Project Report

An Allocator with Memory Pool

3180105481 王泳淇

3180105150 朱雨轩

June 30, 2020

Contents

1 Introduction

An allocator is a tool for dynamic memory allocation used by some STL containers in C++. It can handle all the requests for allocation and deallocation of memory for a given container. Although the C++ Standard Library provides general-purpose allocators that are used by default, custom allocators may also be supplied by the programmer to fit their own requirements.

Basically, the allocation and deallocation can be done by the memory allocating (and deallocating) functions provided by C++, like `malloc` and `free`, or `new` and `delete`. However, these functions may cause fragmentization of the memory. For example, `glibc` maintains a linked list of memory chunks where the lengths of chunks are variable. Every time `malloc` is called, a small chunk of memory will be cut off from a larger chunk and offered to the program. Yet when it is freed, it might not be merged into a larger chunk. So frequent allocations and deallocations will create many small memory fragments, which not only reduces the efficency, but also causes that no spare memory can be offerd when a large chunk of memory is required by the program since the memories are all in small pieces.

A typical design of allocator to avoid this problem is called memory pool. In the following chapters, we will give some details on this design, and show our allocator based on this method.

2 Structural Design

2.1 Design of Allocator Class

We design the allocator class following the class template `std::allocator`. As this template changes with the standard of C++, we basically following the standard of C++ 11, but may contains a little difference. The allocator is designed as a template class with typename `T`. There are some member types defined in the class as follow:

```cpp
typedef void _Not_user_specialized;
typedef T value_type;
typedef T* pointer;
typedef const T* const_pointer;
typedef T& reference;
typedef const T& const_reference;
typedef 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 <class U>
struct rebind {
    typedef Allocator<U> other;
};
```

The interface of the member functions and there usage are as follow:

```cpp
// creates a new allocator instance
Allocator() noexcept;
Allocator(const Allocator& other) noexcept;
template <class U> Allocator(const Allocator<U>& other) noexcept;

// destructs an allocator instance
~Allocator();

// returns the address of a certain element;
pointer address(reference x) const noexcept;
const_pointer address(const_reference x) const noexcept;

// allocate memory with size of n * sizeof(T), which is uninitialized
pointer allocate(size_type n, const void* hint = 0);

// deallocate the memory allocated before pointed by p
void deallocate(pointer p, size_type n)
```

```

// deallocate the memory in use and allocate a new chunk of memory,
    not necessary
pointer reallocate(pointer p, size_type originalSize, size_type
    newSize)

// return the max memory size available
size_type max_size() const noexcept;

// constructs an object in allocated storage
void construct(pointer p, const_reference val);

// destructs an object in allocated storage
void destroy(pointer p);
```

In the following part we will focus on how to realize the `allocate` and `deallocate` with a memory pool.

2.2 Memory Pool

The design of our memory pool is based on SGI STL. The principles can be illustrated with the following diagram.

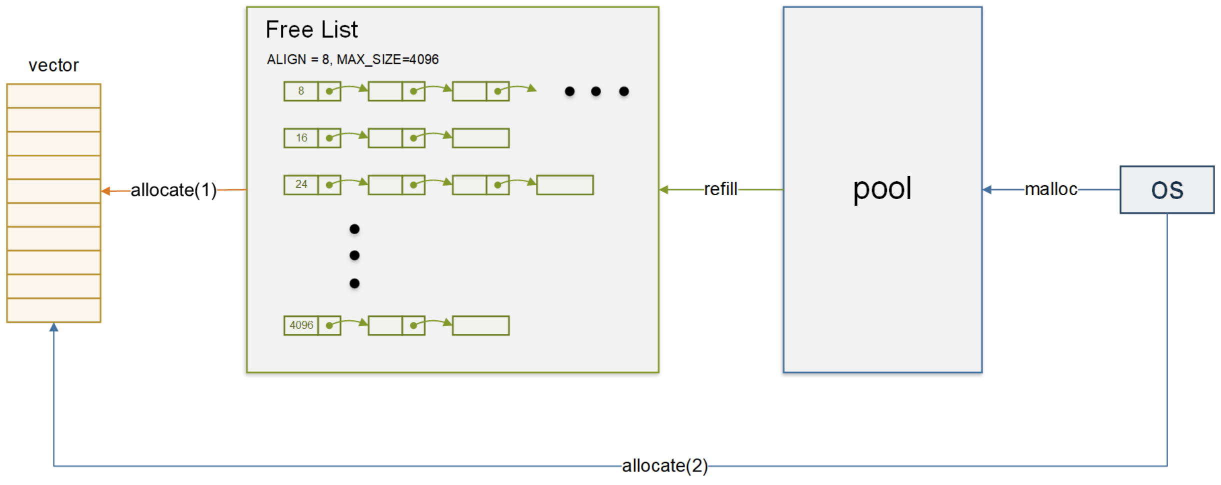


Figure 2-1 Princeple of the Memory Pool

As shown in the diagram, a set of free lists and a pool is maintained in the memory pool. The pool contains a large chunk of memory with sequential address. Each list in the free list set consists of a series of nodes, with each node pointing a piece of memory with certain size. The lists are aligned in order of piece size.

When the container calls `allocate`, the allocator will check the size of memory required. If it is larger than the max size the free lists can provide, it will call `malloc` directly to get memory (path allocate(2)). Otherwise, it will find the list containing nodes with smallest size to satisfy the requirement (path allocate(1)). For example, in the diagram above, if a 17-byte piece of memory is needed, the free list will choose the 24-byte list, allocate the memory pointed by the first node to the container, and remove the first node from the list. The prodecure of calculating fitting size is called `ROUND_UP`.

If the list of required size runs out, the pool will distribute some memory in pool to refill the list by adding a certain number of new nodes. If the pool is out of memory, it will call `malloc` to replenish the pool. Besides, if `malloc` cannot provides more memory, this structure will find some memory in lists of different size as substitution.

On the other hand, when `deallocate` is called, the allocator will judge where this part of memory comes from. If it's larger than the maximum size, it must has been allocated by `malloc`, and the allocator will directly free it. Otherwise this piece of memory will be added to the free list again. This step is done by creating a new node in the corresponding free list and let it point to the piece of memory deallocated.

By using this strategy of memory allocating, the times `malloc` and `free` are called are reduced. The time cost of allocating memory from the pool is much less than that by using `malloc` since `malloc` needs to do registration and some other work. As a result, using memory pool will bring higher efficency. Besides, as illustrated before, using memory pool can alleviate the fragmentization of the memory.

2.3 Realization of the Memory Pool

The memory stored in the pool is denoted by two pointers pointing to the start and end points of the memory chunk and a variable storing the size of it.

```
static char* start_free;
static char* end_free;
static size_t heap_size;
```

The node of the free lists are realized with a union containing a pointer pointing to the next node and another pointer pointing to the free space. By using union, the two pointers can share the memory and reduce the space cost. The free lists are realized with an array of pointers pointing to the node lists.

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

static node* free_list[NFREELEISTS];
```

These varialbles are declared to be static, which guaranteed that many allocator shares one memory pool.

The `allocate` function is designed as follow. When

```
// allocate memory with size of n * sizeof(T), which is uninitialized
pointer allocate(size_type n, const void* hint = 0) {
    size_type required = n * sizeof(T);

    // use malloc directly for large piece of memory
    if (required > MAX_BYTES) {
        pointer p = (pointer)malloc(required);
        return p;
    }

    node* volatile* my_free_list;
    node* result;
    // find the list of fitable size
    my_free_list = this->free_list + freelist_index(required);
    // get the first node from the free list
    result = *my_free_list;
    // free list is out of memory, use memory in the pool to do refilling
    if (result == 0) {
         void* r = refill(ROUND_UP(required));
         return (pointer)r;
    }
    *my_free_list = result->free_list_link;
    return (pointer)result;
}
```

The allocator will judge which way of allocation to use according to the size of the memory. When there's no enough memory in the free lists, `refill` is called to refill the corresponding free list. The `refill` function is designded as follow:

```
// distribute memory from the pool to the free lists
static void* refill(size_t n) {
    // nobjs is the default number of newly refilled node
    // chunk_alloc will reduce this number if there's not enough memory in the pool
    int nobjs = 50;

    // call chunk_alloc to get memory from the pool
    char* chunk = chunk_alloc(n, nobjs);
    node* volatile* my_free_list;
    node* result;
    node* current_node, * next_node;
    int i;
    // the pool can only provide memory for 1 node, then this part of memory will be used for allocation
    if (nobjs == 1) return chunk;
    my_free_list = free_list + freelist_index(n);
    result = (node*)chunk;
    *my_free_list = next_node = (node*)(chunk + n);
    // adhere the nodes to the free lists
    for (i = 1; ; i++) {
        current_node = next_node;
        next_node = (node*)((char*)next_node + n);
        if (nobjs - 1 == i) {
            current_node->free_list_link = 0;
            break;
        }
        else {
            current_node->free_list_link = next_node;
        }
    }
    return (result);
}
```

The refill function call **`chunk_alloc`** to get memory from the pool, divide them into small pieces of corresponding size and adhere them to the list. The number of node is set to a defalut value by **`refill`**. If there's not so much memory in the pool, **`chunk_alloc`** will reduce the number of nodes. The **`chunk_alloc`** function is designed as follow:

```
static char* chunk_alloc(size_t size, int& nobjs) {
  char* result;
  size_t total_bytes = size * nobjs;
  size_t bytes_left = end_free - start_free;

  // there is enough memory for refilling and do allocation directly
  if (bytes_left >= total_bytes) {
      result = start_free;
      start_free += total_bytes;
      return result;
  }
  // not enough for refilling but still enough for allocation
  // provide memory for allocation and use the rest to refill the free list
  else if (bytes_left >= size) {
      nobjs = bytes_left / size;
      total_bytes = size * nobjs;
      result = start_free;
      start_free += total_bytes;
      return result;
  }
  // the pool is out of memory, call malloc to add new memory to the pool
  else {
      size_t bytes_to_get = 80 * total_bytes + ROUND_UP(heap_size >> 4);
      if (bytes_left > 0) {
          node* volatile* my_free_list = free_list + freelist_index(bytes_left);
          ((node*)start_free)->free_list_link = *my_free_list;
          *my_free_list = (node*)start_free;
      }
      start_free = (char*)malloc(bytes_to_get);

      // system is out of memory and malloc doesn't work
      // try to find memory in free lists of other sizes
      if (start_free == 0) {
          size_t i;
  node* volatile* my_free_list;
  node* temp;
  for (i = size; i < MAX_BYTES; i += ALIGN) {
    my_free_list = free_list + freelist_index(i);
    temp = *my_free_list;
    if (NULL != temp) {
```

```
            *my_free_list = temp->free_list_link;
            start_free = (char*)temp;
            end_free = start_free + i;
            return chunk_alloc(size, nobjs);

        }
    }
                // no more memory anywhere
    end_free = NULL;
            }
            heap_size += bytes_to_get;
            end_free = start_free + bytes_to_get;
            return (chunk_alloc(size, nobjs));
    }
}
```

The `chunk_alloc` function find memory in 3 ways:

- Memory in pool now
- Memory get by `malloc`
- Memory in other free lists

The 3 places are checked in sequence. If none of them can provide enough memory, the function returns NULL.

The `deallocate` function moves back the memory to where it comes from. It frees large chunk of memory directly and add the small pieces back to the free lists. The code is as follow:

```
void deallocate(pointer p, size_type n) {

    size_t required = n * sizeof(T);
    // free large chunk of memory
    if (required > (size_t)MAX_BYTES) {
        free(p);
        return;
    }
    node* q = (node*)p;
    node* volatile* my_free_list;
    // move the piece of memory back to the free list
    my_free_list = free_list + freelist_index(required);
    q->free_list_link = *my_free_list;
    *my_free_list = q;
}
```

There are also some indispensable parts of allocator, yet they have little correlation with the memory pool, so we don't post the code here. You can refer to the source code directly.

3 Testing Result and Analysis

The test is done on the MacOS Catalina Version 10.15.5 with processor of 2.4 GHz Quad-Core Intel Core i5 and memory 8 GB 2133 MHz LPDDR3. The translater is clang++ version 11.0.3 .All the original data are generated in the /Data file and all the graphs are in the /test/graph file.

3.1 Extreme Case Test

The extreme case is a test example (TestExtreme) created specifically for the MemoryPool allocator mechanism.

By reasonably arranging the order and size of resize, a large number of memory blocks are recovered in time in the MemoryPool. Each time memory needs to be allocated, **the memory can be directly called from the free_list without malloc**. This arrangement can reflect the characteristics of the memory pool to the greatest extent.

3.1.1 Experimental Conditions

- Int1

  - When creating a vector ,it is equally divided into two blocks, one block is larger, accounting for 128 int, and the other block only occupies 32 int. These two numbers are chosen because the 4 byte int puts 128 small block upper bounds that can be lower than memorypool management memory (here set to 1024 bytes). When resize, it can be handled by memorypool through the free_list mechanism.
  - When resizing, first reduce the larger vector to 64 ints, to **ensure that the free_list of memorypool recovers the free blocks of sufficient size and number**.

    Then enlarge the smaller vector to 64 ints. At this time, the **int block that has just been recovered happens to come in handy**. memorypool directly reallocates the recovered memory to the resize vector, and there is **no repeated operation of malloc and free**, thus ensuring that the efficiency is greatly improved.

- Point2D
  - The pair class's member functions are stored distinctly from the objects, so theoretically every Point2D objects occupies the space of two int members.
  - When creating a Point2D ,it is equally divided into two blocks, one block is larger, accounting for 64 Point2D , and the other block only occupies 16 Point2D.
  - When resizing , first reduce the larger vector to 32 Point2D, to **ensure that the free_list of memorypool recovers the free blocks of sufficient size and number**.

    Then enlarge the smaller vector to 32 Point2D

- Int2
  - At this time, we set the smallest block size in Memory Pool to 8 bytes, the Max_Bytes to 4096 to enlarge the memory pool size.
  - When creating a vector, it is equally divided into two blocks, one block is larger, accounting for 1024 int, and the other block only occupies 256 int.

  - When reisizing ,first reduce the larger list to 512 int, to **ensure that the free__list of memorypool recovers the free blocks of sufficient size and number**. Then enlarge the smaller list to 512 int.

- Int3
  - Under the condition of Int2, we discover that the difference between malloc and memory__pool is not distinct. Even we find that malloc is faster than memory__pool
  - So we use the size in Int1, When creating a vector ,it is equally divided into two blocks, one block is larger, accounting for 128 int, and the other block only occupies 32 int. When resizing, first reduce the larger vector to 64 ints, to **ensure that the free__list of memorypool recovers the free blocks of sufficient size and number**.

3.1.2 Results

	Std::allocator	**Malloc__alloc**	**MemoryPool__alloc**
Resize, int1	0.002894	0.002822	0.001427
Resize, Point2D	0.002911	0.002853	0.001739
Resize, int2	0.016888	0.008885	0.009843
Resize, int3	0.004952	0.004718	0.002473

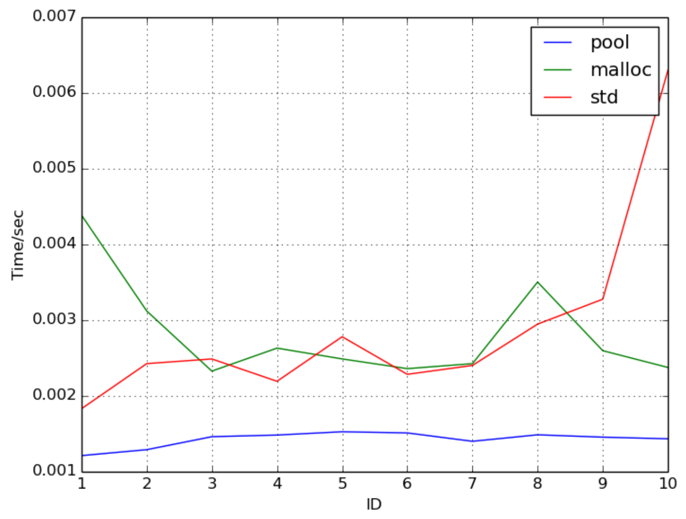


Figure 3.1.2.1 Resize__Int1

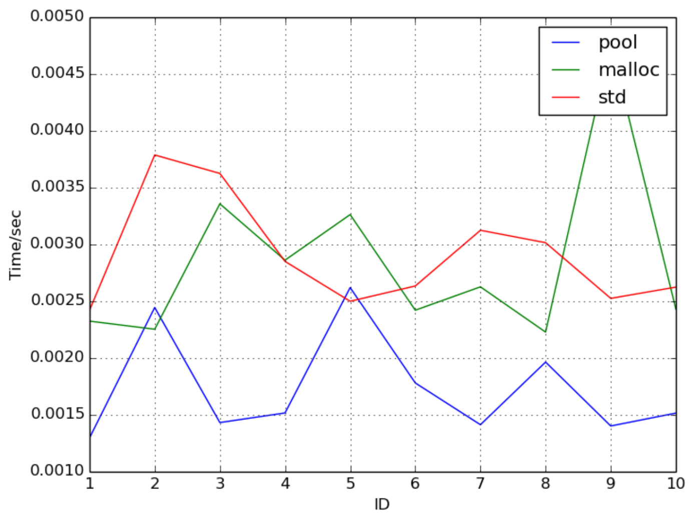


Figure 3.1.2.2 Resize_Point2D

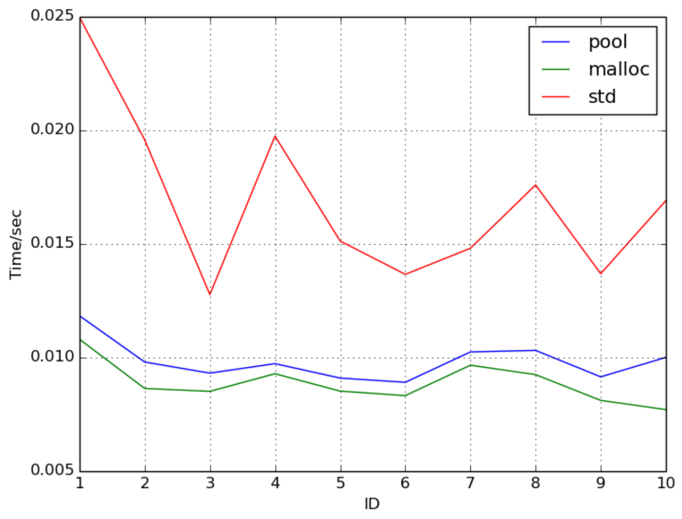


Figure 3.1.2.3 Resize_Int2

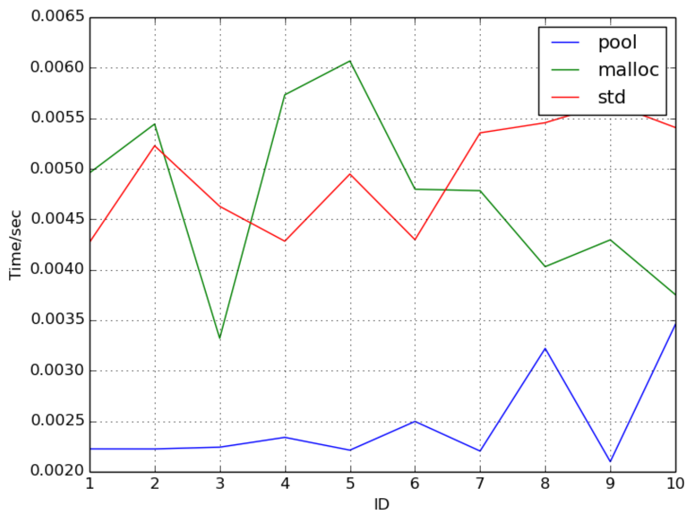


Figure 3.1.2.4 Resize_Int3

3.1.3 Analysis

From the test_graph above, we find that

- In Case1, the performance of memorypool is overwhelming due to its tailor-made characteristics. There is no repeated malloc() and free().

- Comparing Case2 and Case1, we find that the performance of MemoryPool is still better in average. However, the advantage is reducing.

  We guess the reason is that Point2D object is not exactly the size of 2 ints, so when resizing the vector may exceeds the upper bound of MemoryPool Block size. Under this circumstance, it needs to malloc() instead of using pool, which needs more time.

- In Case3, we enlarge the max size of MemoryPool Block and shrink the gap between blocks , so the List we maintain in our pool is getting longer. When we still use the MemoryPool management upper bound in testing, we find that the time consuming of MemoryPool and Malloc is nearly the same. Malloc is even slightly faster.

  We guess the reason is that when the size of vector getting larger, the memory we need for resizing each vector is larger. Pool needs more time for List operations, so it is slightly slower than Malloc.

- With the analysis of Case3, we reduce the reszing size of vector to the same as Case 1 to form Case 4. We can tell the advantage of using MemoryPool is obvious again, and the difference is getting larger than Case 1.

  The reason is that the smaller memory framentation can be used efficiently.

Generally speaking, we find that the MemoryPool is doing a great job in resizing small size of memory, and the performance will get better if the Pool block is finely cut into small pieces.

However, the best arrangement of Block size and Max size is different according to actual situation and still needs to be explored.

3.2 Given Test

The test given by PTA (TestGiven.cpp) is a case with resize size radomly given in the range of TestSize(1e4).

3.2.1 Experimental conditions

- Vector
  - Create: Create 1e4 vectors. The range of allocation is between [1, 1e4].
  - Resize: Randomly pick 1e3 vectors and each vector resizes in the range [1, 1e4].
- List
  - Create: Create 1e4 vectors. The range of allocation is between [1, 1e4].
  - Resize: Randomly pick 1e4 vectors and each vector resizes in the range [1, 1e4].

3.2.2 Results

Std::allocator	**Malloc_alloc**	**MemoryPool_alloc**	**MemoryPool_alloc**
Create, int	0.0925	0.092	0.0911
Create, Point2D	0.1751	0.1745	0.1805
Resize, int	0.0511	0.0513	0.0489
Resize, Point2D	0.0998	0.0928	0.0961

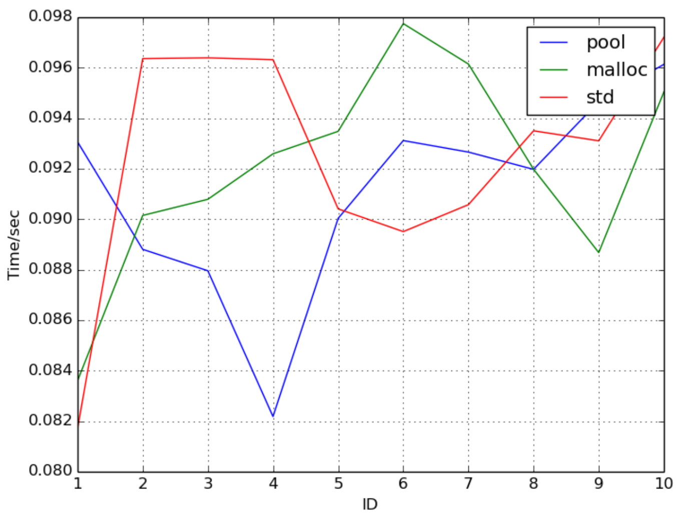


Figure 3.2.2.1 Create_int

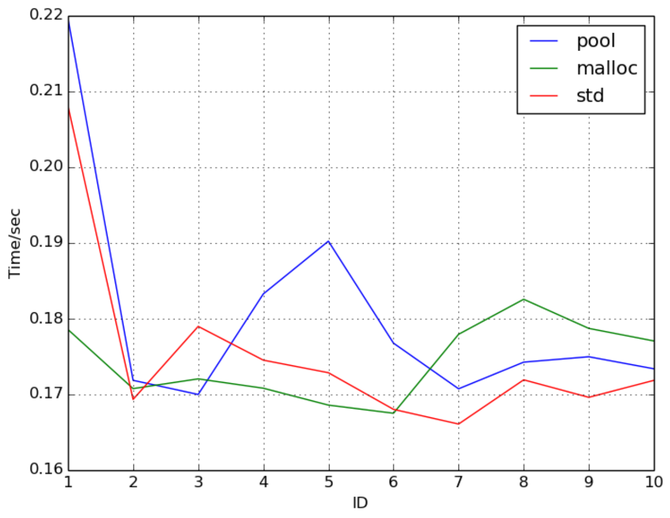


Figure 3.2.2.2 Create_Point2D

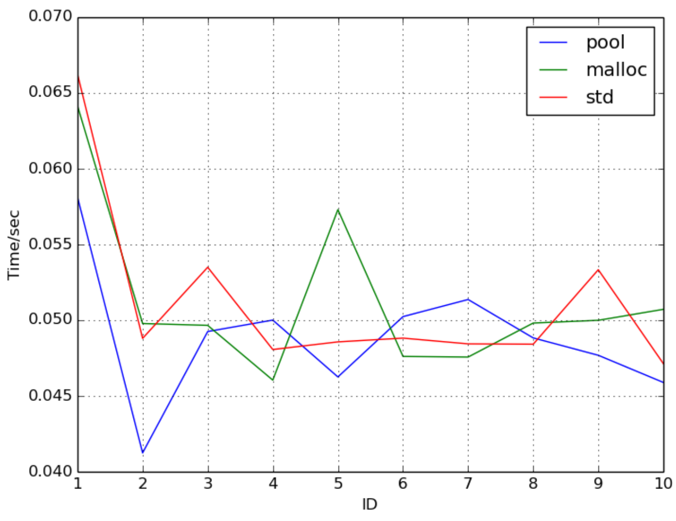


Figure 3.2.2.3 Resize_int

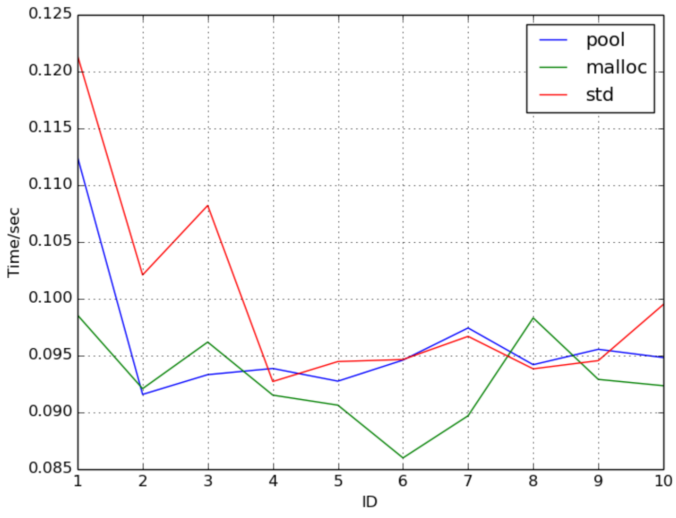


Figure 3.2.2.4 Risize_Point2D

3.2.3 Analysis

From out test_graph above, we find that

- It's hard to tell which allocator does the best job in create int or create Point2D.
  - MemoryPool's performance is similar to that of Malloc, which implies that initially, MemoryPool itself has few free blocks in free_list, and most of them are allocated directly through malloc.

  - Std's performance is not distinctly great, we guess the reason is that the new() and other operations consume a lot of time in C++ compared with malloc() operation in C.

- When resizing ,the performance of MemoryPool is slightly better than the other two, but the advantage is not obvious.
  - When resizing, the size of the vector becomes larger and smaller randomly, the overall total memory basically remains the same, which is more in line with the efficient scenario of MemoryPool. MemoryPool omits the work of free() and malloc() to the greatest extent, which improves efficiency and can effectively reduce the generation of memory fragments.
  - Std's performance is not distinctly great, we guess the reason is that the new() and other operations consume a lot of time in C++ compared with malloc() operation in C.
  - However, because of the radomly setting size, the MemoryPool may need to malloc() most of the time without using pool. So the difference is not very obvious.

3.3 Usability Test

Test if this allocator is usable in other containers.

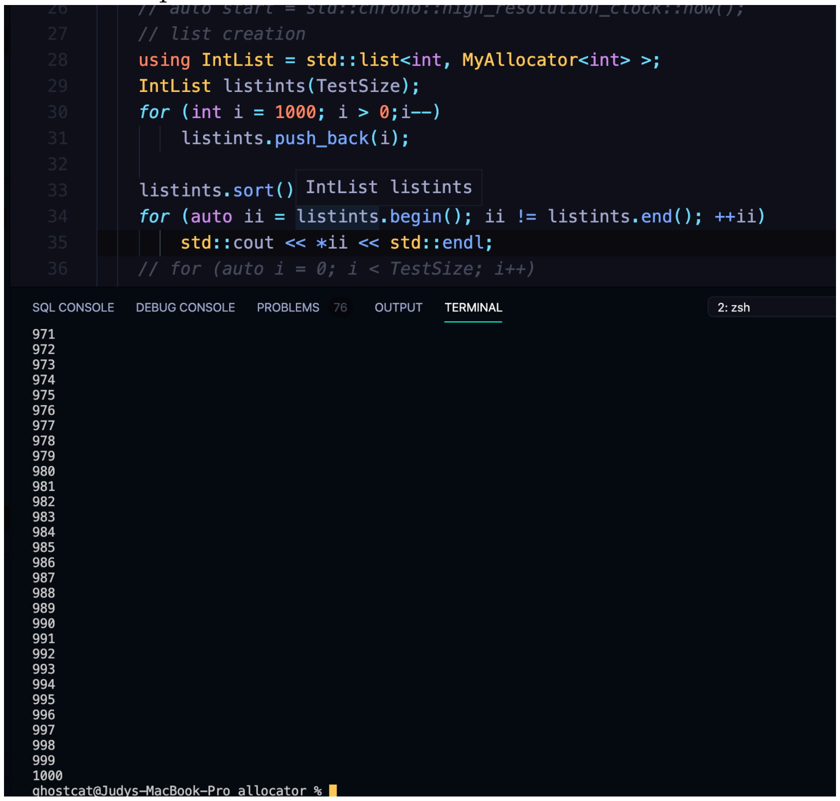
3.3.1 Experimental conditions

- List
  - Create list with size 1e3
  - Push_back ,sort operation test
- Deque
  - Create Deque with size 1e4
  - Push_front , front operation test

- Set
  - Create Set
  - Insert operation test
- Stack
  - Create stack with Deque using our allocator
  - Push, pop operation test

3.3.2 Result

- test on list:pass

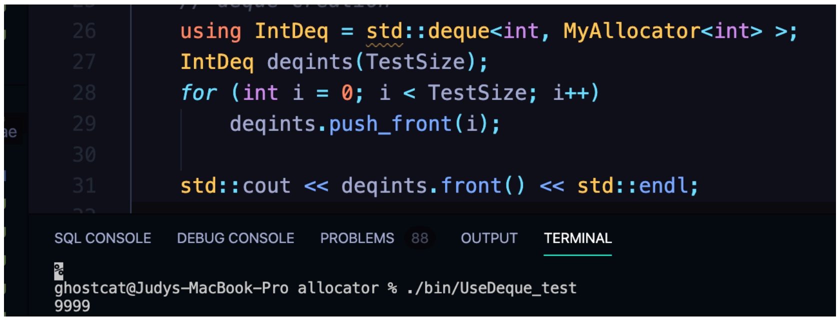


```
// auto start = std::chrono::high_resolution_clock::now();
// list creation
using IntList = std::list<int, MyAllocator<int> >;
IntList listints(TestSize);
for (int i = 1000; i > 0;i--)
    listints.push_back(i);

listints.sort()
for (auto ii = listints.begin(); ii != listints.end(); ++ii)
    std::cout << *ii << std::endl;
// for (auto i = 0; i < TestSize; i++)
```

```
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
ghostcat@Judys-MacBook-Pro allocator %
```

- test on deque:pass

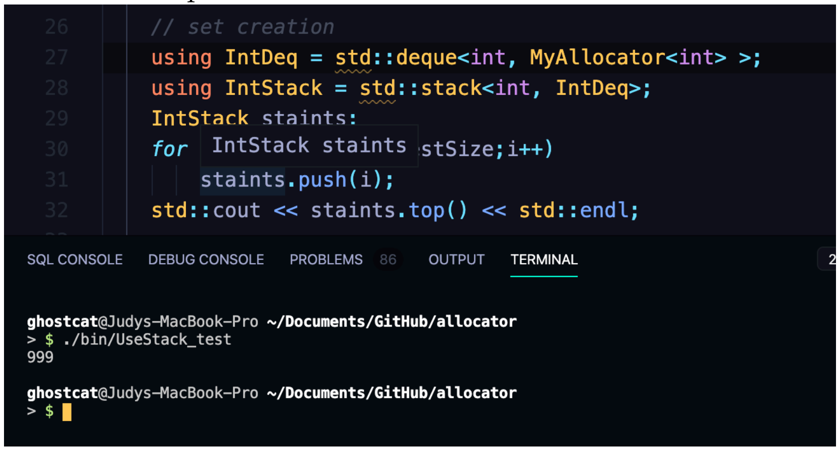


```
using IntDeq = std::deque<int, MyAllocator<int> >;
IntDeq deqints(TestSize);
for (int i = 0; i < TestSize; i++)
    deqints.push_front(i);

std::cout << deqints.front() << std::endl;
```

```
ghostcat@Judys-MacBook-Pro allocator % ./bin/UseDeque_test
9999
```

- test on stack:pass

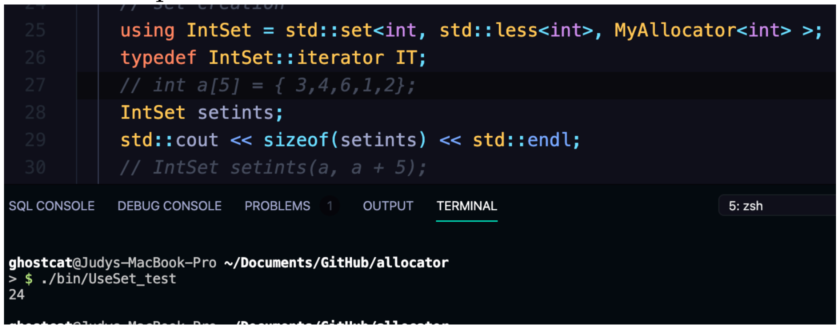


```
// set creation
using IntDeq = std::deque<int, MyAllocator<int> >;
using IntStack = std::stack<int, IntDeq>;
IntStack staints;
for  IntStack staints stSize;i++)
    staints.push(i);
std::cout << staints.top() << std::endl;
```

```
ghostcat@Judys-MacBook-Pro ~/Documents/GitHub/allocator
> $ ./bin/UseStack_test
999

ghostcat@Judys-MacBook-Pro ~/Documents/GitHub/allocator
> $
```

- test on set:pass



```
using IntSet = std::set<int, std::less<int>, MyAllocator<int> >;
typedef IntSet::iterator IT;
// int a[5] = { 3,4,6,1,2};
IntSet setints;
std::cout << sizeof(setints) << std::endl;
// IntSet setints(a, a + 5);
```

```
ghostcat@Judys-MacBook-Pro ~/Documents/GitHub/allocator
> $ ./bin/UseSet_test
24
```

3.3.3 Analysis

Our allocator is widely used in all kinds of container including associated container, sequential container and container adapter.

4 Reference

[1] Cppreference: std::allocator https://en.cppreference.com/w/cpp/memory/allocator

[2] 侯捷. STL 源码剖析. 华中科技大学出版社