My Allocator

Date: June-4th,2022

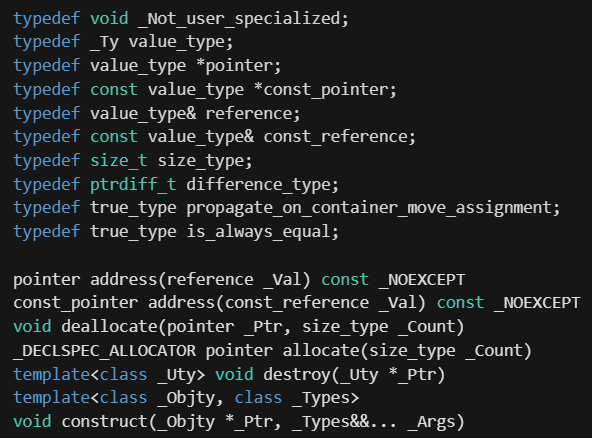
author:韩恺荣

**Chapter One:介绍**

在这个实验中，我们要自定义一个内存池，代替标准模板库中的std::allocator,并对他们的性能进行比较和分析。

一个使用allocator的简单实例：

此外，为保证自定义的内存池能够正确的运行，我们还需要为我们的内存池保留如下的接口:



**Chapter 2: 实验环境：**

C++标准：C++20（C++17也可以）

IDE:Visual Studio2022

系统：windows10

内存大小：16GB

**Chapter 3: 算法描述：**

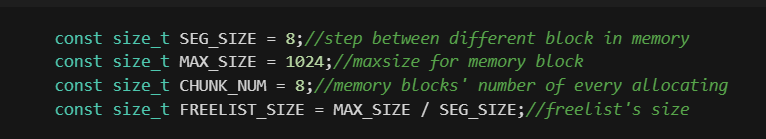
我实现内存池的方式综述如下：



对于一个内存池，我们使用freelist来对该内存池的内容进行管理，具体的管理方式为，我们将我们将要处理的内存对象分为两大类，一类的大小大于1024字节（代码中定义为MAX\_SIZE），另一类的大小小于1024字节，对于大于1024字节的对象，我们采用malloc的方式直接申请内存，而对于小于1024字节的对象，我们将这部分对象以8字节为步长分为1024/8个块区，即128块区，分别是8、16、32……1016、1024，每一个块区都有一个freelist指向他们的内存地址，，当我要处理一个大小为x（x不大于1024)的对象时，我们就将他放在距离他最近且比他大的8的整数倍大小的内存块上，如一个对象大小为10，那我们将他最终放置于内存大小为16的块区中，而寻找内存大小为16的块区是通过freelist实现的，初始化时将所有的freelist链表头部设置为NULL，之后按需扩张我们的内存池即可。

具体的函数描述如下：

（1）变量说明：

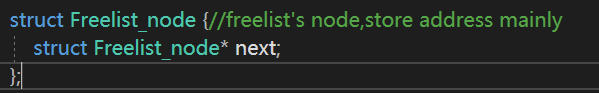


SEG\_SIZE:步长，表示内存大小的间隔

MAX\_SIZE：放置于freelist指向内存的最大对象大小

CHUNK\_NUM：每次分配的内存块数

FREELIST\_SIZE：freelist大小



freelist的节点内容如上图

（2）宏定义说明：

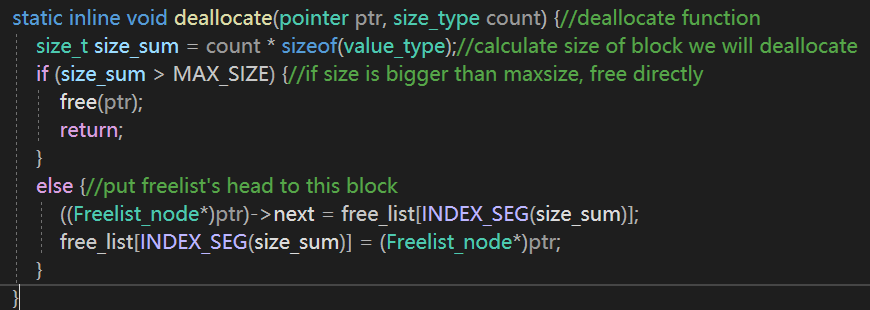


计算距离e最近且不小于e的8的整数倍数字再除以8并减一，如10经过处理变成1，表示他将连接在freelist[1]这个节点指向的内存

（3）重要函数说明：

a）回收函数：

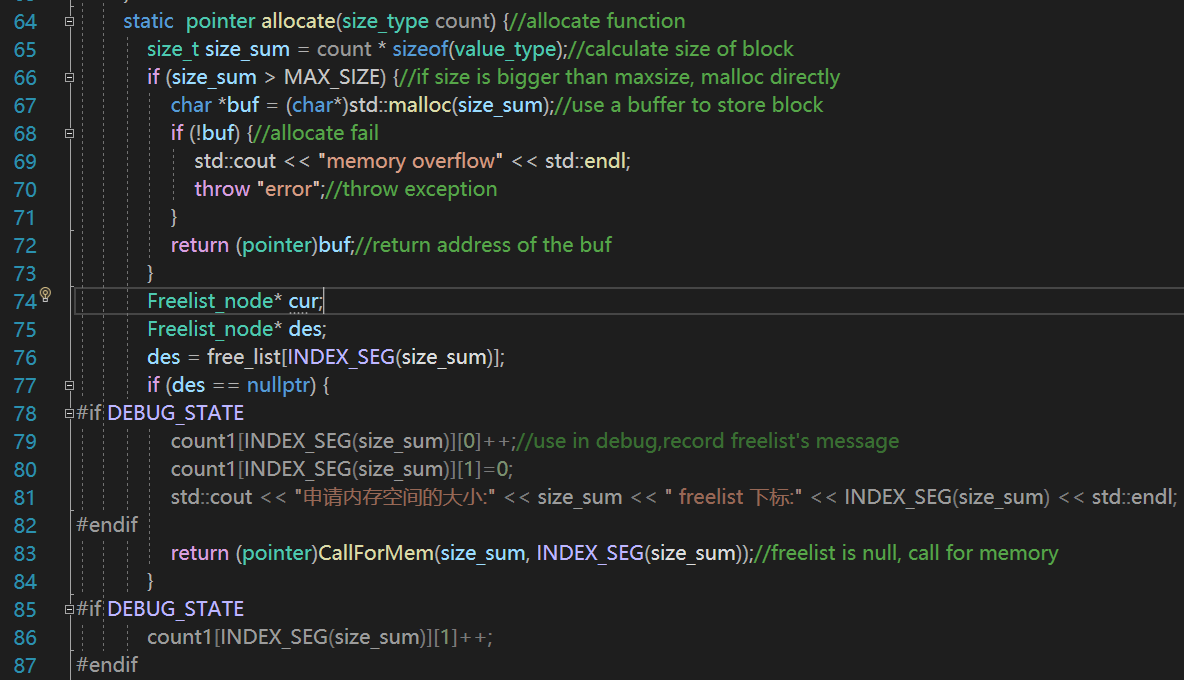
大于maxsize的对象直接调用free方法，否则将freelist对应的节点指向下一个即可。

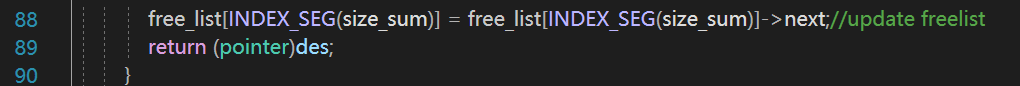


b）分配函数：

根据对象的大小检测和分类，大于1024直接调用malloc函数，否则计算对象大小对应的freelist的下标，然后使用freelist指向的区域放置该对象即可。

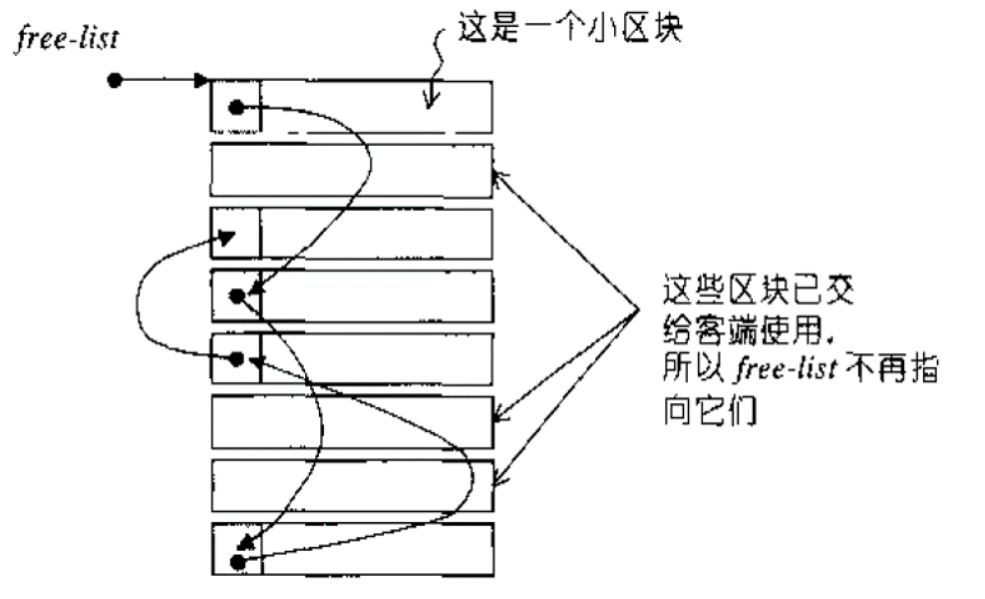
但如果freelist为null，则调用CallForMem函数向堆栈申请内存空间。

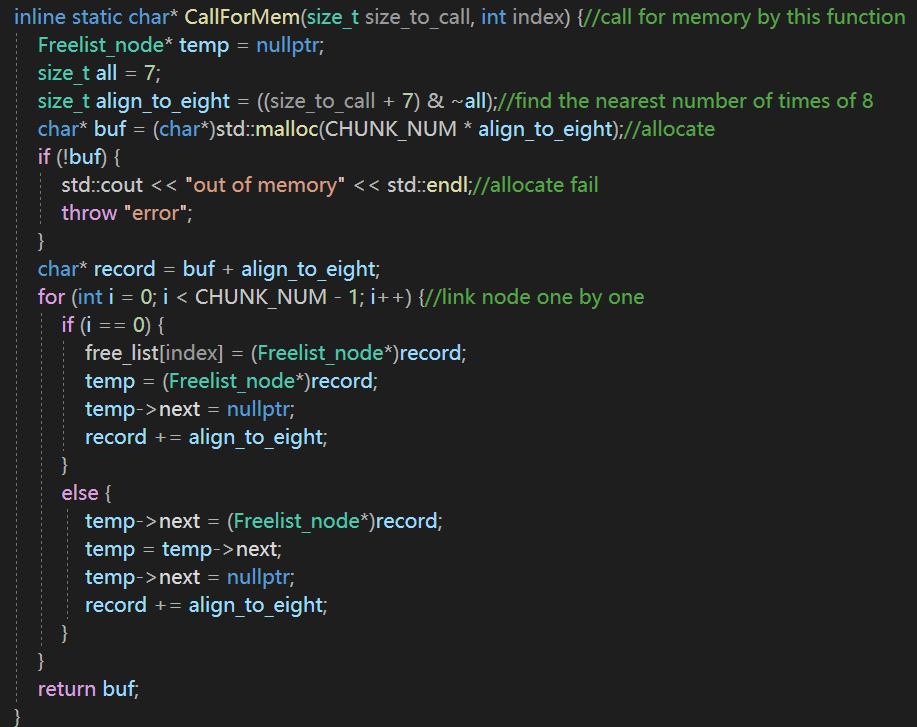




c）内存申请函数：

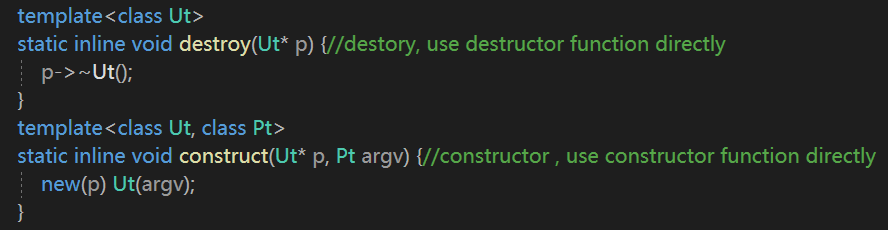
当freelist是null时，证明内存池中无内存存放该对象，即可调用该函数，该函数向内存空间申请一段内存，大小为CHUNK\_NUM \* align\_to\_eight，其中CHUNK\_NUM是内存的块数，align\_to\_eight是对象大小距离最近但不小于的8的整数倍的数值，然后并用freelist将这段内存串联起来。



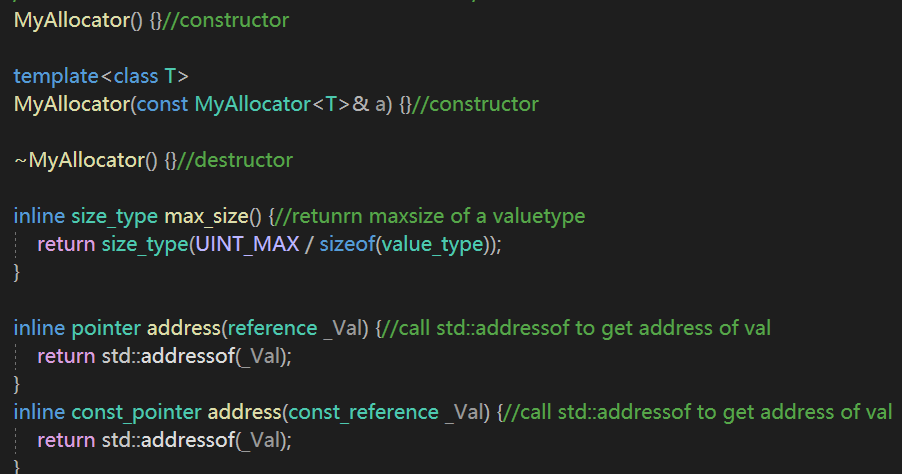


d）其他函数：

构造和摧毁函数直接调用对象类型的构造和析构函数即可

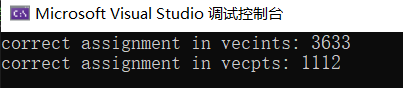


内存池的构造函数和max\_size()、address()方法都简单仿照std::allocator的函数调用方式即可：



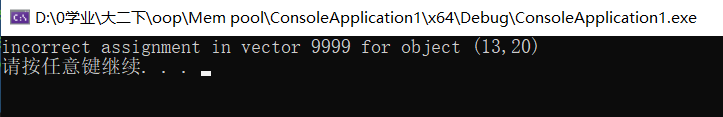
**Chapter 3: Testing Results**

（1）测试一：直接用pta的代码，将std::allocator替换为我的allocator，验证结果的正确性：（为方便测试，我将pta和钉钉群里的代码拷贝在同一个main.cpp文件，并定义宏#define TEST\_TYPE 0，当这个数值设置为1时，对应pta的测试程序，当设置为0时，对应课程钉钉群上传的测试程序）



使用std::allocator和我的allocator结果一致

（2）测试二：运行钉钉群中的测试：



和std::allocator的结果一致

（3）测试和std::allocator相比的时间优化程度(resize 因素)：

我们定义resize的次数是处理规模n，而运行时resize的范围由

函数生成，所以改变生成的规模范围，我们可以发现：

a)当resize的数值范围在1~8时：

b)当resize的数值范围在1~32时：

c)当resize的数值范围在1~64时：

d)当resize的数值范围在1~128时：

e)当resize的数值范围在1~512时：

f)当resize的数值范围在1~1024时：

g)当resize的数值范围在1~2000时：

结论：在面对大量小内存对象时，我们的myallocator有着更强大且优越的性能，这是因为每次申请内存块都申请了CHUNK\_NUM块，因此当再一次插入相同大小区间的对象时，可以直接用内存池取出内存地址直接插入从而加速，但当面对大量分布均匀但范围很广的对象时，我们的结果会慢于标准模板库，这是因为当均匀分布时，每次插一次都有可能引发连续分CHUNK\_NUM块内存的动作，而分配CHUNK\_NUM块时串联freelist需要花费额外的时间。

（4）测试和std::allocator相比的时间优化程度(CHUNK\_NUM 因素)：

a)当resize范围是0~10000时，chunk=8表现更出色，而chunk=16最慢，这是因为resize范围太广，导致再次落入0~1024的某个特定块的概率变小，所以chunk太大会造成大量的建立freelist时的时间冗余，且会浪费大量空间

b）当resize范围是0~32时

此时chunk=4表现最慢，这是因为这个时候，大量落在相同区块的重复会让增加chunk的数量更占时间优势

**Chapter 4:评价与反思**

总结来看，我们的myallocator在大量的小对象内存分配时，展现出较大优势，但当resize范围扩大时，甚至时间会比std::allocator更慢，此外myallocator由于每次都申请CHUNK\_NUM个内存块来为下一次相同大小的内存分配加速，所以会产生一定的内存消耗，并且每次都将对象放在不小于它的内存块中，导致内存会有浪费，所以我们是用空间来换取时间的优化。

**Appendix:source code**

#pragma once

#include<iostream>

#include<cassert>

#define DEBUG\_STATE 0

#if DEBUG\_STATE

int count1[1024/8][2] = {0};

#endif

namespace Alloc {

const size\_t SEG\_SIZE = 8;//step between different block in memory

const size\_t MAX\_SIZE = 1024;//maxsize for memory block

const size\_t CHUNK\_NUM = 4;//memory blocks' number of every allocating

const size\_t FREELIST\_SIZE = MAX\_SIZE / SEG\_SIZE;//freelist's size

#define INDEX\_SEG(e) (((((e)+SEG\_SIZE-1)&~(SEG\_SIZE-1))>>3)-1)//index calculator

struct Freelist\_node {//freelist's node,store address mainly

struct Freelist\_node\* next;

};

template <class \_Ty>

class MyAllocator

{

public:

/\*---value definition and typedef here---------\*/

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 size\_t size\_type;

static inline struct Freelist\_node\* free\_list[FREELIST\_SIZE] = { nullptr };//freelist

template<class T>

struct rebind { typedef MyAllocator<T> other; };

/\*-----------function is defined here ------------\*/

MyAllocator() {}//constructor

template<class T>

MyAllocator(const MyAllocator<T>& a) {}//constructor

~MyAllocator() {}//destructor

inline size\_type max\_size() {//retunrn maxsize of a valuetype

return size\_type(UINT\_MAX / sizeof(value\_type));

}

inline pointer address(reference \_Val) {//call std::addressof to get address of val

return std::addressof(\_Val);

}

inline const\_pointer address(const\_reference \_Val) {//call std::addressof to get address of val

return std::addressof(\_Val);

}

static inline void deallocate(pointer ptr, size\_type count) {//deallocate function

size\_t size\_sum = count \* sizeof(value\_type);//calculate size of block we will deallocate

if (size\_sum > MAX\_SIZE) {//if size is bigger than maxsize, free directly

free(ptr);

return;

}

else {//put freelist's head to this block

((Freelist\_node\*)ptr)->next = free\_list[INDEX\_SEG(size\_sum)];

free\_list[INDEX\_SEG(size\_sum)] = (Freelist\_node\*)ptr;

}

}

static pointer allocate(size\_type count) {//allocate function

size\_t size\_sum = count \* sizeof(value\_type);//calculate size of block

if (size\_sum > MAX\_SIZE) {//if size is bigger than maxsize, malloc directly

char \*buf = (char\*)std::malloc(size\_sum);//use a buffer to store block

if (!buf) {//allocate fail

std::cout << "memory overflow" << std::endl;

throw "error";//throw exception

}

return (pointer)buf;//return address of the buf

}

Freelist\_node\* cur;

Freelist\_node\* des;

des = free\_list[INDEX\_SEG(size\_sum)];

if (des == nullptr) {

#if DEBUG\_STATE

count1[INDEX\_SEG(size\_sum)][0]++;//use in debug,record freelist's message

count1[INDEX\_SEG(size\_sum)][1]=0;

std::cout << "申请内存空间的大小:" << size\_sum << " freelist 下标:" << INDEX\_SEG(size\_sum) << std::endl;

#endif

return (pointer)CallForMem(size\_sum, INDEX\_SEG(size\_sum));//freelist is null, call for memory

}

#if DEBUG\_STATE

count1[INDEX\_SEG(size\_sum)][1]++;

#endif

free\_list[INDEX\_SEG(size\_sum)] = free\_list[INDEX\_SEG(size\_sum)]->next;//update freelist

return (pointer)des;

}

template<class Ut>

static inline void destroy(Ut\* p) {//destory, use destructor function directly

p->~Ut();

}

template<class Ut, class Pt>

static inline void construct(Ut\* p, Pt argv) {//constructor , use constructor function directly

new(p) Ut(argv);

}

private:

inline static char\* CallForMem(size\_t size\_to\_call, int index) {//call for memory by this function

Freelist\_node\* temp = nullptr;

size\_t all = 7;

size\_t align\_to\_eight = ((size\_to\_call + 7) & ~all);//find the nearest number of times of 8

char\* buf = (char\*)std::malloc(CHUNK\_NUM \* align\_to\_eight);//allocate

if (!buf) {

std::cout << "out of memory" << std::endl;//allocate fail

throw "error";

}

char\* record = buf + align\_to\_eight;

for (int i = 0; i < CHUNK\_NUM - 1; i++) {//link node one by one

if (i == 0) {

free\_list[index] = (Freelist\_node\*)record;

temp = (Freelist\_node\*)record;

temp->next = nullptr;

record += align\_to\_eight;

}

else {

temp->next = (Freelist\_node\*)record;

temp = temp->next;

temp->next = nullptr;

record += align\_to\_eight;

}

}

return buf;

}

};

}

**Appendix2:test code**

#include <iostream>

#include <random>

#include <vector>

#include<Windows.h>

#include<iomanip>

#include<list>

#include "MyAllocator.h"

template<class T>

using MyAllocator = Alloc::MyAllocator<T>; // replace the std::allocator with your allocator

template<class T>

using IniAllocator = std::allocator<T>;

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

using namespace std;

const int TestSize = 10000;

const int PickSize = 1000;

const int test\_size = 10;

double time\_counter[2][10][test\_size];

double time\_counter\_list[2][10][test\_size];

int main()

{

std::random\_device rd;

std::mt19937 gen(rd());

//std::uniform\_int\_distribution<> dis(1, TestSize);

int distr[10] = { 8,32,64,128,256,512,1024,2000,5000,10000 };

LARGE\_INTEGER t1, t2, t3,t4,tc;//high precision clock

double time\_t1=0,time\_t2=0;

QueryPerformanceFrequency(&tc);//record frequency of the clock in cpu

/\*test for vector\*/

using IntVec = std::vector<int, MyAllocator<int>>;

using IntVec\_t = std::vector<int>;

std::vector<IntVec, MyAllocator<IntVec>> vecints(TestSize);

std::vector<IntVec\_t> vecints\_t(TestSize);

for (int j = 0; j < 10; j++) {

//实验组，使用自定义内存分配器

std::uniform\_int\_distribution<> dis(1, distr[j]);

time\_t1 = 0;

time\_t2 = 0;

for (int i = 0; i < TestSize; i++) {

size\_t size = dis(gen);

QueryPerformanceCounter(&t1);

vecints[i].resize(size);

QueryPerformanceCounter(&t2);

time\_t1 += (double)((t2.QuadPart - t1.QuadPart) \* 1000.0 / tc.QuadPart);

if (i % 1000 == 0) {

time\_counter[0][j][i / 1000] = time\_t1;

}

}

//对照组，使用std::allocator

for (int i = 0; i < TestSize; i++) {

size\_t size = dis(gen);

QueryPerformanceCounter(&t3);

vecints\_t[i].resize(size);

QueryPerformanceCounter(&t4);

time\_t2 += (double)((t4.QuadPart - t3.QuadPart) \* 1000.0 / tc.QuadPart);

if (i % 1000 == 0) {

time\_counter[1][j][i / 1000] = time\_t2;

}

}

}

cout << "vector test result" << endl;

for (int j = 0; j < 10; j++) {

cout << "when resize is between 0 and " << distr[j] << endl<<endl;

cout << setw(15) << setiosflags(ios::left) << "input scale:";

for (int i = 0; i < 10000; i += 1000)cout << setw(10) << i;

cout << endl;

cout << setw(15) << setiosflags(ios::left) << "myallocator:";

for (int i = 0; i < test\_size; i++) {

cout << setw(10) << time\_counter[0][j][i] ;

}

cout << endl;

cout << setw(15) << setiosflags(ios::left) << "std::allocator:";

for (int i = 0; i < test\_size; i++) {

cout << setw(10) << time\_counter[1][j][i] ;

}

cout << endl;

}

}