实验一：Bigsize

vector create 过程，创建 vector 数量 N=1e4，每一 vector 分配的范围在[1，1e4]之间。

vector resize 过程，每一 vector 进行 resize 的大小范围在[1, 1e4]之间。

Allocator 类型 / 测试项目	Std::allocator	Malloc_alloc	MemoryPool_alloc
create, int	0.169356	0.544501	0.527737
create, Myclass	0.327492	0.527253	0.039883
resize, int	0.007866	0.071805	0
resize, Myclass	0.064506	0.082872	0.011512

Std::allocator 必然是经过更加细致的优化的。在 create 的过程它的表现十分突出。MemoryPool 的表现和 Malloc 接近，因为 MemoryPool 本身初始的时候 free_list 中的可用区块很少，大多还是直接通过 malloc 分配出来，所以效率相差不大。

但是 resize 的时候 MemoryPool 的优势就开始体现出来。因为 resize 的时候 vector 长度有变大有变小，整体总内存占用基本保持不变，比较符合 memorypool 的高效场景。Memorypool 因为管理内存的机制不同，最大程度地省略了先 free()再 malloc()的重复工作，提高效率的同时也能有效减少内存碎片的产生，可以看到 MemoryPool 在 resize 的时候表现是十分不错的

实验二：Smallsize

vector create 过程，创建 vector 数量 N=1e4，每一 vector 分配的范围在[1，1e4]之间。

vector resize 过程，每一 vector 进行 resize 的大小范围在[1, 100]之间。

Allocator 类型 / 测试项目	Std::allocator	Malloc_alloc	MemoryPool_alloc
resize, int	0	0.030674	0.028324

resize, Myclass	0.019846	0.037222	0.03725

这些时候反而是 std::allocator 的表现更好。猜测 std::allocator 中也有内存池协助管理内存，只是支持的最大区块比较小，所以在小的 resize 的时候有比较好的效果。Malloc 的效率也显著提高了，所幸 memorypool 还是表现稍微好一点点，费力的优化没有白做。

实验三：extreme scenario

Allocator 类型 / 测试项目	Std::allocator	Malloc_alloc	MemoryPool_alloc
resize, int	0.034345	0.182184	0.002022

Extreme scenario 是专门为 MemoryPool 的 allocator 机制特别打造的测试样例（TestExtreme.cpp）。通过合理地安排 resize 的顺序和大小，保证 MemoryPool 中及时回收出大量内存区块，每次需要分配内存的时候都可以直接从 free_list 中调出内存而无需 Malloc。可以最大程度地体现出内存池专门优化的特点。

```
    using IntVec = std::vector<int, MyAllocator<int>>;
    std::vector<IntVec, MyAllocator<IntVec>> vecints1(TestSize);
    for (int i = 0; i < TestSize; i++) vecints1[i].resize(128);

    std::vector<IntVec, MyAllocator<IntVec>> vecints2(TestSize);
    for (int i = 0; i < TestSize; i++) vecints2[i].resize(32);


for (int i = 0; i < PickSize; i++) {
        // int idx = dis(gen) - 1;
        int idx=i;
        // int size = dis_small(gen);
        vecints1[idx].resize(64);
        vecints2[idx].resize(64);
    }
```

具体做法是，创建 vector 的时候均等划分成两大块，一块大小大些，占 128 个 int，另一块只占 32 个 int。选取这两个数是因为 4byte 的 int 放 128 个能低于 memorypool 管理内存的小区块上界（在这里设置成 1024byte），resize 的时候

能由 memorypool 通过 free_list 的机制处理。

Resize 时先把较大的 vector 缩小为 64 个 int，保证 memorypool 的 free_list 回收了足够大小和数量的空余区块，再进行把较大的 vector 放大为 64 个 int 的操作。这时刚刚回收的 int 区块正好能派上用场，memorypool 直接把回收的内存重新分配给 resize 的 vector，完全没有进行 malloc 和 free 的重复操作，从而保证效率大大提高。

这个测试案例下由于量身定做，memorypool 的表现是压倒性的。Std::allocator 比直接调用 malloc 的版本运行时间小了 1 个数量级，而 MemoryPool 又比 std::allocator 小了 1 个数量级。从这里可以大致看出 MemoryPool 的运行机制。

实验四：类的构造函数包含多于一个参数

之前的 allocator 中 MyClass 一直都是使用两个参数的构造函数。

原先的写法只是 assignment operator，只用接受一个引用参数，无法测试出 construct 方法的效果

```
// vecmyclass[idx1][idx2] = val;
```

我们改用这个方法进行赋值

```
        /**test whether the allocator supports ctors
         * that has more than 1 argument.
         * if it is replaced by HXZ::allocator,
         * the program cannot pass the compilation.
        */

vecmyclass[idx1].get_allocator().construct(&vecmyclass[idx1][idx2],11,1
5);
```

带有参数包的模板，可以正常编译并执行。

```
/*...Args 是模板参数包, Template parameter pack. */
template <typename T>
template <typename ObjectType,typename... ArgType>
```

```
void Allocator<T>::construct(ObjectType* p,ArgType&& ...args){
    new(p) ObjectType(std::forward<ArgType>(args)...);
    return;
}
```

而简单的使用单一模板的构造函数就无法通过编译了。

```
template <typename T>
template <typename ObjectType, typename ValueType>
void Allocator<T>::construct(ObjectType* p,ValueType value ){
    new(p) ObjectType(value);
}//compile-time error
```