向量类模板的设计与实现

2023年10月29日

1 项目概述

本项目旨在设计一个通用的向量类模板，命名为 Vector，用于管理动态数组。该向量类支持多种常用操作，如元素访问、插入、删除、大小调整，以及向量间的加法、减法、数乘和内积等运算。该向量类是一个模板类，可以用于存储不同类型的元素。

1.1 项目要求

1. 实现基本的向量操作，包括构造、析构、大小管理、元素访问等功能。
2. 支持向量之间的加法、减法、数乘和内积运算。
3. 提供友元函数来实现向量间的比较操作，如相等、不等、小于、小于等于、大于和大于等于。
4. 使用异常处理保证程序的灵活性。
5. 进行全面的测试以确保向量类的正确性和稳定性。

2 项目设计

2.1 数据成员设计

1. `T* data_`: 这是一个指向动态数组的指针，用于存储向量元素的值。它是向量类的主要数据成员，负责存储实际的数据。在构造函数中分配内存，析构函数中释放内存，以及在许多成员函数中用于访问和修改元素。
2. `size_t size_`: 这是一个表示向量中当前存储的元素数量的数据成员。它用于跟踪向量的大小，并在许多成员函数中用于执行大小管理和索引边界检查。
3. `size_t capacity_`: 这是一个表示动态数组的容量的数据成员。它用于跟踪向量当前的容量，以便在需要时扩展容量。容量是动态数组可以容纳的元素数量的上限。

2.2 成员函数（友元函数）原型设计

2.2.1 构造函数

```
Vector(); // 默认构造函数，创建一个空向量。

Vector(size_t size); // 创建一个指定大小的向量，元素默认初始化。

Vector(size_t size, const T& value); // 创建一个指定大小的向量，元素初始化为指定值。

Vector(std::initializer_list<T> init); // 使用初始化列表创建向量。

Vector(const Vector& other); // 拷贝构造函数。

Vector(Vector&& other); // 移动构造函数。
```

2.2.2 析构函数

```
~Vector(); // 析构函数，释放动态数组的内存。
```

2.2.3 赋值运算符

```
Vector<T>& operator=(const Vector& other); // 拷贝赋值运算符。

Vector<T>& operator=(Vector&& other); // 移动赋值运算符。
```

2.2.4 元素访问

```
T& at(size_t pos); // 安全访问元素的方法，检查索引是否越界。

const T& at(size_t pos) const; // 安全访问元素的方法（常量版本），检查索引是否越界。

T& operator[](size_t pos); // 重载下标运算符，用于访问元素。

const T& operator[](size_t pos) const; // 重载下标运算符（常量版本），用于访问元素。

T& front(); // 返回向量的第一个元素。

const T& front() const; // 返回向量的第一个元素（常量版本）。

T& back(); // 返回向量的最后一个元素。

const T& back() const; // 返回向量的最后一个元素（常量版本）.

T* data(); // 返回指向动态数组的指针。

const T* data() const; // 返回指向动态数组的指针（常量版本）.
```

2.2.5 容量

```
bool empty() const; // 判断向量是否为空。

size_t size() const; // 返回向量中当前存储的元素数量。

size_t max_size() const; // 返回向量支持的最大元素数量。

void reserve(size_t new_capacity); // 预留足够的容量以容纳指定数量的元素。
```

2.2.6 修改器

```
void clear(); // 清空向量。

void insert(size_t pos, const T& value); // 在指定位置插入一个元素。

void erase(size_t pos); // 删除指定位置的元素。

void resize(size_t new_size); // 调整向量的大小。

void swap(Vector& other); // 交换两个向量的内容.
```

2.2.7 非成员函数

```
// 判断两个向量是否相等。
template <typename T>
bool operator==(const Vector<T>& lhs, const Vector<T>& rhs);

// 判断两个向量是否不等。
```

```
template <typename T>
bool operator!=(const Vector<T>& lhs, const Vector<T>& rhs);

// 判断一个向量是否小于另一个向量。
template <typename T>
bool operator<(const Vector<T>& lhs, const Vector<T>& rhs);

// 判断一个向量是否小于等于另一个向量。
template <typename T>
bool operator<=(const Vector<T>& lhs, const Vector<T>& rhs);

// 判断一个向量是否大于另一个向量。
template <typename T>
bool operator>(const Vector<T>& lhs, const Vector<T>& rhs);

// 判断一个向量是否大于等于另一个向量。
template <typename T>
bool operator>=(const Vector<T>& lhs, const Vector<T>& rhs);
```

3 测试情况

3.1 测试情况测试样例设计

3.1.1 基本功能测试

1. 测试创建空向量并验证其为空和大小为0:

```
Vector<int> vec1;
assert(vec1.empty());
assert(vec1.size() == 0);
```

2. 测试创建指定大小的向量并验证其大小和元素默认初始化:

```
Vector<int> vec2(5);
assert(!vec2.empty());
assert(vec2.size() == 5);
```

3. 测试创建指定大小的向量，元素初始化为指定值，验证元素值:

```
Vector<int> vec3(3, 10);
assert(vec3.size() == 3);
assert(vec3[0] == 10);
assert(vec3[1] == 10);
assert(vec3[2] == 10);
```

4. 测试使用初始化列表创建向量，验证元素值:

```
Vector<int> vec4 = {1, 2, 3};
assert(vec4.size() == 3);
assert(vec4[0] == 1);
assert(vec4[1] == 2);
assert(vec4[2] == 3);
```

5. 测试拷贝构造一个向量，验证内容相等:

```
Vector<int> vec5 = vec4;
assert(vec4 == vec5);
```

6. 测试移动构造一个向量，验证内容正确：

```
Vector<int> vec6(std::move(vec4));
assert(vec4.empty());  // vec4应为空
assert(vec6.size() == 3);  // vec6应保留内容
```

7. 测试使用拷贝赋值运算符，验证内容相等：

```
vec4 = vec5;
assert(vec4 == vec5);
```

8. 测试使用移动赋值运算符，验证内容正确：

```
vec4 = std::move(vec6);
assert(vec6.empty());  // vec6应为空
assert(vec4.size() == 3);  // vec4应保留内容
```

9. 测试修改向量元素值，验证修改成功：

```
vec4[0] = 42;
assert(vec4[0] == 42);
```

10. 测试向量的元素访问方法，包括 at、front 和 back：

```
assert(vec4.at(0) == 42);
assert(vec4.front() == 42);
assert(vec4.back() == 3);
```

11. 测试获取向量的动态数组指针，验证指针不为空：

```
int* data_ptr = vec4.data();
assert(data_ptr != nullptr);
```

12. 测试判断向量是否为空，验证结果正确：

```
assert(!vec4.empty());
```

13. 测试获取向量的大小，验证大小正确：

```
assert(vec4.size() == 3);
```

14. 测试预留足够的容量，验证容量增加：

```
vec4.reserve(10);
assert(vec4.size() == 3);  // 大小不变
assert(vec4.capacity() >= 10);  // 容量增加
```

15. 测试清空向量，验证向量为空：

```
vec3.clear();
assert(vec3.empty());
```

16. 测试在指定位置插入元素，验证插入成功：

```
vec4.insert(1, 8);
assert(vec4[1] == 8);
assert(vec4.size() == 4);
```

17. 测试删除指定位置的元素，验证删除成功：

```
vec4.erase(2);
assert(vec4.size() == 3);
assert(vec4[2] == 3);
```

18. 测试调整向量大小，验证大小和元素正确：

```
vec4.resize(5);
assert(vec4.size() == 5);
assert(vec4[3] == 0);  // 默认构造的新元素
```

19. 测试交换两个向量的内容，验证内容交换正确：

```
Vector<int> vec7 = {7, 8, 9};
vec4.swap(vec7);
assert(vec4 != vec7);
```

3.1.2 可靠性测试

20. 测试尝试越界访问向量元素，验证异常捕获：

```
try {
    int x = vec4.at(10);  // 试图访问越界元素
} catch (const std::out_of_range& e) {
    std::cerr << "Exception: " << e.what() << std::endl;
}
```

3.1.3 其他测试

21. 验证向量之间的相等性、不等性、大小比较操作：

```
Vector<int> vec8 = {1, 2, 3};
Vector<int> vec9 = {1, 2, 4};
assert(vec8 < vec9);
assert(vec8 <= vec9);
assert(vec8 != vec9);
assert(vec9 > vec8);
assert(vec9 >= vec8);
```

22. 验证向量之间的加法、减法、数乘和内积运算：

```
Vector<int> vec10 = {1, 2, 3};
Vector<int> vec11 = {2, 3, 4};
Vector<int> sum = vec10 + vec11;
Vector<int> diff = vec11 - vec10;
Vector<int> product = vec10 * 2;
int dot_product = vec10 * vec11;
assert(sum == Vector<int>({3, 5, 7}));
assert(diff == Vector<int>({1, 1, 1}));
assert(product == Vector<int>({2, 4, 6}));
assert(dot_product == 20);
```

3.2 测试结果对程序的改进结果

经过测试，程序在基本功能测试、可靠性测试和其他测试中均表现出良好的稳定性和正确性。没有出现异常或错误。测试结果表明，向量类模板 `Vector` 的设计和实现是可靠的，符合项目要求。

测试结果对程序的改进结果主要有以下几点：

1. **验证正确性和可靠性**： 测试有助于验证程序的正确性和可靠性，确保向量类在各种情况下都能正确工作。
2. **提高代码质量**： 测试能够发现潜在的问题和错误，有助于改进代码质量，减少潜在的错误和异常情况。
3. **指导优化和扩展**： 测试还可以指导对程序进行优化和扩展。通过测试，可以发现需要改进的地方，从而优化程序。