C++语言程序设计作业2

助教：叶开 ye_kai@pku.edu.cn

2022年10月11日

- 说明：完成代码的各个部分，实现一个自己的vector容器
- 如果有问题随时联系助教，包括讲义作业错误和学习上的困难
- 评分规则：
  - 每个文件都通过编译，输出无误 (5*15%=75%)
  - 代码是按照说明正确实现的 (5*5%=25%)
  - 每逾期一天，减少 10%，至多减少 50%
- 提交：
  - 截止：2022年10月25日23:59
  - 请在教学网提交 `main1.cpp ~ main5.cpp` 的**压缩包**，重命名为学号，例如 `2100012345.zip`，无需其他任何文件

Part0

- 我们的目标：实现可变长度的数组容器Vector
- 如何实现数组？思路1：我们能这样写吗？

```
template<typename T>
struct Vector {
private:
    T m_data[N]; // ...?
};
```

不可以！这种数组的长度不能改变！

- 思路2：我们能这样写吗？

既然数组长度不能改变，我们干脆预先准备一个超大数组，然后用一个变量记录其中有效的长度。

```
template<typename T>
struct Vector {
private:
    T m_data[100000];
    int m_size;
};
```

从 `m_data[0]` 到 `m_data[m_size - 1]` 之间的元素，才是数组实际上有效的元素；其他的部分仅仅是备用的“槽位”，如果数组发生了增长，那么会有越来越多的槽位被占用。这样我们只要通过调整 `m_size` 就可以实现长度的变化了。

不可以！这种写法占用了太多的内存！无论我们实际上需要几个元素，都会申请十万个元素的空间。

- 思路3：我们能这样写吗？

  既然定长数组有各种困难，我们干脆使用 `new[]` 来分配空间，需要多少就分配多少。

```
template<typename T>
struct Vector {
public:
    Vector(int size) {
        this->m_data = new T[size];
        this->m_size = size;
    }
private:
    T* m_data;
    int m_size;
};
```

问题：`push_back` 时应该如何操作呢？`m_data` 已经装不下新的元素了？

解决方案：重新分配 `m_data` 的空间（先申请 `m_size + 1` 的空间，然后把 `m_data` 复制过去，再删除原有的 `m_data` ）

不可以！这样每次 `push_back` 都会使得整个空间重新分配，效率太低！

- 思路4：结合思路2和3，每次重新分配，但额外分配一倍的空间
  - 这时用两个变量来记录空间长度
  - `m_size` 记录实际的元素数量（不包括备用的“槽位”）
  - `m_capacity` 记录最大可用的元素数量（即，包括那些额外分配的备用空间）

```
template<typename T>
struct Vector {
public:

    // ...

    void push_back(const T& element) {
        // ...

        if (m_size == m_capacity) {   // no more space
            int new_capacity = m_capacity * 2; // twice space
            if (new_capacity <= m_size) {
                new_capacity = m_size + 1; // when m_capacity = 0
            }
```

```
            T* new_data = new T[new_capacity];
            for (int i = 0; i < m_size; i++) {
                new_data[i] = m_data[i]; // copy old data
            }
            new_data[m_size] = element; // add new element
            ++m_size; // only add 1
            m_capacity = new_capacity; // twice space
            delete[] m_data; // delete old data first
            m_data = new_data; // assign new data then
        }

        // ...
    }

private:
    T* m_data;
    int m_size;
    int m_capacity;
};
```

这样即保证了长度可以变化，又兼顾了空间利用率和时间效率。

Part1 初始化

- 阅读第一部分代码 `main1.cpp`
- 根据提示，尝试补全代码并运行，检查程序是否正常运行

Code

```
#include <iostream>

template<typename T>
struct Vector {
public:
    Vector();
    Vector(int size);
    Vector(int size, const T& value);
    ~Vector();
private:
    T* m_data;
    int m_capacity;
    int m_size;
};

int main() {
    Vector<int> a;
    Vector<std::string> b(10);
    Vector<double> c(10, 3.14);
}
```

- 阅读代码，理解 `Vector` 类的成员

    - `m_data`，用来储存创建的数组
        - 可以想见，我们应该根据所需元素的数量，通过 `new[]` 来申请空间
    - `m_capacity`，用来记录 `m_data` 存储空间的长度
        - 即，通过 `new[]` 申请空间时的长度，这就是 `m_data` 最多存储元素的数量
    - `m_size`，用来记录 `m_data` 已使用空间的长度
        - 注意这里与 `m_capacity` 不同，详见Part0思路4
- **注意：为了方便起见，我们不考虑恶意/非法输入，即，不需要特地处理 `Vector(-1)` 这样的情况**
- 请补充三个构造函数和一个析构函数的定义，使得代码通过编译、顺利运行。
- 三个构造函数的功能如下：
    - `Vector()`，创建一个空数组
        - 思考，此时 `m_capacity` 和 `m_size` 取何值，而 `m_data` 应该如何设置？
    - `Vector(int size)`，创建一个具有 `size` 个元素的数组
    - `Vector(int size, const T& value)`，创建一个具有 `size` 个元素的数组，每个元素的值都是 `value`
- 析构函数的功能：
    - 使用 `delete[]` 销毁已经分配的空间
        - 如果不这么做，显然会导致内存泄漏，即，忘记释放不再使用的空间，会使得内存越来越少
    - 思考：销毁空间时，有特殊情况吗？还是直接 `delete[] m_data;` 就可以？

Part2 复制

- 将 `main1.cpp` 拷贝成 `main2.cpp`，然后按照要求修改 `main2.cpp`
- 根据提示，尝试补全代码并运行，检查程序是否正常运行

Code

向 `main2.cpp` 中的 `Vector` 类加入复制构造函数，并修改 `main` 函数为：

```
int main() {
    const Vector<int> a(10);
    Vector<int> b(a);

    const Vector<std::string> c;
    Vector<std::string> d(c);

    const Vector<std::string>& e = d;
    Vector<std::string> f(e);
}
```

- 思考：如果不自己写复制构造函数，而是使用默认的，是否可以？
    - 不可以！存在两个问题：
        1. 直接复制 `m_data` 会导致，对一个 `Vector` 的修改，将影响到另一个 `Vector`
        2. 析构时将会发生致命错误，对同一个 `m_data` 进行两次 `delete[]`（思考：为什么）
- 因此，我们必须自己重新写复制构造函数：

```
1. template<typename T>
2. struct Vector {
3. public:
4.     // ...
5.     Vector(const Vector<T>& other);
6. private:
7.     // ...
8. };
9.
10. template<typename T>
11. Vector<T>::Vector(const Vector<T>& other) {
12.     // ...
13. }
```

这里有几个问题：

1. 第10行为什么要写 `template<typename T>` ？

   因为我们想要在类外定义函数，必须告诉编译器要定义哪一个类。而 `Vector` 并不是一个类，它仅仅是类模板；`Vector<T>` 才是一个类。但问题在于，编译器并不理解T是什么，所以我们需要 `template<typename T>` 来表示T是一个typename，这样一来 `Vector<T>` 才是一个类名。

2. 第11行为什么要写 `Vector<T>::` ？

   同上，我们必须告诉编译器，现在是对 `Vector<T>` 这个类的函数进行定义。

3. 第11行为什么要写 `Vector<T>::Vector(...)` ，而不是 `Vector<T>::Vector<T>(...)` ？

   前者是简化的、符合标准的构造函数声明方式，当我们声明或定义模板类的构造函数时，可以省略函数名后面的模板参数（即第二个 `<T>` ）。

4. 第11行为什么要写 `const Vector<T>& other` ，而不是 `const Vector& other` ？

   同上，`Vector` 并不是一个类型；函数的参数必须是完整的类型，而 `Vector<T>` 是一个完整的类型。

- 最后，注意复制构造函数的细节

  1. `other` 可能是空的
  2. `m_capacity` 如何设置?

Part3 赋值

- 将 `main2.cpp` 拷贝成 `main3.cpp` ，然后按照要求修改 `main3.cpp`
- 根据提示，尝试补全代码并运行，检查程序是否正常运行

Code

向 `main3.cpp` 中的 `Vector` 类加入赋值运算符 `operator=` ，并修改 `main` 函数为：

```
int main() {
    Vector<std::string> a(10);

    const Vector<std::string> c(5, "20");

    a = c;

    const Vector<std::string>& d = c;

    Vector<std::string> e = (a = d);
}
```

- 思考：如果不自己写赋值运算符 `operator=`，而是使用默认的，是否可以？
- 如果不可以用默认的，思考如何自己写赋值运算符：
  - 返回类型应该是什么？
    - 自身的引用
  - 原有的数据怎么办？
    - 原有的 `m_data` 如果非空，则必须销毁，否则内存就泄露了（思考为什么）

Part4 遍历元素

- 将 `main3.cpp` 拷贝成 `main4.cpp`，然后按照要求修改 `main4.cpp`
- 根据提示，尝试补全代码并运行，检查程序是否正常运行

Code

向 `main4.cpp` 中的 `Vector` 类分别加入索引和查询元素数量的功能 `operator[]` 和 `size`，并修改 `main` 函数为：

```
int main() {
    Vector<int> a(10);
    for (int i = 0; i < a.size(); i++) {
        a[i] = i + 1;
    }
    const Vector<int> b = a;
    for (int i = 0; i < b.size(); i++) {
        std::cout << b[i] << std::endl; //
    }
}
```

- 我们需要两个版本的 `operator[]`，思考为什么、它们的返回类型分别是什么（如果不确定下面的问题，可以先按自己想法试一试，尝试编译能否通过）：
  - **简便起见，我们不需要检查索引 `index` 是否合法**
  - `const T& operator[](int index) const;`
    - 为什么是 `const T&` 而不是 `T`？
    - 为什么是 `const T&` 而不是 `T&`？
    - 为什么最后要加 `const`？
  - `T& operator[](int index);`

    - 为什么是 `T&` 而不是 `T`？
    - 为什么这里最后不加 `const`？
- 我们只需要一个版本的 `size()`，**简便起见，它的返回类型是 `int`，**功能是返回元素的数量（就像 `std::vector<T>` 里的那样）：
    - 为什么不需要另一个版本?
    - 返回 `m_size` 还是 `m_capacity`？
    - 为什么不让 `m_size` 或 `m_capacity` 变成 `public` 的，然后直接读取，而是要专门写一个 `size()` 方法?

Part5 增删

- 将 `main4.cpp` 拷贝成 `main5.cpp`，然后按照要求修改 `main5.cpp`
- 根据提示，尝试补全代码并运行，检查程序是否正常运行

Code

向 `main5.cpp` 中的 `Vector` 类分别加入增删的功能 `push_back` 和 `pop_back`，并修改 `main` 函数为：

```
int main() {
    Vector<int> a, b;
    for (int i = 1; i <= 5; i++) {
        a.push_back(i);
        b.push_back(i);
        b.push_back(i);
    }
    a = b;
    for (int i = 1; i <= 5; i++) {
        a.push_back(i);
    }
    for (int i = 0; i < a.size(); i++) {
        std::cout << a[i] << std::endl;
    }
}
```

- 这里对这两个函数进行讨论：
    - `void push_back(const T& element);`

      按照我们在Part0介绍的技术，`push_back` 内部应该检查 `m_size` 和 `m_capacity`，来决定是否需要重新分配空间；对于使用者来说，它的效果（看起来）仅仅是在数组的末尾加上了一个元素，就像我们在 `std::vector<T>` 里看到的那样。
    - `void pop_back();`

      这个函数的效果是删除最末尾的元素，显然，只需要修改 `m_size` 就可以实现这个功能（思考：为什么）。**为了简便起见，我们不需要考虑其他事情**（比如重新分配空间，用更小的 `m_capacity` 来适应更小的 `m_size`，以节约内存；比如主动调用元素的析构函数等），**另外，如果在空数组上使用这个函数，则什么事也不做。**