的大小大于它，则将直接使用 malloc ()。
- 显然,MAX_BYTES 太小会大大降低效率,而 MAX_BYTES 太大会给系统带来负担。此外，如果 MAX_BYTES 是 2 的幂，则足够好。因此，选择作为候选值。

ALIGN:

- free_list 中使用的块大小增量，即 free_list[i]的块大小应为$(i-1)\times ALIGN$.
- 在此测试中，最小的元组不小于 int64_t。因此 ALIGN 的值大于 8 个字节。由于太大会导致空间浪费。选择 8,16,32 作为候选值，因为它们都是 2 的幂。

N_CHUNK:

- 在内存池模式中，每次分配一些空间时，不仅会分配每个 ser 的空间，还会分配一块 chunk。 N_CHUNK 会定期提供此块。
- 鉴于 GNU 2.9 中的 N_CHUNK 设置为 20，我们试图找到更好的 N_CHUNK 来满足我们的应用场景。因此，选择的数字是 2 的幂。它们是 2、4、8、16、20、32。

如项目要求中所述，分配器应在处理小规模分配方面表现出色。测试中的最大元组大小设置为 50，而测试数设置为 100,000。

每个测试及其结果都包含在从属目录中，测试框架为 test.bat。可以轻松地为每个测试用例更改.bat 文件中的参数。

在每个测试中，将 std::allocator 添加为对照组。在评估完参数之后，选择最佳参数并在源代码中对其进行修改，将其作为不变量，估计剩余的参数。

7 测试结果与分析

7.1 MAX_BYTES

MAX_BYTES 是我们测试的第一个参数，我们希望控制我们分配的最大内存，来合理地使用内存池。实际上这一参数会和测试的数据规模有一定的关系，我们利用老师给的样例测试程序，进行了简单的修改，完成了此次测试，用 TICKS 来体现分配器的分配速率，结果如下：

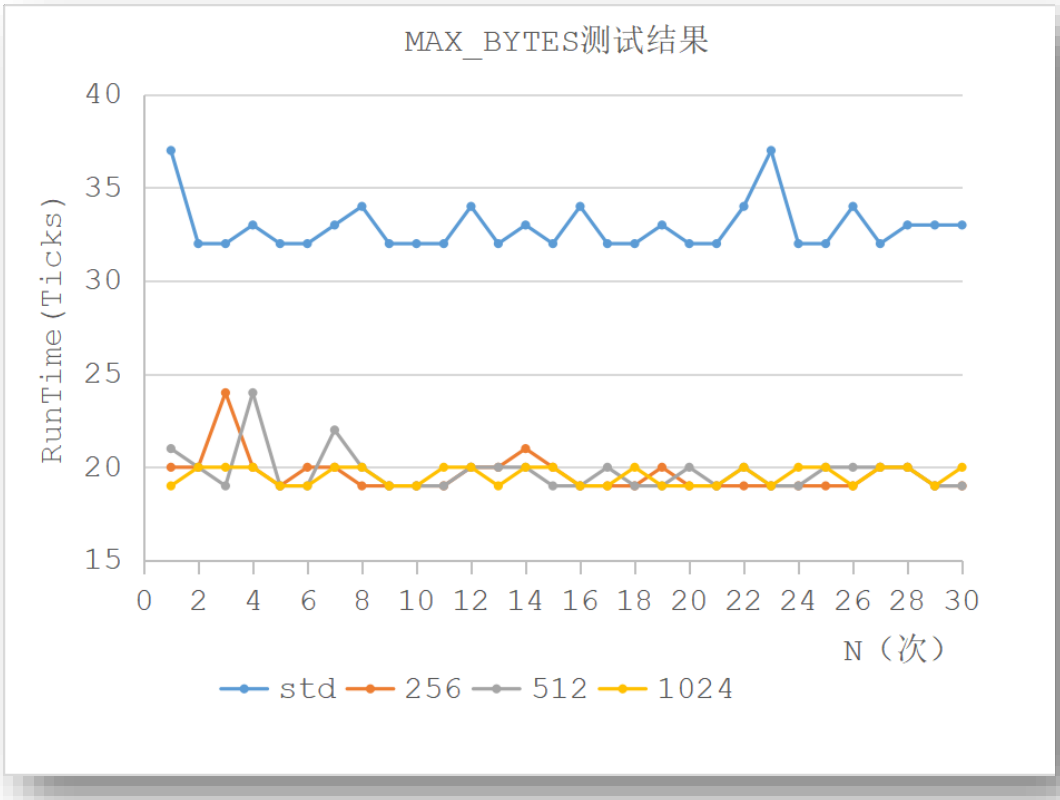

图 6-1 MAX_BYTES 测试结果图

考察这 30 次测试花费的平均时间，有如下结果：

表 6-1 MAX_BYTES 测试结果表

Runtime_ver1(Ticks) 1<=RESIZE<=50				
	std	MAX_BYTES		
		256	512	1024
Average Ticks	32.90	19.60	19.73	19.53

从结果可以看出，本内存器的实现比 std::allocator 的效率要高，在 MAX_BYTES

设计为 1024B 时，分配时间最为迅速，并且也没有出现极端的测试数据，非常稳定，因此将 MAX_BYTES 设置为 1024.

7.2 ALIGN

ALGIN 控制了 free_list 的密度，利用同样的测试方法，我们对 ALIGN 进行了测试，整体情况如下图：

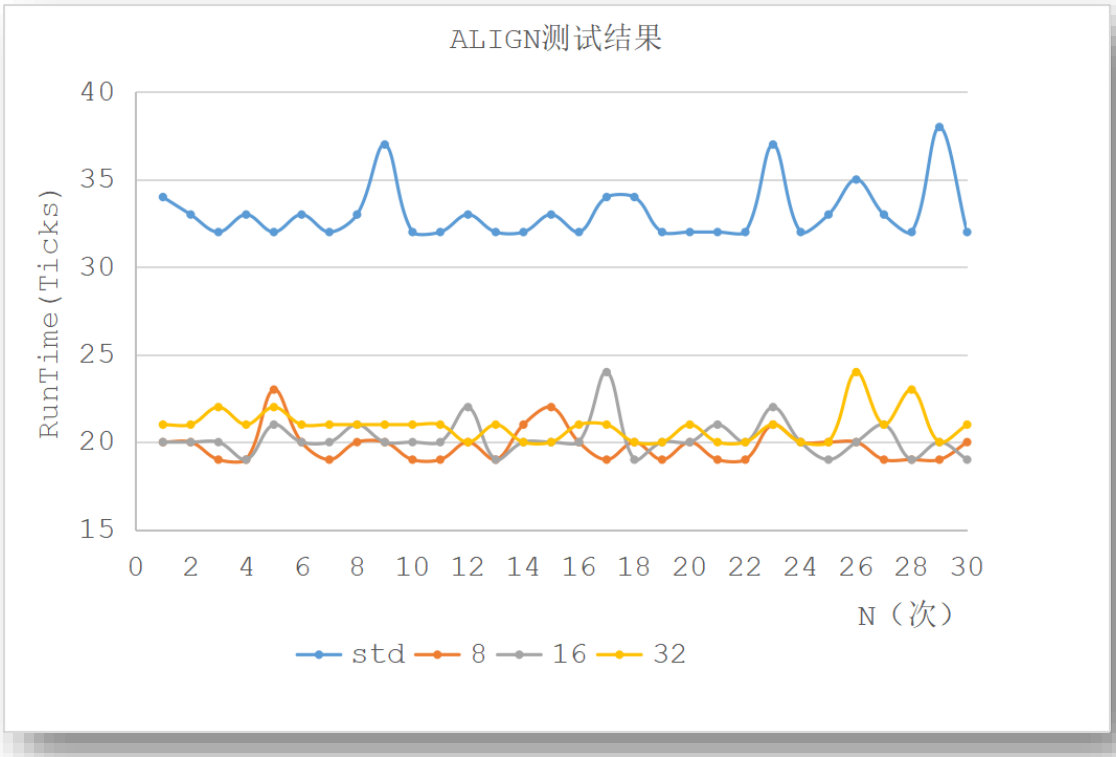

图 6-2 ALIGN 测试结果图

考察这 30 次测试花费的平均时间，有如下结果：

表 6-2 ALIGN 测试结果表

Runtime_ver2(Ticks) 1<=RESIZE<=50, MAX_BYTE=1024				
	std	ALIGN		
		8	16	32
Average Ticks	33.10	19.80	20.20	20.90

ALIGN=8 时候的分配器体现了比较平稳、迅速的运行情况，因此设置参数 ALIGN=8.

7.3 N_CHUNK

在 GNU2.9 中，N_CHUNK 是 20，但是作者并没有解释这样安排的原因，为了搭建适用

于小内存的分配器，我们给 N_CHUNK 进行了大量的赋值:2,4,8,16,20,32,测试结果如下:

表 6-3 N_CHUNK 测试结果表

Runtime_ver3(Ticks) **1<=RESIZE<=50, MAX_BYTE=1024, ALIGN=8**							
N	std	N_CHUNK					
		20	16	32	8	4	2
1	44	22	24	21	29	25	28
2	37	20	19	20	19	21	22
3	41	23	24	20	22	23	21
4	34	20	20	21	19	27	23
5	36	21	20	20	22	23	22
6	37	21	20	27	22	21	23
7	39	21	20	21	21	20	23
8	39	26	**42**	31	24	20	28
9	37	22	21	26	28	20	22
10	36	21	20	21	20	21	21
11	38	25	24	22	20	19	22
12	32	19	19	20	19	20	22
13	33	20	21	21	22	20	20
14	34	27	23	20	20	19	23
15	42	23	20	20	24	24	26
16	35	21	20	26	22	27	24
17	32	19	19	20	19	19	19
18	38	20	19	20	19	19	20
19	33	20	19	24	27	29	21
20	33	20	19	20	19	20	28
21	33	20	20	20	26	26	19
22	33	20	22	25	19	19	26
23	56	21	20	20	19	20	20
24	34	20	21	25	20	21	21
25	34	24	21	21	21	24	20
26	35	20	20	21	19	21	21
27	35	21	19	20	20	20	22
28	39	20	19	20	20	20	20
29	34	21	20	20	19	22	23
30	42	25	25	23	25	20	24
Average Ticks	36.83	21.43	21.33	21.87	21.50	21.67	22.47

N_CHUNK=16 时，虽然在第 8 次测试中，出现了极端数据 42，但是仍然表现出了最好的性能，因此此参数定为 16.

7.4 最终测试

在确定完所有参数之后，我们需要正式比较此分配器和 std::allocator 的性能，此次测试的结果在 Test_TUPLE_SIZE 中，我们观察不同的 Resize 大小时，两个内存分配器的效率变化，测试中 Resize 的最大值限制依次为 10，25，50，75，100，200.

表 6-4 最终测试结果表

Runtime_ver4(Ticks) MAX_BYTE=1024，ALIGN=8，N_CHUNK=16							
Tuple Size		10	25	50	75	100	200
Average Ticks	std	17.97	23.33	31.70	31.70	46.73	76.33
	my	11.13	15.13	21.50	27.93	34.73	64.97
std/my		1.59	1.54	1.47	1.13	1.35	1.17

从数据中不难发现，Tuple Size 从 10 变化到 200 的过程中，MyAllocator 的运行时间始终比 std::allocator 短，而且数据规模很小时，运行速度几乎接近原生的两倍。50 是常规定义下的小内存的分配界限，在此时 MyAllocator 比 std::allocator 快 1.47 倍左右，有非常好的性能，但是随着数据规模的增大，运行速度逐渐接近原生的内存分配器，可以预见当数据规模极大时，MyAllocator 的优势将会消失，这是因为当每次的内存请求都比较大时，内存碎片化的情况反而会减小，因此内存池的设计不会具备太大的优势。总体来说，MyAllocator 比较符合内存分配器的性能要求。

8 附录

8.1 MyAllocator.hpp

```
// MyAllocator.cpp
// All the objects in this file should be used
// in the namespace my.
// The schema refers to GNU2.9, written by Zihao Tang.

#include<iostream>
#include<cassert>
#pragma once

#define _DEBUG 0
namespace my {

// Parameters used to observe the status of alloc.
```

```
#if _DEBUG
    static int cnt_big = 0;
    static int cnt_small = 0;
#endif

    // Basic parameters of the allocator.
    // ALIGN & ALIGN_BIT denote the increament of the
    // size of Free_List.
    // MAX_BYTES denotes the maximum size of free_list.
    // N_FREELIST denotes the number of free_list, which
    // is decided by ALIGN & MAX_BYTES.
    // N_CHUNK denotes the number allocated blocks of each new a
lloc
    const size_t ALIGN = 8;
    const size_t ALIGN_BIT = 3;
    const size_t MAX_BYTES = 1024;
    const size_t N_FREELIST = MAX_BYTES / ALIGN;
    const size_t N_CHUNK = 16;

    // We use the struct Obj to link the free_list.
    typedef struct node
    {
        struct node* Next;
    }Obj;

    // The template of customized allocator.
    template <class _Ty>
    class MyAlloc
    {
    public:

        // The typename required by a std allocator.
        typedef void _Not_user_specialized;
        typedef _Ty 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 std::ptrdiff_t difference_type;
        typedef std::true_type propagate_on_container_move_assig
nment;
        typedef std::true_type is_always_equal;
```

```
        // A necessary class require by a std allocator,
        // used to substitute for different types.
        template<class T>
        struct rebind { typedef MyAlloc<T> other; };

        // The start and end of memory pool, struct Obj
        // is embeded in the pool.
        // alloc_size
        // free_list records the free space remained for
        // incoming memory allocation.
        inline static char* start_pool = NULL;
        inline static char* end_pool = NULL;
        inline static size_type alloc_size = 0;
        inline static Obj* free_list[N_FREELIST] = { NULL };

        // Since the class allocator shares the same memory pool
,
        // ctors & copy ctors are not in need.
        MyAlloc() noexcept {}

        // Round bytes up to the nearest multiples of ALIGN that
        // no less than bytes.
        // e.g. ALIGN=8, 7->8, 8->8, 10->8
        static inline size_type ROUND_UP(size_type bytes)
        {
            return (bytes + ALIGN - 1) & ~(ALIGN - 1);
        }
        // Get the index of the free_list 0-N_FREELIST-1,
        // 0-ALIGN-1 -> 0, ALIGN-2ALIGN-1 ->1, ...
        // Bit operation is used to accelerate calculations.
        static inline size_type FREELIST_INDEX(size_type bytes)
        {
            return ((bytes + ALIGN - 1) >> ALIGN_BIT) - 1;
        }
        static void* refill(size_type _Count)
        {
            // Default to allocate N_CHUNK objects.
            int n_obj = N_CHUNK;
            char* chunk = chunk_alloc(_Count, n_obj);
            Obj** cur_free_list;
            Obj* dst;
            Obj* cur_obj, * next_obj;
            int i;
```

```
        // If chunk_alloc returns only one object, return.
        if (n_obj == 1) return chunk;
        cur_free_list = free_list + FREELIST_INDEX(_Count);

        dst = (Obj*)chunk;
        *cur_free_list = next_obj = (Obj*)(chunk + _Count);
        for (i = 1; i < n_obj; i++)
        {
            // Build the link.
            cur_obj = next_obj;
            next_obj = (Obj*)((char*)next_obj + _Count);
            if (i < n_obj - 1)
            {
                cur_obj->Next = next_obj;
            }
            else
            {
                cur_obj->Next = NULL;
            }
        }

        return dst;
    }
    // Alloc a chunk of space
    static char* chunk_alloc(size_type size, int& n_obj)
    {
        char* dst;
        // Calculate the bytes to allocate and
        // the bytes left.
        size_type total_bytes = size * n_obj;
        size_type bytes_left = end_pool - start_pool;

        // If pool has enough space, return it.
        if (bytes_left >= total_bytes)
        {
            dst = start_pool;
            start_pool += total_bytes;
            return dst;
        }
        // If it's not enough for a chunk, but enough for
        // several tuples, return it.
        else if (bytes_left >= size)
        {
            dst = start_pool;
```

```
                // Number of tuples we allocate.
                n_obj = bytes_left / size;
                total_bytes = n_obj * size;
                start_pool += total_bytes;
                return dst;
            }
            // If the pool is not enough for one tuple
            // malloc new space.
            else
            {
                // Allocated spaced is the doubled bytes require
d
                // together with some margin decided by alloc_si
ze:
                // the amount we have allocated already.
                size_type bytes_alloc = (total_bytes << 1) +
                    ROUND_UP(alloc_size >> 4);
                Obj** cur_free_list;

                // If the pool is not empty, add it to free_list
.
                if (bytes_left)
                {
                    cur_free_list = free_list + FREELIST_INDEX(b
ytes_left);
                    ((Obj*)start_pool)->Next = *cur_free_list;
                    *cur_free_list = (Obj*)start_pool;
                }

                start_pool = (char*)std::malloc(bytes_alloc);
                // We should test whether malloc successes.
                // If it does, recall chunk_alloc.
                if (start_pool)
                {
                    end_pool = start_pool + bytes_alloc;
                    alloc_size += bytes_alloc;
                    return chunk_alloc(size, n_obj);
                }
                else
                {
                    throw "Cannot allocate enough space!";
                }
            }
            return NULL;
```

```
        }

        // Copy ctor is not in need.
        template<class T>
        MyAlloc(const MyAlloc<T>& a) noexcept {}

        // This implement is diffcult to dtor.
        // Leave it to the compiler.
        ~MyAlloc() noexcept {}

        // max_size is necessary in a std allocator.
        // Hence, return a  sufficiently large number.
        static inline size_type max_size() noexcept
        {
            return size_type(UINT_MAX / sizeof(value_type));
        }
        // Since & can be overloaded, use addressof to
        // complete the operation.
        static inline pointer address(reference _Val) noexcept
        {
            return std::addressof(_Val);
        }
        // The const version of &.
        static inline const_pointer address(const_reference _Val
) noexcept
        {
            return std::addressof(_Val);
        }
        // Dellocate the memory allocated.
        static inline void deallocate(pointer _Ptr, size_type _C
ount)
        {
            // Get the size of space.
            _Count *= sizeof(value_type);
            // If the space > MAX_BYTES, we used malloc
            // directly to give it memory. Therefore, free it.
            // Else, we should return it to the free_list.
            if (_Count > MAX_BYTES)
            {
                std::free(_Ptr);
                return;
            }
            else
            {
```

```
                // Find the index of free_list
                Obj** cur_free_list = free_list + FREELIST_INDEX
(_Count);
                // Append it to the head of the free_list.
                // It saves time to find the tail.
                ((Obj*)_Ptr)->Next = *cur_free_list;
                *cur_free_list = (Obj*)_Ptr;
            }
        }
        // allocate _Count value_type objects.
        // In this part we round the space up to power of 2
        // and return the total memory.
        // Obviously, every time we use allocate, only one
        // block in the free_list is used.
        static _DECLSPEC_ALLOCATOR pointer allocate(size_type _C
ount)
        {
            Obj** cur_free_list;
            Obj* dst;
            int index;

            // Get the real space.
            _Count *= sizeof(value_type);
            if (_Count > MAX_BYTES)
            {
#if _DEBUG      // Record number of big allocs for optimizations
.
                ++cnt_big;
                if (cnt_big % 1000 == 0)
                    printf("cnt_big %d: %d\n", cnt_big, _Count);
#endif //  _DEBUG

                // The space required is quite large.
                // Use malloc directly.
                return (pointer)std::malloc(_Count);
            }

#if _DEBUG  // Record small allocs for optimizations,
            ++cnt_small;
            if (cnt_small % 10 == 0)
                printf("cnt_small %d: %d\n", cnt_small, _Count);
#endif // _DEBUG

            // Get the index of the free_list.
```

```
            index = FREELIST_INDEX(_Count);
            cur_free_list = free_list + index;
            dst = *cur_free_list;

            // The pool is empty.
            // We need to refill the pool.
            if (!dst)
            {
                return (pointer)refill(ROUND_UP(_Count));
            }

            // Give the first block, and free_list points
            // to the next block.
            *cur_free_list = dst->Next;
            return (pointer)dst;
        }

        // dtor for the object using this allocator.
        template<class _Uty>
        static inline void destroy(_Uty* _Ptr)
        {
            _Ptr->~_Uty();
        }
        // ctor for the object using this allocator.
        // Here PLACEMENT-NEW is used, for we should construct
        // the object with the memory allocated in advance.
        // If we use operation new here, new space will be alloc
ated
        // instead of contruct the object on the space we give i
t.
        template<class _Objty, class _Types>
        static inline void construct(_Objty* _Ptr, _Types _Args)
        {
            new(_Ptr) _Objty(_Args);
        }
    };
}
```

8.2 test.cpp

```
#include <iostream>
#include <random>
#include <ctime>
```

```cpp
#include <iomanip>
#include <vector>
#include "MyAllocator.hpp"

// paramter used to choice allocator
#define _MY_ALLOC_ 0
#define _TEST_TUPLE_SIZE_ 1
#if _MY_ALLOC_

template<class T>
using MyAllocator = my::MyAlloc<T>;

#else

template<class T>
using MyAllocator = std::allocator<T>;

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

// For we need to test small cases, each vector has size
// no more than 50. All the data are subject to uniform
// distribution.
const int TestSize = 100000;
const int PickSize = 1000;

#if not _TEST_TUPLE_SIZE_
const int TupleSize = 50;
#endif

int main()
{
#if _TEST_TUPLE_SIZE_
    int TupleSize;
    std::cin >> TupleSize;
#endif // _TEST_TUPLE_SIZE_

    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(1, TestSize);
    std::uniform_int_distribution<> dis_size(1, TupleSize);

    // Record time.
    clock_t begin, end;
```

```
    begin = clock();
    // vector creation
    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_size(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_size(gen));

    // vector resize
    for (int i = 0; i < PickSize; i++)
    {
        int idx = dis(gen) - 1;
        int size = dis_size(gen);
        vecints[idx].resize(size);
        vecpts[idx].resize(size);
    }
    end = clock();

    std::cout << "Run-
time:" << std::scientific << (end - begin) << " Ticks\n";
    // vector element assignment, used to debug
#if _DEBUG
    {
        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: " << id
x1 << 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: " << idx
1 << std::endl;
        else
            std::cout << "incorrect assignment in vecpts: " << i
dx1 << std::endl;
    }
#endif

#if _DEBUG
    system("PAUSE");
#endif
    return 0;
}
```

8.3 test.bat（参数测试）

```
@echo off

set /a N_REC = 30
set /a N_CASE = 6

echo "Testing..."
for /L %%i in (1,1,%N_REC%) do (
    :: echo Test%%i
    for /L %%j in (1,1,%N_CASE%) do (
        :: echo Test%%i.%%j
        %%j.exe >> test%%j.log
    )
)

echo "Extracting..."
for /L %%i in (1,1,%N_CASE%) do (
    echo Test%%i >> output.log
    cat test%%i.log | sed "s/[^0-9]//g" >> output.log
    echo # >> output.log
)

pause
```

8.4 test.py（性能测试）

```
import subprocess
import re

N_REC = 30
N_CASE = 2
N_TEST = 6
N_LOG = N_TEST * N_CASE
TEST_SIZE = [10, 25, 50, 75, 100, 200]
_DEBUG = 0

def Generate():
    f = {}

    for i in range(1, N_LOG + 1):
        logName = "test"+str(i)+".log"
        f[i] = open(logName, 'a')

    for i in range(1, N_REC + 1):
        print("Test"+str(i))
        for j in range(0, N_TEST):
            for k in range(1, N_CASE + 1):
                ExeName = "./" + str(k) + ".exe"
                FILE = subprocess.run(args=ExeName, input=str(TE
ST_SIZE[j]),
                                      capture_output=True, text=
True)
                line = FILE.stdout
                print("writing to " + str((k-
1)*N_TEST+j+1) + ".log\nSize="+str(TEST_SIZE[j]))
                f[(k-1)*N_TEST+j+1].write(line)

    for i in range(1, N_LOG + 1):
        f[i].close()

    return

def Filter():
    f = {}
    f[0] = open("output.log", 'a')
    string = re.compile('[^0-9\n]*')
```

```
    for i in range(1, N_LOG + 1):
        logName = "test"+str(i)+".log"
        f[i] = open(logName, 'r')
        lines = ''.join(f[i].readlines())
        lines = re.sub(string, '', lines)
        f[0].write("Test"+str(i)+'\n')
        f[0].writelines(lines)
        f[0].write("#\n")
        f[i].close()
    f[0].close()
    return

def main():
    if _DEBUG:
        FILE = subprocess.run(args='./1.exe', input=str(200),
                              capture_output=True, text=True)
        print(FILE.stdout)
    else:
        print("Testing...")
        Generate()
        print("Filtering...")
        Filter()

    return

main()

import subprocess
import re

N_REC = 30
N_CASE = 2
```