static_vector 实现说明

一、概述

C++ 标准库中提供了 std::vector 和 std::array 两种容器。前者是动态分配内存、支持自动扩充容量的顺序容器，后者是静态分配内存、大小编译时可知的顺序容器。

std::vector 在分配内存时不会预先构造对象（需要区分内存分配和对象构造），因此，std::vector 不能使用 new 来分配内存，因为 new 会自动在分配的空间上构造对象。

这样做会带来怎样的影响呢？std::vector 将能够分配不支持默认构造的对象，例如

```
struct A {
  A() = delete;     // cannot be default constructed
  A(int) { }
};

A a1;               // Error, no default constructor
A a2(3);            // OK
A* p1 = new A;      // Error, no default constructor
A* p2 = new A[3];   // Error, no default constructor
std::vector<A> v1;  // OK
v1.reserve(10);     // OK. Allocate memory but do not construct objects
v1.push_back(a2);   // OK
v1.resize(3);       // Error, no default constructor
```

相反，std::array 则模拟了 C++ 中原始数组的行为：分配（静态）内存时构造所有成员：

```
A arr1[3];              // Error, no default constructor
A arr2[3] = {1, 2, 3};  // OK
std::array<A, 3> a1;    // Error, no default constructor
std::array<A, 3> a2 = {1, 2, 3};  // OK
```

static_vector 也是一种顺序容器，它将结合上述两种容器的特点：是静态分配的，但分配内存时不构造对象：

```
static_vector<A, 3> arr1;   // OK. Allocate memory
                            // but do not construct objects.
arr1.push_back(3);          // OK
```

二、具体成员

static_vector 是类模板，其声明如下：

```
template<typename Ty, std::size_t Length>
class static_vector;
```

上述声明仅为对类名和模板形参的要求，不限定继承关系及实现方式。
static_vector 的内存必须是静态分配的（分配在栈上）。

1. 类型成员

static_vector 有下列公有类型成员：

```
template<typename Ty, std::size_t Length>
class static_vector {
public:
  using value_type             = Ty;
  using size_type              = std::size_t;
  using difference_type        = std::ptrdiff_t;
  using reference              = value_type&;
  using const_reference        = const value_type&;
  using pointer                = value_type*;
  using const_pointer          = const value_type*;
  using iterator               = /* implementation-defined */;
  using const_iterator         = /* implementation-defined */;
  using reverse_iterator       = /* implementation-defined */;
  using const_reverse_iterator = /* implementation-defined */;
};
```

其中，static_vector 的迭代器应满足随机访问迭代器 和连续迭代器的所有要求。

2. 特殊成员函数

```
template<typename Ty, std::size_t Length>
class static_vector {
public:
  constexpr static_vector() noexcept;                                  // (1)
  constexpr static_vector(const static_vector& other);                 // (2)
  constexpr static_vector(static_vector&&) noexcept;                   // (3)
  constexpr static_vector(std::initializer_list<value_type> list);     // (4)
  constexpr explicit static_vector(std::size_t n);                     // (5)
  constexpr static_vector(std::size_t, const_reference v);             // (6)
  template<std::size_t L>
```

```
  constexpr static_vector(const value_type(&)[L]);                    // (7)
  template<std::size_t L>
  constexpr static_vector(const value_type(*)[L]);                    // (7)
  template<typename Iter>
  constexpr static_vector(Iter beg, Iter end);                         // (8)
  constexpr ~static_vector();
  constexpr static_vector& operator=(const static_vector&);            // (9)
  constexpr static_vector& operator=(static_vector&&) noexcept;        // (10)
  constexpr static_vector& operator=(
    std::initializer_list<value_type>);                                // (11)
};
```

（1）默认构造函数，得到已分配好内存的空容器；

（2）复制构造函数；复杂度与 other 的大小成线性；

（3）移动构造函数；复杂度与 other 的大小成线性；

（4）构造拥有 std::initializer_list 所持有内容的容器。若容器中元素数量超过容器能保存的最大数量，则行为未定义（程序不为处理此情况付出开销）；复杂度与 list 的大小成线性；

（5）构造拥有 n 个元素的容器，这些元素被值初始化，元素不被复制；若 n 超过容器大小，则行为未定义；复杂度与 n 的大小成线性；

（6）构造拥有 n 个值为 v 的元素的容器；若 n 超过容器大小，则行为未定义；复杂度与 n 的大小成线性；

（7）从指向数组的指针或到数组的引用初始化，若 L > Length，则程序将编译错误；复杂度与数组大小成线性；

（8）从迭代器范围 [beg, end) 构造，若迭代器范围中的元素数量超过容器大小，则行为未定义。该构造函数仅当 Iter 为输入迭代器时才参与重载决议；复杂度与 beg 和 end 的距离成线性。

（9）复制赋值运算符，逐元素复制赋值；复杂度为线性；

（10）移动赋值运算符，逐元素移动赋值；复杂度为线性；

（11）从 std::initializer_list 复制赋值；复杂度为线性。

3. 访问器

```
template<typename Ty, std::size_t Length>
class static_vector {
public:
  constexpr reference at(size_type n);                    // (1)
  constexpr const_reference at(size_type n) const;        // (1)
```

```
  constexpr reference operator[](size_type n);              // (2)
  constexpr const_reference operator[](size_type n) const;  // (2)
  constexpr reference front();                              // (3)
  constexpr const_reference front() const;                  // (3)
  constexpr reference back();                               // (4)
  constexpr const_reference back() const;                   // (4)
  constexpr pointer data() noexcept;                        // (5)
  constexpr const_pointer data() const noexcept;            // (5)
};
```

（1）返回位于位置 n 处的元素的引用并进行边界检查。若索引 n 超出已有元素的范围，则抛出 std::out_of_range 异常；复杂度为常数；

（2）返回位于位置 n 处的元素的引用，不进行边界检查；复杂度为常数；

（3）返回首元素的引用，若容器为空，行为未定义；复杂度为常数；

（4）返回尾元素的引用，若容器为空，行为未定义；复杂度为常数；

（5）返回指向底层数组首元素的指针；复杂度为常数。

4. 容量

```
template<typename Ty, std::size_t Length>
class static_vector {
public:
  constexpr bool empty() const noexcept;      // (1)
  constexpr bool full() const noexcept;       // (2)
  constexpr size_type size() const noexcept;  // (3)
};
```

（1）检查容器是否无元素，复杂度为常数；

（2）检查容器是否已满，复杂度为常数；

（3）获取容器中的元素数量，复杂度为常数；

5. 修改器

```
template<typename Ty, std::size_t Length>
class static_vector {
public:
  template<class... Args>
  constexpr reference emplace_back(Args&&... args);   // (1)
  constexpr void push_back(const_reference x);        // (2)
```

```
    constexpr void push_back(value_type&& x);           // (3)
    constexpr void pop_back() noexcept;                 // (4)
};
```

（1）使用参数 `args` 原位构造新元素到容器尾部。使用 `std::forward<Args>(args)...` 将实参转发给构造函数。复杂度为常数；

（2）将新元素初始化为 `x` 的副本并插入到容器尾部；

（3）将新元素移动到容器尾部；

（4）移除尾部元素。若容器为空，则行为未定义。

实现提醒

1. 考虑为迭代器实现 Debug 版本和 Release 版本，在 Debug 模式下迭代器能检测失效情况和越界情况；

2. 注意当类型是可平凡复制类型时的特殊成员函数优化；

3. 注意赋值运算符的强异常保证实现；