Project - MyAlloctor & MemoryPool - Report

Haoyi DUAN Date: 2021-6-24

Introduction

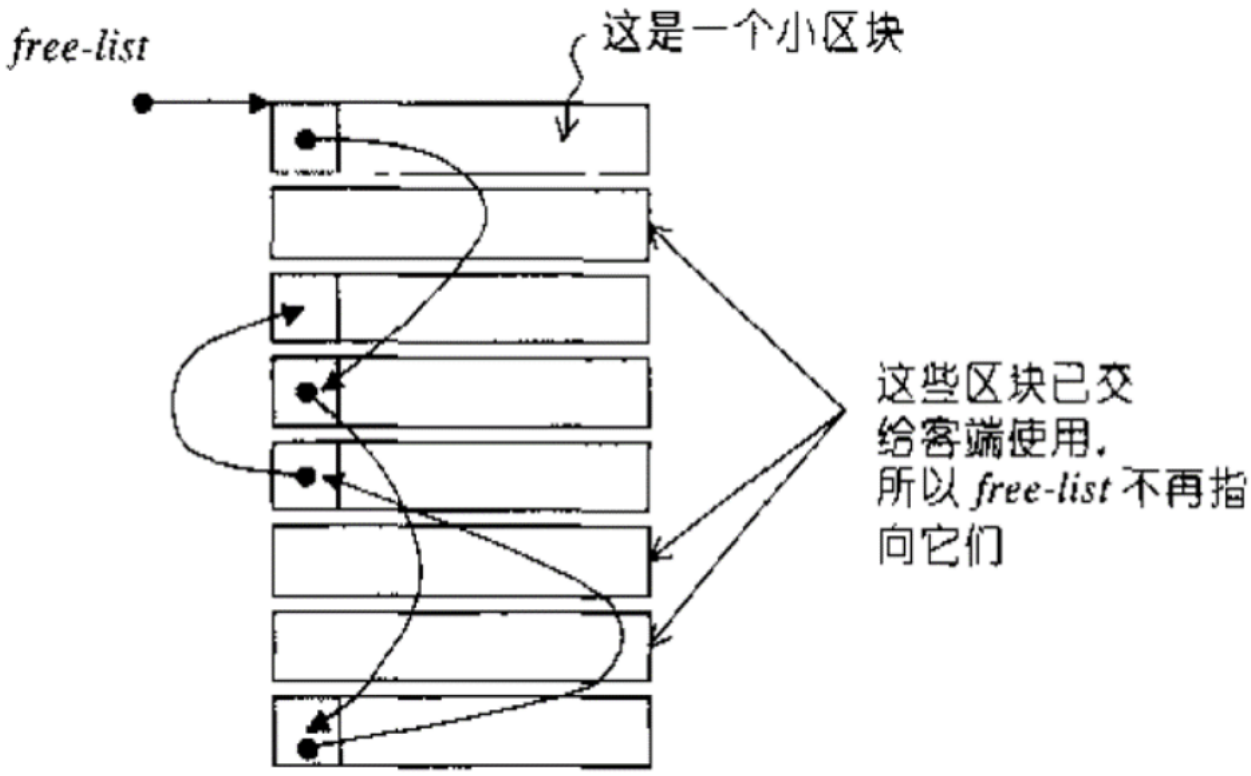
Allocator & MemoryPool

An allocator is used by standard library containers as a template parameter:

```
template < class T, class Alloc = allocator<T> > class vector;
template < class T, class Alloc = allocator<T> > class list;
```

In this project, we are required to implement our own allocator to replace the std::allocator which is a default allocator provided by STL.

To lower the memory fragmentation, we can design a memory pool to speed up the dynamic allocation of a large number of small blocks. The structure of the memory pool is shown below:



Task

- Implement my own memory allocator for STL vector.
- The allocator should optimize memory allocation speed using memory pool.
- The allocator should support arbitraty memory size allocation request.

Algorithm Specification

Allocator

After viewing the official website and check the standard of the stl::allocator, my own template class allocator is finally designed under some compromise, since many functions std::allocator provided have been removed or deprecated after C++20 has been released. So, some functions presented in my own allocator may have been removed from the new standard, but that doesn't matter that much. The definition of my template class MyAllocator is shown below:

```
template <class T>
class MyAllocator
{
public:
    typedef void _Not_user_specialized;
    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 size_t size_type;
    typedef ptrdiff_t difference_type;

    template <typename U> struct rebind
    {
        typedef MyAllocator<U> Other;
    };
    MyAllocator() = default;
    MyAllocator(const MyAllocator &myallocator) = default;
    MyAllocator(MyAllocator &&myallocator) = default;
    template <class U>
    MyAllocator(const MyAllocator<U> &myallocator) noexcept;
    pointer address(reference _Val) const noexcept{
```

```
        //return address of value
        return &_Val;
    }
    const_pointer address(const_reference _Val) const noexcept{
        return &_Val;
    }
    void deallocate(pointer _Ptr, size_type _Count){
        MemoryPool::deallocate(_Ptr,_Count* sizeof(value_type));
    }
    pointer allocate(size_type _Count) {
        if(auto res = static_cast<pointer>
(MemoryPool::allocate(_Count * sizeof(value_type))))
            return res;
        else throw bad_alloc();
    }
    template <class _Uty>
    void destroy(_Uty *_Ptr){
        _Ptr->~_Uty();
    }
    template <class _Objty, class... _Types>
    void construct(_Objty *_Ptr, _Types &&... _Args){
    new(_Ptr) _Objty(forward<_Types>(_Args)...);
    return;
    }
};
```

Obviously, the **allocate** and **deallocate** is done by memorypool, with caller

```
static_cast<pointer>(MP.allocate(_Count * sizeof(value_type)))
```

and

```
MP.deallocate(_Ptr,_Count* sizeof(value_type))
```

respectively.

MemoryPool

From my personal point of view, MemoryPool is very significant since it is the key to the performance of MyAllocator. This part, I will give you a brief discription of the MemoryPool.

definition

NAME	SIZE
__ALIGN	8
__MAX_BYTES	128
__NFREELISTS	**MAX_BYTES** / ALIGN

data structure

```
union obj
{
    union obj * free_list_link;
    char client_data[1];
};
```

union freeNode is used to store the pointer to each block, as each only call one pointer, so we can use union instead of structure to minimize the memory cost.

allocate

```
    static _Not_user_specialized * allocate(size_type n)
    {
        obj * volatile * my_free_list;
        obj * result;
        // 大于128就调用一级配置器
        if(n > (size_type) __MAX_BYTES)
            return malloc(n);

        // 寻找16个free lists中适当的一个
        my_free_list = free_list + FREELIST_INDEX(n);
        result = *my_free_list;
        if(result == nullptr){
```

```
            // 未找到可用的free list，准备重新填充free list
            return refill(ROUND_UP(n));
        }
        // 调整free list
        *my_free_list = result->free_list_link;
        return result;
    }
```

Given the bytes n ($sizeof(values\ type) \times nunber\ of\ elements$), MemoryPool::allocate will return the pointer of the space allocated. If $n > MAX\ BYTE$, use malloc instead.

If res = nullptr, it means we meet an overflow when allocating, so we call refill function.

deallocate

```
    static _Not_user_specialized deallocate(_Not_user_specialized *
p, size_type n)
    {
        obj * q = (obj*)p;
        obj * volatile * my_free_list;

        // 大于128就调用第一级配置器
        if(n > (size_type) __MAX_BYTES)
        {
            free(p);
            return;
        }

        // 寻找对应的free list
        my_free_list = free_list + FREELIST_INDEX(n);
        //调整free list，回收区块
        q->free_list_link = *my_free_list;
        *my_free_list = q;
    }
```

If the space to be deallocate is larger than MAX_BYTE, correspondent with *new*, use free to deallocate.

Otherwise, we should find the position of the pointer in the freeNode, and add to the freeList.

freeListIndex

```
    static size_type FREELIST_INDEX(size_type bytes)
    {
        return ((bytes + __ALIGN - 1)/ __ALIGN - 1);
    }
```

Compute the offset of the block.

Test Result

Test 1

```
#define _TEST_STD_SIMPLE_
```

This test is to test the performance of stl::allocator when doing a lot of push_back to a vector.

```
int main ()
{
    clock_t begin, end;
    std::vector<int> integer;

    cout << "std::llocator" << endl;
    begin = clock();
    for (auto i = 0; i < TESTSIZE; i++)
        integer.push_back(i);
    end = clock();
    cout << "The runtime of " << TESTSIZE << " push operations is "
<< (double)(end - begin) / CLOCKS_PER_SEC << "s" << endl;

    std::vector<vector<int>> vectorTmp;
    cout << "MyAllocator" << endl;
    begin = clock();
    for (auto i = 0; i < TESTSIZE; i++)
        vectorTmp.push_back(integer);
    end = clock();
```

```
    cout << "The runtime of " << TESTSIZE << " push operations is "
<< (double)(end - begin) / CLOCKS_PER_SEC << "s" << endl;

    return 0;
}
```

- **Result**

```
std::llocator
The runtime of 10000 push operations is 0s
MyAllocator
The runtime of 10000 push operations is 0.111s
```

Test 2

```
#define _TEST_SIMPLE_
```

This test is to test the performance of MyAllocator when doing a lot of push_back to a vector. Test code is similar to Test 1.

```
MyAllocator
The runtime of 10000 push operations is 0.002s
MyAllocator
The runtime of 10000 push operations is 1.768s
```

Test 3

```
#define _TEST_CORRECTNESS_
```

This test shows that whether my allocator is functional well.

```
    //test assignment
    tIndex = (int)((float)rand() / (float)RAND_MAX * (TESTSIZE - 4
- 1));
    int tIntValue = 10;
    testVec[tIndex]->setElement(testVec[tIndex]->size() / 2,
&tIntValue);
    if (!testVec[tIndex]->checkElement(testVec[tIndex]->size() / 2,
&tIntValue))
        std::cout << "incorrect assignment in vector %d\n" <<
tIndex << std::endl;
```

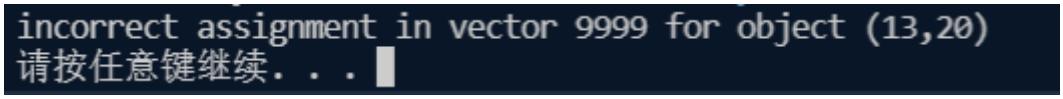
```
    tIndex = TESTSIZE - 4 + 3;
    myObject tObj(11, 15);
    testVec[tIndex]->setElement(testVec[tIndex]->size() / 2,
&tObj);
    if (!testVec[tIndex]->checkElement(testVec[tIndex]->size() / 2,
&tObj))
        std::cout << "incorrect assignment in vector %d\n" <<
tIndex << std::endl;

    myObject tObj1(13, 20);
    if (!testVec[tIndex]->checkElement(testVec[tIndex]->size() / 2,
&tObj1))
        std::cout << "incorrect assignment in vector " << tIndex <<
" for object (13,20)" << std::endl;
```

When testing the object tobj(13, 20), skip the step

```
testVec[tIndex]->setElement(testVec[tIndex]->size() / 2, &tObj);
```

on purpose, so that if error is output, it means our allocator is correct, which is the truth.



Test 4

```
#define _TEST_STD_
```

After making some chages to the source code of the testbench.cpp, this test can test the performance of the stl::allocator, divided into the allocate, resize, and delete parts.

What deserves your attention is that the default number of TESTSIZE and PICKSIZE is 10000, and 1000, respectively.

```
#define TESTSIZE 10000
#define PICKSIZE 1000
```

```
The runtime of 10000 allocate operations is 0.112s
The runtime of 1000 resize operations is 0.006s
The runtime of 10000 delete operations is 0.028s
Well done!
```

Test 5

```
#define _TEST_MYALLOCATOR_
```

Replace the stl::allocator and use MyAllocator instead, this time, the result is as below:

```
Myallocator
The runtime of 10000 allocate operations is 0.599s
The runtime of 1000 resize operations is 0.008s
incorrect assignment in vector 9999 for object (13,20)
The runtime of 10000 delete operations is 0.368s
Well done!
```

Test 6

```
#define _TEST_STD_PTA_
```

The test code is from PTA. Test 6 test the performance of stl::allocator.

```
std::allocator
The runtime of 10000 allocate operations is 0.671s
The runtime of 1000 resize operations is 0.057s
correct assignment in vecints: 3266
correct assignment in vecpts: 1631
```

Test 7

```
#define _TEST_MYALLOCATOR_PTA_
```

```
Myllocator
The runtime of 10000 allocate operations is 0.725s
The runtime of 1000 resize operations is 0.055s
correct assignment in vecints: 3266
correct assignment in vecpts: 1631
```

Now it is the result of MyAllocator.

Test 8

```
#define _TEST_WITH_FILE_
```

Read file to test the performance of different values_type. You can use

```
#define _TEST_FILE_
```

to construct random test file for testing.

Further Disscution

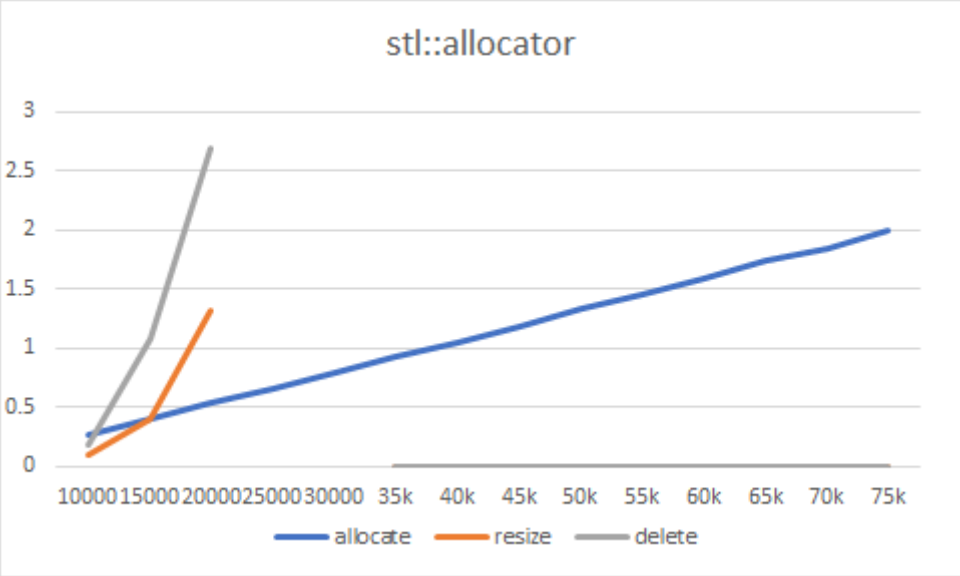
use larger TESTSIZE and PICKSIZE, to compare the performance of both stl::allocator and MyAllocator.

Result for stl::allocactor

"-" represents bad::alloc()

time: seconds

SIZE	10K	15K	20K	25K	30K	35K	40K	45K	50K	55K	60K	65K	70K	75K
allocate	0.263s	0.397s	0.535s	0.661s	0.799s	0.927s	1.05s	1.181s	1.328s	1.457s	1.589s	1.741s	1.849s	1.987s
resize	0.104s	0.395s	1.311s	-	-	-	-	-	-	-	-	-	-	-
delete	0.179s	1.076s	2.684s	-	-	-	-	-	-	-	-	-	-	-

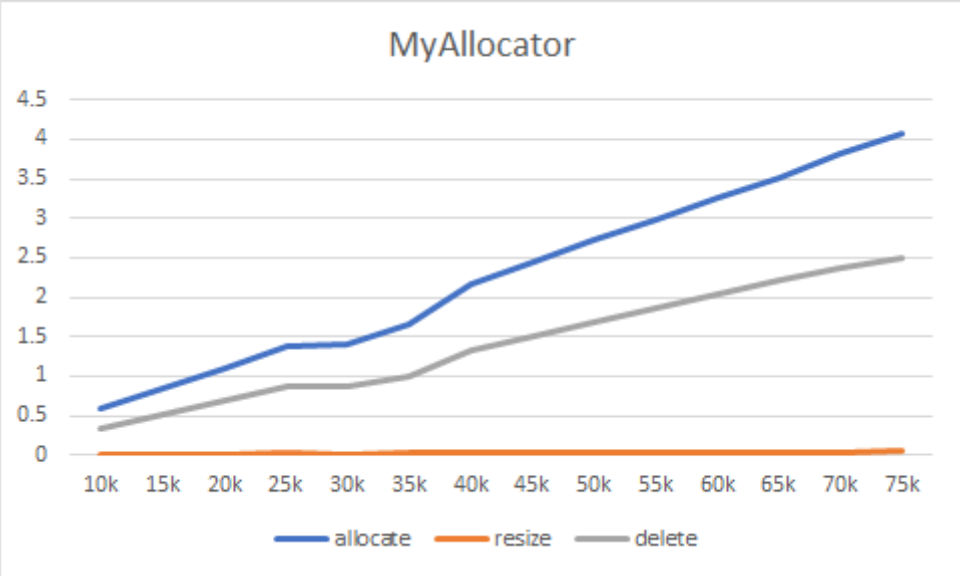


From the result of the allocator, we can see resize and delete grow rapidly, and that after the size is larger than 20k, resize and delete may cause bad::alloc().

The time cost of allocate is linearly growinig with size.

Result for MyAllocator

SIZE	10K	15K	20K	25K	30K	35K	40K	45K	50K	55K	60K	65K	70K	75K
allocate	0.58s	0.855s	1.101s	1.375s	1.392s	1.668s	2.167s	2.456s	2.717s	2.989s	3.259s	3.52s	3.819s	4.08s
resize	0.013s	0.015s	0.013s	0.022s	0.018s	0.023s	0.031s	0.029s	0.036s	0.038s	0.039s	0.041s	0.042s	0.049s
delete	0.332s	0.506s	0.698s	0.859s	0.858s	1.008s	1.33s	1.518s	1.687s	1.863s	2.031s	2.22s	2.371s	2.509s



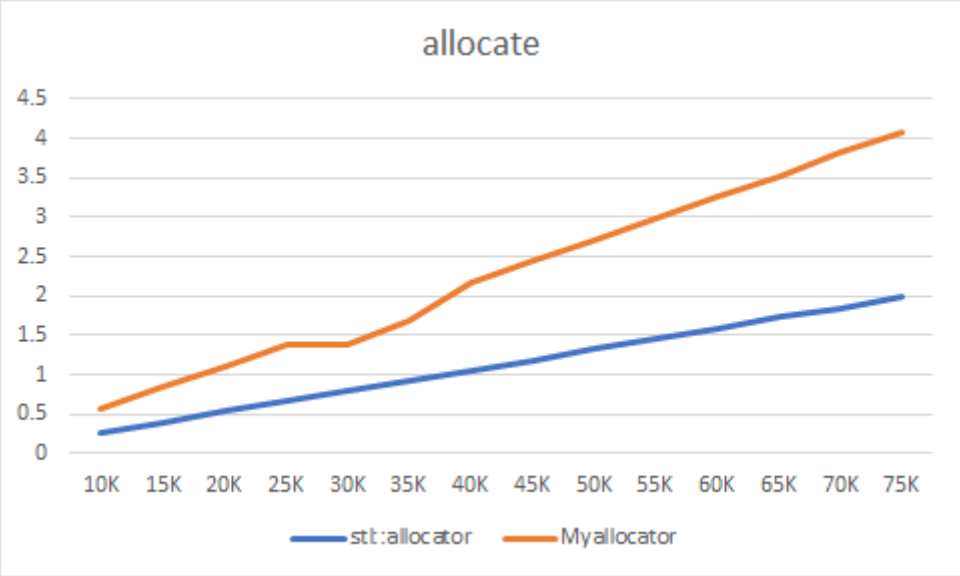
As for MyAllocator, the resize and delete operations are much faster, meanwhile the allocate operation is twice the time of stl::allocator.

Resize, delete and allocate are all linearly growing with size.

TESTSIZE

allocate

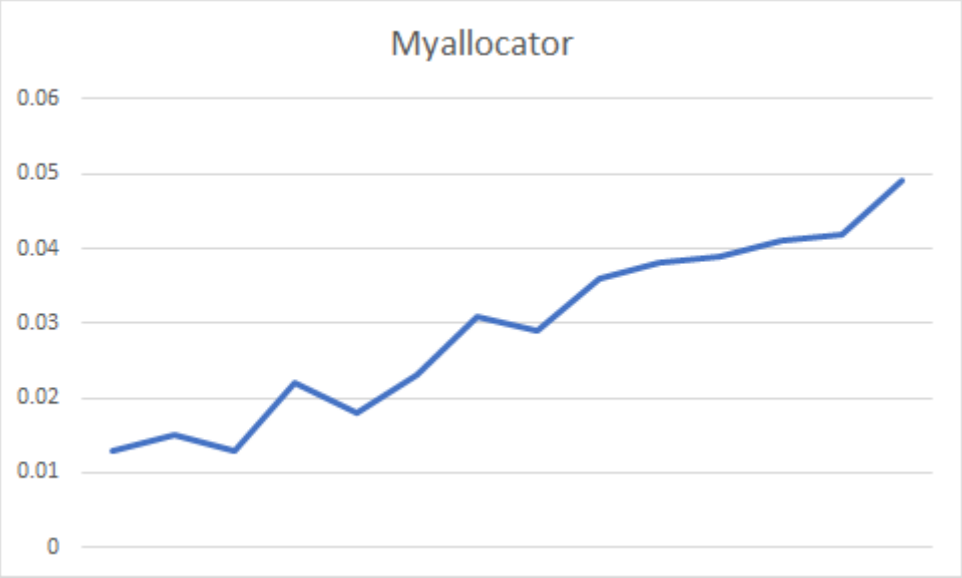
SIZE	10K	15K	20K	25K	30K	35K	40K	45K	50K	55K	60K	65K	70K	75K
stl::allocator	0.263	0.397	0.535	0.661	0.799	0.927	1.05	1.181	1.328	1.457	1.589	1.741	1.849	1.987
Myallocator	0.58	0.855	1.101	1.375	1.392	1.688	2.167	2.456	2.717	2.989	3.259	3.52	3.819	4.08



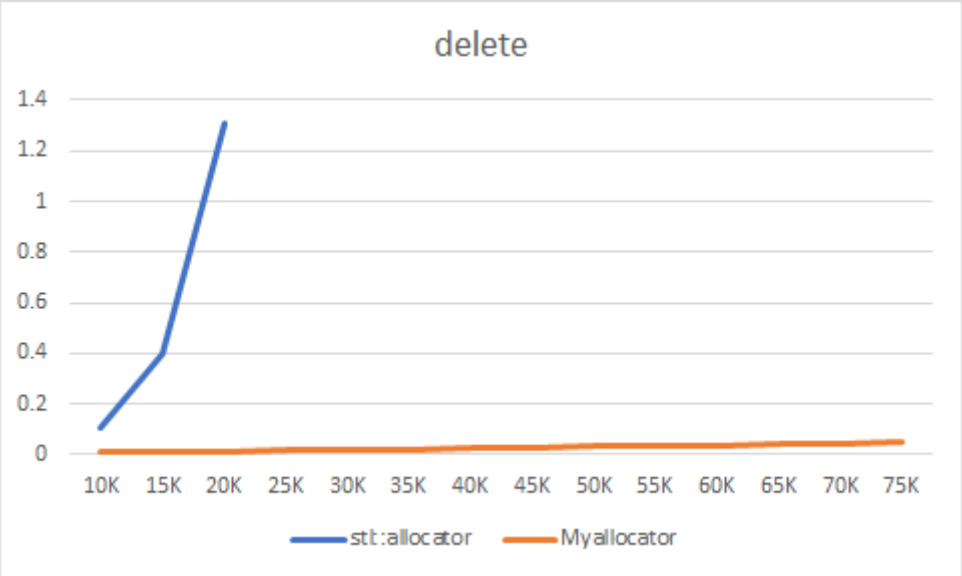
For allocate operation MyAllocator is nearly twice the time of stl::allocator.

delete

SIZE	10K	15K	20K	25K	30K	35K	40K	45K	50K	55K	60K	65K	70K	75K
stl::allocator	0.179	1.076	2.684	-	-	-	-	-	-	-	-	-	-	-
Myallocator	0.013	0.015	0.013	0.022	0.018	0.023	0.031	0.029	0.036	0.038	0.039	0.041	0.042	0.049



The delete time cost of Myallocator, we can see that its not very smooth, which means the memory-pool causes some fragment.

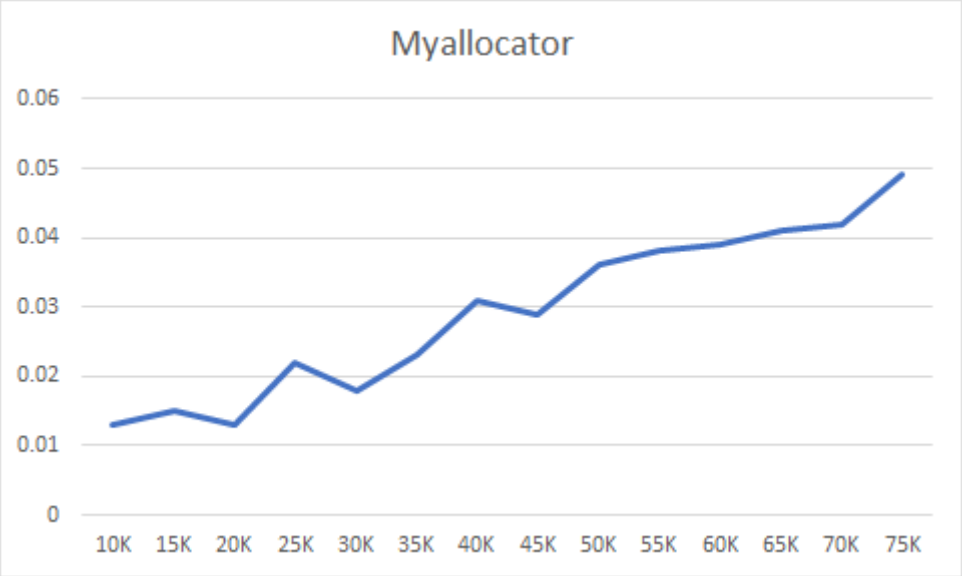


However, the time cost by deleting is much smaller than stl::allocactor, so we can draw the conclusion that MyAllocator avoids higher fragmentation and lower utilization of the physical memory.

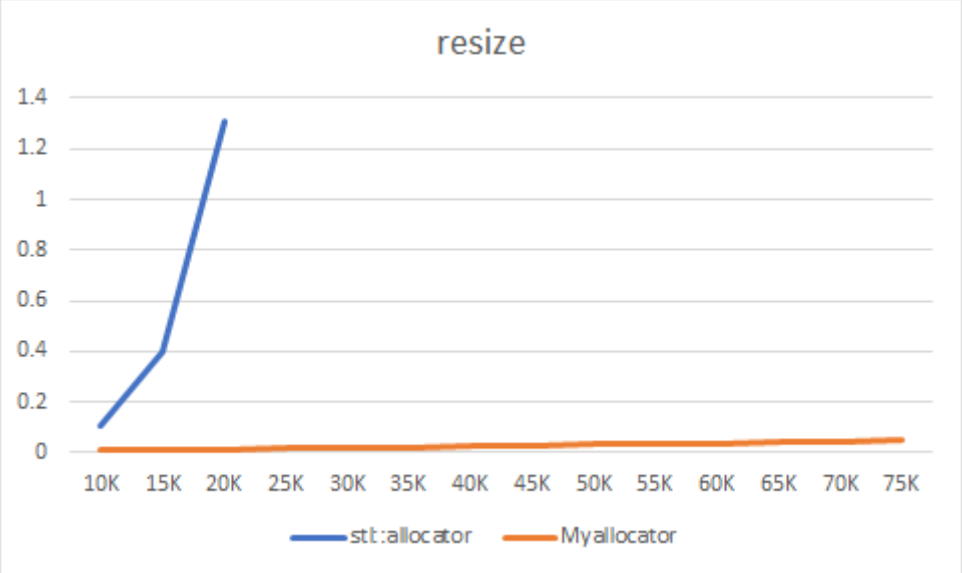
PICKSIZE

resize

SIZE	10K	15K	20K	25K	30K	35K	40K	45K	50K	55K	60K	65K	70K	75K
stl::allocator	0.104	0.395	1.311	-	-	-	-	-	-	-	-	-	-	-
Myallocator	0.013	0.015	0.013	0.022	0.018	0.023	0.031	0.029	0.036	0.038	0.039	0.041	0.042	0.049



The resize is still not very smooth, fragment will never be completely avoided.



Source Code

Source Code of myallocator.h

```
#ifndef MYALLOCATOR_H
#define MYALLOCATOR_H

#include "memorypool.h"
```

```
using namespace std;

template <class T>
class MyAllocator
{
public:
    typedef void _Not_user_specialized;
    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 size_t size_type;
    typedef ptrdiff_t difference_type;

    template <typename U> struct rebind
    {
        typedef MyAllocator<U> Other;
    };
    MyAllocator() = default;
    MyAllocator(const MyAllocator &myallocator) = default;
    MyAllocator(MyAllocator &&myallocator) = default;
    template <class U>
    MyAllocator(const MyAllocator<U> &myallocator) noexcept;
    pointer address(reference _Val) const noexcept{
        //return address of value
        return &_Val;
    }
    const_pointer address(const_reference _Val) const noexcept{
        return &_Val;
    }
    void deallocate(pointer _Ptr, size_type _Count){
        MP.deallocate(_Ptr,_Count* sizeof(value_type));
    }
    pointer allocate(size_type _Count) {
        if(auto res = static_cast<pointer>(MP.allocate(_Count *
sizeof(value_type))))
            return res;
        else throw bad_alloc();
    }
    template <class _Uty>
    void destroy(_Uty *_Ptr){
        _Ptr->~_Uty();
```

```
    }
    template <class _Objty, class... _Types>
    void construct(_Objty *_Ptr, _Types &&... _Args){
    new(_Ptr) _Objty(forward<_Types>(_Args)...);
    return;
    }
private:
    MemoryPool MP;
};

#endif
```

Source Code of memorypool.h

```
#include <iostream>

enum {__ALIGN = 8}; // 小型区块的上调边界
enum {__MAX_BYTES = 128}; // 小型区块的上限
enum {__NFREELISTS = __MAX_BYTES/__ALIGN}; // free-lists个数

// 维护16个free-lists，各自管理大小分别为8，16，24，32，40，48，56，64，
72，80，88，96，104，112，120，128bytes的小额区块
union obj
{
    union obj * free_list_link;
    char client_data[1];
};

// 以下是MemoryPool，参考stl源码的第二级配置器
// 注意，无"template型别参数"
class MemoryPool
{
public:
    typedef void _Not_user_specialized;
    typedef size_t size_type;

    static _Not_user_specialized * allocate(size_type n)
    {
        obj * volatile * my_free_list;
```

```
        obj * result;
        // 大于128就调用一级配置器
        if(n > (size_type) __MAX_BYTES)
            return malloc(n);

        // 寻找16个free lists中适当的一个
        my_free_list = free_list + FREELIST_INDEX(n);
        result = *my_free_list;
        if(result == nullptr){
            // 未找到可用的free list，准备重新填充free list
            return refill(ROUND_UP(n));
        }
        // 调整free list
        *my_free_list = result->free_list_link;
        return result;
    }

    static _Not_user_specialized deallocate(_Not_user_specialized *
p, size_type n)
    {
        obj * q = (obj*)p;
        obj * volatile * my_free_list;

        // 大于128就调用第一级配置器
        if(n > (size_type) __MAX_BYTES)
        {
            free(p);
            return;
        }

        // 寻找对应的free list
        my_free_list = free_list + FREELIST_INDEX(n);
        //调整free list，回收区块
        q->free_list_link = *my_free_list;
        *my_free_list = q;
    }

    static _Not_user_specialized * reallocate(_Not_user_specialized
* p, size_type old_sz, size_type new_sz);
private:
    //内存池起始位置
    static char *begin;
    // 内存池结束位置
```

```
static char *end;
static size_type memSize;

// 将bytes上调至8的倍数
static size_type ROUND_UP(size_type bytes)
{
    return (((bytes) + __ALIGN - 1) & ~(__ALIGN - 1));
}
// 16个free_list
static obj * volatile free_list[__NFREELISTS];
// 以下函数根据数据区块大小，决定使用第n号free-list。n从1起算
static size_type FREELIST_INDEX(size_type bytes)
{
    return ((bytes + __ALIGN - 1)/ __ALIGN - 1);
}

// 返回一个大小为n的对象，并可能加入大小为n的其他区块到free-list
static _Not_user_specialized* refill(size_type n)
{
    int nobjs = 20;
    // 调用chunk_alloc()，尝试取得nobjs个区块作为free list的新节点
    // 注意参数nobjs是pass by reference
    char* chunk = chunk_alloc(n, nobjs);

    obj * volatile * my_free_list;
    obj * result;
    obj * current_obj, * next_obj;
    int i;

    // 如果只获得一个区块，这个区块就分配给调用者用，free list无新节点
    if (1 == nobjs)
        return chunk;
    // 否则调整free list，纳入新节点
    my_free_list = free_list + FREELIST_INDEX(n);

    // 以下在chunk空间内建立free list
    result = (obj *)chunk; //这一块准备返回客端
    // 以下导引free list指向新配置的空间（取自内存池）
    *my_free_list = next_obj = (obj *)(chunk + n);
    // 以下将free list的各节点串联起来
    for (i = 1; ; i++) { // 从1开始，因为第0个将分会给客端
        current_obj = next_obj;
        next_obj = (obj *)((char *)next_obj + n);
```

```
            if (nobjs - 1 == i)
            {
                current_obj->free_list_link = nullptr;
                break;
            }
            else current_obj->free_list_link = next_obj;
        }
        return result;
    }

    // 假设size已经适当上调至8的倍数
    // 注意参数nobjs是pass by reference
    static char * chunk_alloc(size_type size, int& nobjs)
    {
        char * result;
        size_type total_bytes = size * nobjs;
        size_type bytes_left = end_free - start_free; //内存池剩余空间

        if (bytes_left >= total_bytes) {
            // 内存池剩余空间满足需求量
            result = start_free;
            start_free += total_bytes;
            return result;
        } else if (bytes_left >= size) {
            // 内存池剩余空间不能完全满足需求量，但足够供应一个及以上的区块
            nobjs = bytes_left / size;
            total_bytes = size * nobjs;
            result = start_free;
            start_free += total_bytes;
            return result;
        } else {
            // 内存池剩余空间连一个区块的大小都无法提供
            size_type bytes_to_get = 2 * total_bytes +
ROUND_UP(heap_size >> 4);
            // 以下试着让内存池中的残余零头还有利用价值
            if (bytes_left > 0) {
                // 内存池内还有一些零头，先配给适当的free list
                // 首先寻找适当的free list
                obj * volatile * my_free_list = free_list +
FREELIST_INDEX(bytes_left);
                // 调整free list，将内存池中的剩余空间编入
                ((obj *)start_free)->free_list_link =
*my_free_list;
```

```
                *my_free_list = (obj *)start_free;
            }

            // 配置heap空间，用来补充内存池
            start_free = (char *)malloc(bytes_to_get);
            if (0 == start_free) {
                // heap空间不足，malloc()失败
                int i;
                obj * volatile * my_free_list, *p;
                // 以下搜寻“尚未用区块，且区块足够大”之free list
                for (i = size; i < __MAX_BYTES; i += __ALIGN) {
                    my_free_list = free_list + FREELIST_INDEX(i);
                    p = *my_free_list;
                    if (0 != p) { // free list内尚有未用区块
                        // 调整free list以释放出为用区块
                        *my_free_list = p->free_list_link;
                        start_free = (char *)p;
                        end_free = start_free + i;
                        // 递归调用自己，为了修正nobjs
                        return(chunk_alloc(size, nobjs));
                        // 注意，任何残余零头终将被编入适当的free-list中备
用
                    }
                }
                end_free = nullptr;
                start_free = (char *)malloc(bytes_to_get);
            }
            heap_size += bytes_to_get;
            end_free = start_free + bytes_to_get;
            // 递归调用自己，为了修正nobjs
            return chunk_alloc(size, nobjs);
        }
    }

    // Chunk allocation state
    static char * start_free; // 内存池起始位置，只在chunk_alloc()中变化
    static char * end_free; // 内存池结束位置，只在chunk_alloc()中变化
    static size_type heap_size;
};

char *MemoryPool::start_free = nullptr;
char *MemoryPool::end_free = nullptr;
size_t MemoryPool::heap_size = 0;
```

```
obj * volatile MemoryPool::free_list[__NFREELISTS] = {nullptr};
```

Source Code of test.cpp

```
// testallocator.cpp : 定义控制台应用程序的入口点。
//
#include "myallocator.h"
#include <iostream>
#include <chrono>
#include <random>
#include <random>
#include <vector>
#include <limits>
#include <ctime>
#include <memory>
#include <fstream>
#include <cstring>
#include <cstdlib>
#include <string>

using namespace std;
using Point2D = std::pair<int, int>;

#define INT 1
#define FLOAT 2
#define DOUBLE 3
#define CLASS  4

#define TESTSIZE 10000
#define PICKSIZE 1000

// #define _TEST_STD_SIMPLE_ //检验替换sdt::allocator对vector的
push_back性能
// #define _TEST_SIMPLE_ //检验替换allocator后vector的push_back性能
// #define _TEST_CORRECTNESS_ //测试程序正确性的代码段
// #define _TEST_STD_ //测试std::allocator运行速度的代码段
 #define _TEST_MYALLOCATOR_ //测试myallocator运行速度的代码段
// #define _TEST_STD_PTA_ //PTA提供的测试代码段测试std的运行速度
```

```
// #define _TEST_MYALLOCATOR_PTA_ //PTA提供的测试代码段测试myallocator
的运行速度
// #define _TEST_FILE_ //自动生成测试样例文件的代码段
// #define _TEST_WITH_FILE_ //用自动生成的文件测试的代码段

class vecWrapper {
public:
    vecWrapper() {
        m_pVec = NULL;
        m_type = INT;
    }
    virtual ~vecWrapper() {}
public:
    void setPointer(int type, void* pVec) { m_type = type; m_pVec =
pVec; }
    virtual void visit(int index) = 0;
    virtual int size() = 0;
    virtual void resize(int size) = 0;
    virtual bool checkElement(int index, void* value) = 0;
    virtual void setElement(int idex, void* value) = 0;
protected:
    int m_type;
    void* m_pVec;
};

#ifdef _TEST_STD_
template<class T>
class vecWrapperT : public vecWrapper {
public:
    vecWrapperT(int type, vector<T, allocator<T> >* pVec) {
        m_type = type;
        m_pVec = pVec;
    }
    virtual ~vecWrapperT() {
        if (m_pVec)
            delete ((vector<T, allocator<T> > *)m_pVec);
    }
public:
    virtual void visit(int index) {
        T temp = (*(vector<T, allocator<T> > *)m_pVec)[index];
    }
    virtual int size() {
        return ((vector<T, allocator<T> > *)m_pVec)->size();
```

```
    }
    virtual void resize(int size) {
        ((vector<T, allocator<T> > *)m_pVec)->resize(size);
    }
    virtual bool checkElement(int index, void* pValue) {
        T temp = (*(vector<T, allocator<T> > *)m_pVec)[index];
        if (temp == (*((T*)pValue)))
            return true;
        else
            return false;
    }

    virtual void setElement(int index, void* value) {
        (*(vector<T, allocator<T> > *)m_pVec)[index] = *
((T*)value);
    }
};
#endif

#ifndef _TEST_STD_
template<class T>
class vecWrapperT : public vecWrapper
{
public:
    vecWrapperT(int type, std::vector<T, MyAllocator<T> > *pVec)
    {
        m_type = type;
        m_pVec = pVec;
    }
    virtual ~vecWrapperT() {
        if (m_pVec)
            delete ((std::vector<T, MyAllocator<T> > *)m_pVec);
    }
public:
    virtual void visit(int index)
    {
        T temp = (*(std::vector<T, MyAllocator<T> > *)m_pVec)
[index];
    }
    virtual int size()
    {
        return ((std::vector<T, MyAllocator<T> > *)m_pVec)->size();
    }
```

```
    virtual void resize(int size)
    {
        ((std::vector<T, MyAllocator<T> > *)m_pVec)->resize(size);
    }
    virtual bool checkElement(int index, void *pValue)
    {
        T temp = (*(std::vector<T, MyAllocator<T> > *)m_pVec)
[index];
        if (temp == (*((T *)pValue)))
            return true;
        else
            return false;
    }

    virtual void setElement(int index, void *value)
    {
        (*(std::vector<T, MyAllocator<T> > *)m_pVec)[index] = *((T
*)value);
    }
};
#endif

class myObject {
public:
    myObject() : m_X(0), m_Y(0) {}
    myObject(int t1, int t2) :m_X(t1), m_Y(t2) {}
    myObject(const myObject& rhs) { m_X = rhs.m_X; m_Y = rhs.m_Y; }
    ~myObject() { /*cout << "my object destructor called" <<
endl;*/ }
    bool operator == (const myObject& rhs) {
        if ((rhs.m_X == m_X) && (rhs.m_Y == m_Y))
            return true;
        else
            return false;
    }
protected:
    int m_X;
    int m_Y;
};

#ifdef _TEST_STD_SIMPLE_

int main ()
```

```
{
    clock_t begin, end;
    std::vector<int> integer;

    cout << "std::llocator" << endl;
    begin = clock();
    for (auto i = 0; i < TESTSIZE; i++)
        integer.push_back(i);
    end = clock();
    cout << "The runtime of " << TESTSIZE << " push operations is "
<< (double)(end - begin) / CLOCKS_PER_SEC << "s" << endl;

    std::vector<vector<int>> vectorTmp;
    cout << "MyAllocator" << endl;
    begin = clock();
    for (auto i = 0; i < TESTSIZE; i++)
        vectorTmp.push_back(integer);
    end = clock();
    cout << "The runtime of " << TESTSIZE << " push operations is "
<< (double)(end - begin) / CLOCKS_PER_SEC << "s" << endl;

    return 0;
}
#endif

#ifdef _TEST_SIMPLE_

int main ()
{
    clock_t begin, end;
    std::vector<int, MyAllocator<int> > integer;

    cout << "MyAllocator" << endl;
    begin = clock();
    for (auto i = 0; i < TESTSIZE; i++)
        integer.push_back(i);
    end = clock();
    cout << "The runtime of " << TESTSIZE << " push operations is "
<< (double)(end - begin) / CLOCKS_PER_SEC << "s" << endl;

    std::vector<vector<int, MyAllocator<int> >,
MyAllocator<vector<int, MyAllocator<int> > > > vectorTmp;
    cout << "MyAllocator" << endl;
```

```
    begin = clock();
    for (auto i = 0; i < TESTSIZE; i++)
        vectorTmp.push_back(integer);
    end = clock();
    cout << "The runtime of " << TESTSIZE << " push operations is "
<< (double)(end - begin) / CLOCKS_PER_SEC << "s" << endl;

    return 0;
}
#endif

#ifdef  _TEST_CORRECTNESS_

int main()
{
    vecWrapper **testVec;
    testVec = new vecWrapper*[TESTSIZE];

    int tIndex, tSize;
    //test allocator
    for (int i = 0; i < TESTSIZE - 4; i++)
    {
        tSize = (int)((float)rand() / (float)RAND_MAX * 10000);
        vecWrapperT<int> *pNewVec = new vecWrapperT<int>(INT, new
std::vector<int, MyAllocator<int>>(tSize));
        testVec[i] = (vecWrapper *)pNewVec;
    }

    for (int i = 0; i < 4; i++)
    {
        tSize = (int)((float)rand() / (float)RAND_MAX * 10000);
        vecWrapperT<myObject> *pNewVec = new vecWrapperT<myObject>
(CLASS, new std::vector<myObject, MyAllocator<myObject>>(tSize));
        testVec[TESTSIZE - 4 + i] = (vecWrapper *)pNewVec;
    }

    //test resize
    for (int i = 0; i < 100; i++)
    {
        tIndex = (int)((float)rand() / (float)RAND_MAX *
(float)TESTSIZE);
        tSize = (int)((float)rand() / (float)RAND_MAX *
(float)TESTSIZE);
```

```
        testVec[tIndex]->resize(tSize);
    }

    //test assignment
    tIndex = (int)((float)rand() / (float)RAND_MAX * (TESTSIZE - 4
- 1));
    int tIntValue = 10;
    testVec[tIndex]->setElement(testVec[tIndex]->size() / 2,
&tIntValue);
    if (!testVec[tIndex]->checkElement(testVec[tIndex]->size() / 2,
&tIntValue))
        std::cout << "incorrect assignment in vector %d\n" <<
tIndex << std::endl;

    tIndex = TESTSIZE - 4 + 3;
    myObject tObj(11, 15);
    testVec[tIndex]->setElement(testVec[tIndex]->size() / 2,
&tObj);
    if (!testVec[tIndex]->checkElement(testVec[tIndex]->size() / 2,
&tObj))
        std::cout << "incorrect assignment in vector %d\n" <<
tIndex << std::endl;

    myObject tObj1(13, 20);
    if (!testVec[tIndex]->checkElement(testVec[tIndex]->size() / 2,
&tObj1))
        std::cout << "incorrect assignment in vector " << tIndex <<
" for object (13,20)" << std::endl;

    for (int i = 0; i < TESTSIZE; i++)
        delete testVec[i];

    delete[]testVec;
    system("pause");
    return 0;

}
#endif

#ifdef _TEST_STD_

int main() {
    vecWrapper** testVec;
```

```
    testVec = new vecWrapper *[TESTSIZE];
    clock_t begin, end;
    int tIndex, tSize;

    cout << "std::allocator" << endl;
    begin = clock();
    for (int i = 0; i < TESTSIZE - 4; i++) {
        tSize = (int)((float)rand() / (float)RAND_MAX * 10000);
        vecWrapperT<int>* pNewVec = new vecWrapperT<int>(INT, new
vector<int, allocator<int>>(tSize));
        testVec[i] = (vecWrapper*)pNewVec;
    }

    for (int i = 0; i < 4; i++) {
        tSize = (int)((float)rand() / (float)RAND_MAX * 10000);
        vecWrapperT<myObject>* pNewVec = new vecWrapperT<myObject>
(CLASS, new vector<myObject, allocator<myObject>>(tSize));
        testVec[TESTSIZE - 4 + i] = (vecWrapper*)pNewVec;
    }
    end = clock();
    cout << "The runtime of " << TESTSIZE << " allocate operations
is " << (double)(end - begin) / CLOCKS_PER_SEC << "s" << endl;

    //test resize
    begin = clock();
    for (int i = 0; i < PICKSIZE; i++) {
        tIndex = (int)((float)rand() / (float)RAND_MAX *
(float)TESTSIZE);
        tSize = (int)((float)rand() / (float)RAND_MAX *
(float)TESTSIZE);
        testVec[tIndex]->resize(tSize);
    }
    end = clock();
    cout << "The runtime of " << PICKSIZE << " resize operations is
" << (double)(end - begin) / CLOCKS_PER_SEC << "s" << endl;

    //test assignment
    tIndex = (int)((float)rand() / (float)RAND_MAX * (TESTSIZE - 4
- 1));
    int tIntValue = 10;
    testVec[tIndex]->setElement(testVec[tIndex]->size() / 2,
&tIntValue);
```

```
    if (!testVec[tIndex]->checkElement(testVec[tIndex]->size() / 2,
&tIntValue))
        cout << "incorrect assignment in vector %d\n" << tIndex <<
endl;

    tIndex = TESTSIZE - 4 + 3;
    myObject tObj(11, 15);
    testVec[tIndex]->setElement(testVec[tIndex]->size() / 2,
&tObj);
    if (!testVec[tIndex]->checkElement(testVec[tIndex]->size() / 2,
&tObj))
        cout << "incorrect assignment in vector %d\n" << tIndex <<
endl;

    myObject tObj1(13, 20);
    testVec[tIndex]->setElement(testVec[tIndex]->size() / 2,
&tObj1);
    if (!testVec[tIndex]->checkElement(testVec[tIndex]->size() / 2,
&tObj1))
        cout << "incorrect assignment in vector " << tIndex << "
for object (13,20)" << endl;

    begin = clock();
    for (int i = 0; i < TESTSIZE; i++)
        delete testVec[i];
    delete[]testVec;
    end = clock();
    cout << "The runtime of " << TESTSIZE << " delete operations is
" << (double)(end - begin) / CLOCKS_PER_SEC << "s" << endl;
    cout << "Well done!" << endl;
    return 0;

}
#endif

#ifdef _TEST_MYALLOCATOR_

int main()
{
    vecWrapper** testVec;
    testVec = new vecWrapper *[TESTSIZE];
    clock_t begin, end;
    int tIndex, tSize;
```

```
    cout << "Myallocator" << endl;
    begin = clock();
    //test MyAllocator
    for (int i = 0; i < TESTSIZE - 4; i++)
    {
        tSize = (int)((float)rand() / (float)RAND_MAX * 10000);
        vecWrapperT<int> *pNewVec = new vecWrapperT<int>(INT, new
std::vector<int, MyAllocator<int> >(tSize));
        testVec[i] = (vecWrapper *)pNewVec;
    }

    for (int i = 0; i < 4; i++)
    {
        tSize = (int)((float)rand() / (float)RAND_MAX * 10000);
        vecWrapperT<myObject> *pNewVec = new vecWrapperT<myObject>
(CLASS, new std::vector<myObject, MyAllocator<myObject> >(tSize));
        testVec[TESTSIZE - 4 + i] = (vecWrapper *)pNewVec;
    }
    end = clock();
    cout << "The runtime of " << TESTSIZE << " allocate operations
is " << (double)(end - begin) / CLOCKS_PER_SEC << "s" << endl;

    //test resize
    begin = clock();
    for (int i = 0; i < 100; i++)
    {
        tIndex = (int)((float)rand() / (float)RAND_MAX *
(float)TESTSIZE);
        tSize = (int)((float)rand() / (float)RAND_MAX *
(float)TESTSIZE);
        testVec[tIndex]->resize(tSize);
    }
    end = clock();
    cout << "The runtime of " << PICKSIZE << " resize operations is
" << (double)(end - begin) / CLOCKS_PER_SEC << "s" << endl;

    //test assignment
    tIndex = (int)((float)rand() / (float)RAND_MAX * (TESTSIZE - 4
- 1));
    int tIntValue = 10;
    testVec[tIndex]->setElement(testVec[tIndex]->size() / 2,
&tIntValue);
```

```
    if (!testVec[tIndex]->checkElement(testVec[tIndex]->size() / 2,
&tIntValue))
        cout << "incorrect assignment in vector %d\n" << tIndex <<
std::endl;

    tIndex = TESTSIZE - 4 + 3;
    myObject tObj(11, 15);
    testVec[tIndex]->setElement(testVec[tIndex]->size() / 2,
&tObj);
    if (!testVec[tIndex]->checkElement(testVec[tIndex]->size() / 2,
&tObj))
        cout << "incorrect assignment in vector %d\n" << tIndex <<
std::endl;

    myObject tObj1(13, 20);
    if (!testVec[tIndex]->checkElement(testVec[tIndex]->size() / 2,
&tObj1))
        std::cout << "incorrect assignment in vector " << tIndex <<
" for object (13,20)" << std::endl;

    begin = clock();
    for (int i = 0; i < TESTSIZE; i++)
        delete testVec[i];
    delete[]testVec;

    end = clock();
    cout << "The runtime of " << TESTSIZE << " delete operations is
" << (double)(end - begin) / CLOCKS_PER_SEC << "s" << endl;
    cout << "Well done!" << endl;

    system("pause");
    return 0;

}

#endif

#ifdef _TEST_STD_PTA_

int main()
{
    std::random_device rd;
    std::mt19937 gen(rd());
```

```
    std::uniform_int_distribution<> dis(1, TESTSIZE);

    clock_t begin, end;

    cout << "std::allocator" << endl;
    begin = clock();

    // vector creation
    using IntVec = std::vector<int, allocator<int>>;
    std::vector<IntVec, allocator<IntVec>> vecints(TESTSIZE);
    for (int i = 0; i < TESTSIZE; i++)
        vecints[i].resize(dis(gen));

    using PointVec = std::vector<Point2D, allocator<Point2D>>;
    std::vector<PointVec, allocator<PointVec>> vecpts(TESTSIZE);
    for (int i = 0; i < TESTSIZE; i++)
        vecpts[i].resize(dis(gen));
    end = clock();
    cout << "The runtime of " << TESTSIZE << " allocate operations
is " << (double)(end - begin) / CLOCKS_PER_SEC << "s" << endl;

    // vector resize
    begin = 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);
    }
    end = clock();
    cout << "The runtime of " << PICKSIZE << " resize operations is
" << (double)(end - begin) / CLOCKS_PER_SEC << "s" << endl;

    // 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;
    }

    return 0;
}
#endif

#ifdef _TEST_MYALLOCATOR_PTA_

int main()
{
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(1, TESTSIZE);
    clock_t begin, end;

    // vector creation
    cout << "Myllocator" << endl;
    begin = clock();
    using IntVec = std::vector<int, allocator<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, allocator<Point2D>>;
    std::vector<PointVec, MyAllocator<PointVec>> vecpts(TESTSIZE);
    for (int i = 0; i < TESTSIZE; i++)
        vecpts[i].resize(dis(gen));
    end = clock();
```

```
    cout << "The runtime of " << TESTSIZE << " allocate operations
is " << (double)(end - begin) / CLOCKS_PER_SEC << "s" << endl;

    // vector resize
    begin = 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);
    }
    end = clock();
    cout << "The runtime of " << PICKSIZE << " resize operations is
" << (double)(end - begin) / CLOCKS_PER_SEC << "s" << endl;

    // 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;
    }

    return 0;
```

```
}
#endif

#ifdef _TEST_FILE_

int main() {
    int test_size, pick_size;
    int tIndex, tSize;
    int type;
    string stype;
    string filename;
    cout << "Input the filename: ";
    cin >> filename;
    cout << "Input the number to be allocated: ";
    cin >> test_size;
    cout << "Input the number to be resized: ";
    cin >> pick_size;

    ofstream fout(("test//" + filename).c_str(), ios::out);
    if (!fout) {
        cout << "Error opening file." << endl;
        return 0;
    }
    fout << test_size << endl;
    srand(time(0));
    for (int i = 0; i < test_size; i++) {
        tSize = (int)((float)rand() / (float)RAND_MAX * 10000);
        type = (int)((float)rand() / (float)RAND_MAX * 4) + 1;
        switch (type) {
            case 1: stype = "int"; break;
            case 2: stype = "float"; break;
            case 3: stype = "double"; break;
            case 4: stype = "class"; break;
            default: break;
        }
        fout << tSize << " " << stype << endl;
        cout << tSize << " " << stype << endl;
    }
    fout << pick_size << endl;
    for (int i = 0; i < pick_size; i++) {
        tIndex = (int)((float)rand() / (float)RAND_MAX * 10000);
        tSize = (int)((float)rand() / (float)RAND_MAX * 10000);
        if (i == pick_size - 1) fout << tIndex << " " << tSize;
```

```
        else fout << tIndex << " " << tSize << endl;
        cout << tIndex << " " << tSize << endl;
    }
    fout.close();
    cout << "Finish successfully!" << endl;
    system("pause");
    return 0;
}
#endif

#ifdef _TEST_WITH_FILE_

int main()
{
    int TestSize;
    string filename;
    cout << "Input the filename: ";
    cin >> filename;
    ifstream file(("test\\" + filename).c_str(), ios::in);
    if (!file)//test if the file is opened successfully
    {
        cout << "Error opening file." << endl;
        return 0;
    }
    file >> TestSize;

    vecWrapper **testVec;
    testVec = new vecWrapper*[TestSize];
    clock_t start, stop;//the start and end of the runtime
    double runtime = 0.0;

    int tIndex, tSize;
    string type;

    //test allocator
    for (int i = 0; i < TestSize; i++)
    {
        file >> tSize >> type;
        if ((!strcmp(type.c_str(), "int")) ||
(!strcmp(type.c_str(), "Int")))
        {
            start = clock();
```

```
            vecWrapperT<int> *pNewVec = new vecWrapperT<int>(INT,
new std::vector<int, MyAllocator<int>>(tSize));
            testVec[i] = (vecWrapper *)pNewVec;
            stop = clock();
            runtime += (double)(stop - start) / CLOCKS_PER_SEC;
        }
        else if ((!strcmp(type.c_str(), "float")) ||
(!strcmp(type.c_str(), "Float")))
        {
            start = clock();
            vecWrapperT<float> *pNewVec = new vecWrapperT<float>
(INT, new std::vector<float, MyAllocator<float>>(tSize));
            testVec[i] = (vecWrapper *)pNewVec;
            stop = clock();
            runtime += (double)(stop - start) / CLOCKS_PER_SEC;
        }
        else if ((!strcmp(type.c_str(), "double")) ||
(!strcmp(type.c_str(), "Double")))
        {
            start = clock();
            vecWrapperT<double> *pNewVec = new vecWrapperT<double>
(INT, new std::vector<double, MyAllocator<double>>(tSize));
            testVec[i] = (vecWrapper *)pNewVec;
            stop = clock();
            runtime += (double)(stop - start) / CLOCKS_PER_SEC;
        }
        else if ((!strcmp(type.c_str(), "class")) ||
(!strcmp(type.c_str(), "Class")))
        {
            start = clock();
            vecWrapperT<myObject> *pNewVec = new
vecWrapperT<myObject>(CLASS, new std::vector<myObject,
MyAllocator<myObject>>(tSize));
            testVec[i] = (vecWrapper *)pNewVec;
            stop = clock();
            runtime += (double)(stop - start) / CLOCKS_PER_SEC;
        }
    }
    std::cout << "The runtime of " << TESTSIZE << " allocate
operations is " << runtime << " s" << endl;

    //test resize
    double runtime2 = 0.0;
```

```
    int count = 0;
    while (!file.eof())
    {
        file >> tIndex >> tSize;
        if (tSize == 0) break;
        start = clock();
        testVec[tIndex]->resize(tSize);
        stop = clock();
        runtime2 += (double)(stop - start) / CLOCKS_PER_SEC;
        count++;
    }
    std::cout << "The runtime of " << count << " resize operations
is " << runtime2 << " s" << endl;

    file.close();//close the file

    double runtime3 = 0.0;
    for (int i = 0; i < TESTSIZE; i++)
        delete testVec[i];
    std::cout << "circle delete OK!" << endl;
    delete[]testVec;

    std::cout << "delete OK!" << endl;
    stop = clock();
    runtime3 += (double)(stop - start) / CLOCKS_PER_SEC;
    std::cout << "The runtime of delete operations is " << runtime3
<< " s" << endl;

    system("pause");
    return 0;
}
#endif
```