字符串类、向量类模板的设计

1 字符串类

1．1 项目概述

1.1.1 项目内容及要求

本项目旨在设计一个字符串类（MyString）以封装 C 风格的字符串。在本类中我们小组仿照 C++标准模板库中的“string”在“MyString”类中设计了转换构造函数、拷贝构造函数、移动构造函数、析构函数、返回对象属性的 len()、c_string（）函数、重载了适当的运算符（如：<<、>>、+、+=、<、<=、>、>=、==、!=、[ ]等）。为了提高程序的健壮性，使其能够具备处理异常的能力,我们小组在“MyString”类中定义了“OutfBoundsException”异常处理类。

1.1.2 研究人员和分工

序号	学号	姓名	角色和贡献
1			组长 项目字符串类的编写
2			组员 字符串类的修改 研讨报告的编写
3			组员 程序的调试 主调函数的编写
4			组员 汇报 PPT 的制作

1.2 类的设计

在 MyString 类中我们设计了两个私有数据成员——size_t 类型的 len 和字符指针 data，分别表示 MyString 对象的字符串长度和其 C_字符串成员的地址。为了实现该字符串类的基本功能，我们模仿 C++标准模版库中的 string 类设计了转换构造函数、拷贝构造函数、移动构造函数、析构函数、返回对象属性的 len()、c_string（）函数、重载了适当的运算符

（如：<<、>>、+、+=、<、<=、>、>=、==、!=、[ ]等）。

首先，对于转换构造函数：

```
MyString(const char* str = nullptr);
```

```
MyString::MyString(const char* str) {
    if (str) {
        len = std::strlen(str);
        data = new char[len + 1];
        std::strcpy(data, str);
    } else {
        len = 0;
        data = new char[1];
        data[0] = '\0';
    }
}
```

为了使用者的方便以及提高程序运行的安全性，我们为形参字符常量指针 str 指定了默认值 nullptr。对于 MyString 对象的 len 成员，我们调用 C 语言的字符串处理函数 strlen()来获得字符串长度，以保证属性的准确性。我们为 data 字符串单独开辟了堆空间，以保证数据封装。应项目要求中 MyString 类能封装 C_字符串的要求，我们在 data 字符串末尾加上了结束符’\0’。

为了提高性能和避免不必要的资源浪费，我们还设计了移动构造函数：

```
MyString(MyString&& other) noexcept;
```

```
MyString::MyString(MyString&& other) noexcept : data(other.data),
len(other.len) {
    other.data = nullptr;
    other.len = 0;
}
```

该函数将对象资源从另一个对象“转移”到本对象而不是进行复制。在具体实现上，我们用冒号语法将另一个对象的资源浅拷贝给本对象，然后释放另一个对象的资源。因为这种构造形式不会造成对象之间共享资源以致影响对象之间的独立性，所以运用浅拷贝即可。

为了让使用者在类外也能够读取对象的数据成员，我们设计了以下两个成员函数：

```
size_t length() const;
const char* c_str() const;
```

这两个函数分别返回对象的数据成员。

在运算符重载方面，为了实现字符串对象的赋值，我们重载了“=”赋值运算符：

```
MyString& operator=(const MyString& other);
```

```
MyString& MyString::operator=(const MyString& other) {
    if (this != &other) {
        delete[] data;
        len = other.len;
        data = new char[len + 1];
        std::strcpy(data, other.data);
    }
    return *this;
}
```

这里我们通过深赋值确保运算符两端对象的独立性。为了使赋值运算符能实现连续赋值，我们令函数的返回值为本对象。

像构造函数一样，我们还重载了移动赋值运算符：

```
  MyString& operator=(MyString&& other) noexcept;
```

```
MyString& MyString::operator=(MyString&& other) noexcept {
    if (this != &other) {
        delete[] data;
        data = other.data;
        len = other.len;
        other.data = nullptr;
        other.len = 0;
    }
    return *this;
}
```

同样地，在为实现字符串合并设计的“+”运算符重载中也通过返回合并后的对象实现使用中的连加功能。

为了实现字符串对象内容的访问，我们重载了取下标运算符：

```
char& operator[](size_t index);
```

```
char& MyString::operator[](size_t index) {
    if (index >= len) throw OutOfBoundsException("Index out of
bounds");
    return data[index];
}
```

为了处理下表访问时的越界问题，我们在类中设计了异常处理类：

```
class OutOfBoundsException : public std::runtime_error {
    public:
        OutOfBoundsException(const std::string& message)
            : std::runtime_error(message) {}
    };
```

此类派生自 C++标准库中的 runtime_error 类。在调用该函数时用“what（）;”函数；来读取捕捉的异常类对象的内容并输出“Index out of bounds”异常。

我们还重载了流插入运算符和流提取运算符：

```
friend std::ostream& operator<<(std::ostream& os, const MyString&
str);
friend std::istream& operator>>(std::istream& is, MyString& str);
```

```
std::ostream& operator<<(std::ostream& os, const MyString& str) {
    os << str.data;
    return os;
}

std::istream& operator>>(std::istream& is, MyString& str) {
    char buffer[1000];
    is >> buffer;
    str = MyString(buffer);
    return is;
}
```

首先，与上述原因类似，为了实现在处理输出、输入流对象时的连续性，我们令函数返回值为输入或输出流对象的引用。另外，在重载流提取运算符时，我们调用了转换构造函数来实现本对象的修改。

对于“<、<=、>、>=、==、!=”等关系运算符的重载，我们都调用了相应功能类似的 C_字符串处理函数。

此外，我们还设计了有插入，擦除以及生成子串功能的成员函数：

```
  MyString& insert(size_t pos, const MyString& str) ;
    MyString& insert(size_t pos, const char* str)  ;
    MyString& erase(size_t pos, size_t n);
    MyString& substring(size_t pos, size_t n) const;
```

这四个函数分别提供了：在本对象的指定位置插入字符串内容、从本对象的指定位置开始擦除指定长度个字符以及以本对象指定范围的字符为内容，生成相应子串。

这里以第一个函数为例子说明函数实现：

```
MyString &MyString::insert(size_t pos, const char *str)
{
    int n = strlen(data);
    if(pos>n || pos<0)
    throw OutOfBoundsException("position out of bounds");
    char *p = new char[strlen(data)+strlen(str)+1];
    strncpy(p, data,pos );
    p[pos] = '\0';
    strcat(p, str);
    strcat(p, data+pos);
```

```
    delete[] data;
    data = p;
    len = strlen(data);
    return *this;
}
```

首先，insert 函数的两个形式参数分别传入字符串的插入位置。然后通过 if 语句判断插入位置是否合法，如果 pos 大于本字符串的长或 pos 小于零，则抛出内容为 position out of bounds 的异常。再通过重新创建一个字符型数组来和动态的申请和释放空间来实现本字符串内容的改变。最后返回本对象实现字符串插入操作。

1.3.测试情况

2 向量类模板

2.1 项目概述

2.1.1 项目内容及要求

本项目旨在仿照 C++标准库中的 vector 容器设计一个向量类模板（Vector<T>）。这个动态数组类模板能够随时插入和删除元素，与一般数组相比，它能够动态的调节所占空间的大小。在此模版类的中，我们首先设计了构造、析构以及赋值相关的四大函数，而后从动态数组的功能性出发为其设计了增添、删除插入元素以及返回数组大小、容量的成员函数。为了使用者的便捷，我们在类中重载了“：<<、>>、+、+=、==、!=、[ ]等”必要的运算符。在重载这些运算符时，我们运用了 C++标准库中的 invalid_argument 以及 out_of_range 异常处理类来处理下标运算符中的越界问题以及加法运算符、比较运算符的向量维度不匹配的问题。

2.1.2 研究人员和分工

序号	学号	姓名	角色和贡献
1			组长 向量类模板的编写
2			组员 向量类模板的修改 研讨报告的编写

3			组员 程序的调试 主调函数的编写
4			组员 汇报 PPT 的制作

2.2 类的设计

在类的私有数据成员的设计上，出于对在增加数组成员时内存分配高效性的考虑，我们在数组 data 和动态数组的大小 size 外还设计了动态数组的最大容量 capacity。这个数据成员的存在使得使用者在对动态数组进行增添元素操作时，对象可以“未雨绸缪”地提前将最大容量增大到原来的两倍，避免了后续的增添维度时繁琐的内存增加操作，提高了运行效率。

```
T* data;          // 指向数组的指针
    size_t size;      // 当前元素数量
    size_t capacity;  // 数组的总容量
```

在构造函数的设计上，我们在常规的构造方法之外增添设计了与常规数组定义形式更为相近的初始化列表构造方法：

```
template<typename T> Vector<T>::Vector(std::initializer_list<T>
init) : size(init.size()), capacity(init.size()) {
        data = new T[capacity];
        size_t index = 0;
        for (const auto& value : init) {
            data[index++] = value;
        }
    }
```

我们引用了 C++标准库中的 initializer_list<T>类模板来传入初始化列表。运用 auto 类型来初始化迭代变量 value 遍历 init 中的每一个元素。

在设计 Vector 类的拷贝构造函数时，出于对对象资源独立性的考虑，我们小组采用了深拷贝的构造方法：

```
 template<typename T> Vector<T>::Vector(const Vector& other) :
size(other.size), capacity(other.capacity) {
        data = new T[capacity];
        for (size_t i = 0; i < size; ++i) {
            data[i] = other.data[i];
        }
    }
```

对于赋值运算符的重载，出于与以上相同的原因，我们仍采用深赋值的方法。另外，考考虑到使用时连续赋值的需求，我们让函数的返回值为本对象的引用：

```
template<typename T> Vector<T>::Vector(const Vector& other) :
size(other.size), capacity(other.capacity) {
        data = new T[capacity];
        for (size_t i = 0; i < size; ++i) {
            data[i] = other.data[i];
        }
    }
```

对于析构函数而言，由于对象是“带资源”，我们重载了析构函数：

```
template<typename T> Vector<T>::  ~Vector() {
        delete[] data;
    }
```

为了使用者在类外也能够访问对象的属性，我们设计了返回对象容量和实际大小的成员函数：

```
// 获取元素数量
    template<typename T> size_t Vector<T>::getSize() const {
        return size;
    }

    // 获取容量
    template<typename T> size_t Vector<T>:: getCapacity() const {
        return capacity;
    }
```

接下来就是动态数组模版类增添元素的成员函数：

```
template<typename T> void Vector<T>:: push_back(const T& value) {
        if (size >= capacity) {
            resize();
        }
        data[size++] = value;
    }
```

在进行增添操作之前，函数先检查对象容量的大小是否足够大，如果不够大就调用定义在 private 访问权限下的 resize 函数来增加动态数组的容量。这里需要说明，resize 函数之所以声明在类的 private 访问权限下，是因为要保证用户只能以公开接口中的方法操作动态数组，避免 size 大于容量的情况出现，增加类在运行时的稳定性。

为了实现动态数组插入元素的功能，我们重载了两个 insert 函数以实现在指定位置插入单个元素和在指定位置插入指定个数的元素。对于函数实现时插入位置大于数组大小或 index<0 的情况，我们引用 std::out_of_range 异常处理类抛出异常来解决。这里还需要说明的是：在拓展数组容量时，调用的是类中重载的第二个 resize 函数。原因是它能根据实

际需要“一步到位”拓展容量，而另一个 resize 函数仅能将数组对象的容量拓展为原来的两倍，无法保证容量满足插入需求。

```
 template<typename T> void Vector<T>:: insert(size_t index, const T&
value) {
        if(index > size)
            throw std::out_of_range("Index out of range");
        if (size >= capacity) {
            resize();
        }
        for (size_t i = size; i > index; --i) {
            data[i] = data[i - 1];
        }
        data[index] = value;
        ++size;
    }

    //向量范围插入
    template<typename T> void Vector<T>:: insert(size_t index, size_t
count, const T& value) {
        if(index > size)
            throw std::out_of_range("Index out of range");
        if (size + count > capacity) {
               resize();
        }
        for (size_t i = size; i > index; --i) {
            data[i+count] = data[i ];
        }
        for (size_t i = index; i < index + count; ++i) {
            data[i] = value;
        }
        cout << capacity << endl;
        size += count;
    }
```

类中定义的删除指定元素的成员函数实现方法则相对简单，只需直接对数组 data 进行相应的操作，但仍需处理 index 的越界问题。

对于取下标运算符的重载，也需要处理下标越界的情况。

```
template<typename T> T& Vector<T>::  operator[](size_t index) const{
        if (index >= size||index<0) {
            throw std::out_of_range("Index out of range");
        }
        return data[index];
    }
```

接下来的算数运算符的重载仍需要率先考虑两个向量维数不等的非法情况，并抛出异常。值得注意的是，加法运算符的重载中，函数的返回值不能为引用，因为作为临时变量被创建的对象 result 在函数调用结束时便会被销毁。而“+=”因为返回的是原来就存在的对象的引用所以返回引用。

2.3 测试情况