抽象向量类模板及其衍生类的设计

1、 项目概述

1.1、 项目内容及要求

在本项目中我们小组根据项目要求，设计了抽象向量类模板以及其衍生出来的字符串类和向量类模板。我们将抽象向量类模板（Vector）作为字符串类（String）以及向量类模版（DerivedVector）的父类，在其中包含了基类（Vector）的构造函数、析构函数、纯虚运算符重载、用于设置对象元素内容的纯虚函数（set）、用于访问对象元素内容的纯虚函数（get）以及用于输出对象内容的纯虚函数（display）。在向量类模板和向量类模板中覆盖定义了相应纯虚函数，已实现程序“运行时的多态性”，进而完成子类的具体实现。

1.2 研究人员和分工

序号	学号	姓名	角色和贡献
1			组长 项目向量类模板和字符串类的制作
2			组员 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、 类的设计

2.1、抽象向量类类模板的设计

抽象向量类模板将作为由其衍生出的字符串类和向量类模板的父类，其对象属性，即类模板数据成员，的设计要与其衍生类由一定的相关性。所以我们为其设计了用于储存向量大小和储存数据的两个数据成员：

```
protected:
    //size 储存向量大小
        size_t size;
```

```
    //data 储存数据
        T *data;
```

这里需要说明的是：考虑到在子类中定义函数时访问父类数据成员的需要，我们将父类数据成员的访问权限设置为“protected”。

接下来我们为基类设计了构造函数，具体来说分别是：初始化构造函数、带有参数的构造函数以及拷贝构造函数：

```
 //1.初始化构造函数
        Vector(size_t size):size(size),data(new T[size]){}

    //2. 参数化构造函数
        Vector(size_t size,const T *arr):size(size),data(new T[size]){
            std::copy(arr,arr+size,data);
        }
    //3. 拷贝构造函数
        Vector(const Vector<T> & other):size(other.size),data(new
T[other.size]){
            std::copy(other.data,other.data+other.size,data);
        }
```

这里需要书说明的是，我们为了设计方便，使用了 C++标准库中的 copy（）方法，其作用是将一个范围内的元素复制到另一个地址。

由于基类对象是“带资源的”所以我们为其设计了析构函数来释放堆空间，防止内存泄漏：

```
virtual ~Vector(){
            std::cout<<"调用了基类的纯虚函数"<<std::endl;
            //std::cout<<"调用了基类的纯虚函数"<<std::endl;
            delete []data;
        }
```

我们根据功能需要，还重载了移动赋值运算符：

```
//4. 虚移动赋值运算符
        virtual Vector& operator=(Vector && other){
            if(size!=other.getSize()){
                throw "维度不同！！无法赋值";
            }
            delete[]data;
            size=other.size;
            data=other.data;
            other.size=0;
            other.data=nullptr;
            return *this;
        }
```

这里设计移动赋值运算符仍是出于对程序运行效率的考虑。此处我们在进行赋值操作前进行了对两对象维度是否相等的判断，如维度不同，就抛出异常，以供捕捉和处理。需要说明的是：由于 other 对象的内容在赋值操作完成后就被“删除”了，所以此处对本对象数据 data 的浅赋值并不会影响对象资源之间的独立性。

我们重载了“自加运算符”。这里考虑到子类实现自加运算符的方法存在差异，为了实现“一个接口，多个方法”的动态多样性，我们将自加运算符的重载声明为虚函数：

```
DerivedVector<T>& operator+=(const Vector<T> & other) override{
            if(this->size!=other.getSize()){
                throw "维度不一样的向量不能相加";
            }
            for(size_t i=0;i<other.getSize();++i){
                this->data[i]+=other.get(i);
            }
            return *this;
            //return DerivedVector()
        }
```

这里于重载移动赋值运算符是的操作相似，也是在自加操作前先比较两向量的为度是否相等，如不相等，就抛出异常。之后调用定义在类中的 get（）方法访问对象元素。

我们还增加了在向量尾部增添元素的成员函数 append（），并将此函数声明为纯虚函数。

之后我们声明了成员函数 get（）、set（）以及 display（）分别实现对都想元素的访问和设置以及输出对象。对于前两者我们都设计了对于越界异常的处理。对于 display（）则使用常规的标准输出流实现。

2.2、字符串衍生类的设计

正如项目简介中所说的，字符串类为衍生自抽象向量类模板的子类。这里考虑到字符串的要求，我们选择用 char 类型实例化的 Vector 类模板作为字符串类的父类。我们将 String 类设计成为一个可以包装 C——字符串的字符串类。首先对该类的构造函数来说，我们将父类中的一个构造函数覆盖定义了：

```
String(size_t Size, const char *arr)  {
            int n = strlen(arr);
            if(Size>n)
            Size = n;
            else if(Size < 0)
            Size = 0;
            size=Size;
            data=new char[Size+1];
            for(int i=0;i<Size;++i)
            data[i]=arr[i];
            data[Size]='\0';
        }
```

通过调用 C_字符串处理函数，我们实现了用 C_字符串构造 String 类对象的功能。这里需要说明的是形参 Size 和对象的 size 数据成员均为本 String 类对象所封装字符串的有效长度。

接下来我们定义了访问对象大小的成员函数 getSize()：

```
size_t getSize()const{
            return size;
        }
```

在类中我们覆盖定义了基类中的纯虚函数：get（）和 set（）：

```
void set(size_t index,const char & value)override{
            if(index>=this->size){
                throw "超出界限！！不能设置！";
            }
            this->data[index]=value;
        }
        char get(size_t index)const override{
            if(index>=this->size){
                throw "超出界限！！没有数据！";
            }
            return this->data[index];
        }
```

在函数实现时我们加入了异常抛出的机制，已处理访问越界的问题。

我们还覆盖定义了基类中的自加运算符的重载：

```
String & operator+=(const Vector<char>& other)override{
            size_t newsize=this->size+other.getSize();
            char* newdata=new char[newsize+1];

            for(size_t i=0;i<this->size;++i){
                newdata[i]=this->data[i];
            }
            for(size_t i=0;i<other.getSize();++i){
                newdata[i+this->size]=other.get(i);
            }
            newdata[newsize]='\0';
            this->size=newsize;
            this->data=newdata;
            return *this;
```

在函数实现中，我们加入了 String 类封装 C_字符串的属性。

对 append（）纯虚函数的实现，我们仍然遵循封装 C_字符串的原则：

```
void append(const char &value) override {
            char* newdata=new char [size+1];
```

```
            for(size_t t=0;t<size;t++)
            {
                newdata[t]=data[t];
            }
            newdata[size]=value;
            size++;
            data=newdata;
            delete[]newdata;
        }
```

这里为了防止内存泄漏，我们在使用了 newdata 里的内容后就将其申请的堆空间释放了。

2.3、向量模板衍生类的设计

此向量类模板衍生自基类抽象向量类模板。在设计构造函数时，我们仅显式地使用冒号语法来调用基类的构造函数，就完成了对本类对象的构造：

```
DerivedVector(size_t size, const T *arr) : Vector<T>(size, arr) {}
```

此向量类的纯虚函数的覆盖定义与字符串类相似。这里要着重说明的是次类中对父类中 append 纯虚函数的实现因为 capacity 数据成员的存在而更加简洁，有着更高的效率：

```
//实现纯虚函数 append
        void append(const T &value) override
        {
            if(this->size+1>capacity)
            resize();
            this->data[this->size]=value;
            this->size++;
        }
```

与 String 类中 append 纯虚函数的实现相比，此处明显少了对 data 内数据的大范围转移。