视觉组第五次培训 C++进阶

1. 面向对象的一些补充

1.1 将多个逻辑上相关的变量包装成一个类，既能提高代码的可读性，又能避免程序员犯错

例如你想实现一个可变大小的数组，且数组初始化为0，如果没有面向对象，你可能会这么写，可见如果没有面向对象，你要管理这个可变数组，就得额外管理别的变量，例如这里的size。而且size变量可以被随意修改，这是很危险的，你可能会忘记修改数组的大小，导致数组越界的。

```
int size = 4;
int *array = new int[size];
for (int i = 0; i < size; i++) {
    array[i] = 0;
}
//当你需要改变数组的大小时
delete[] array;
size = 8;
array = new int[size];
for (int i = 0; i < size; i++) {
    array[i] = 0;
}
//危险操作，无缘无故修改size变量
size = 6;
```

但如果你是面向对象的，你可以这么写

```
class Array {
public:
    Array(int size) {
        this->size = size;
        array = new int[size];
        for (int i = 0; i < size; i++) {
            array[i] = 0;
        }
    }

    ~Array() {
        delete[] array;
    }

    void resize(int new_size) {
        delete[] array;
        size = new_size;
        array = new int[size];
        for (int i = 0; i < size; i++) {
            array[i] = 0;
```

```
        }
    }

    int operator[](int index) {
        return array[index];
    }

    void size() {
        return size;
    }

private:
    int size; // 防止外部修改size
    int *array;
};
```

这样你使用起来就会方便很多，这在大型项目里十分有用

```
Array array(4);
// 改变数组大小
array.resize(8);
for (int i = 0; i < array.size(); i++) {
    std::cout<<array[i]<<" ";
}
```

1.2 将相关的量封装在一起

在实际工程中，经常遇到需要修改一个成员时，其他成员也需要被修改，否则出错。当遇到这种情况时，意味着你需要把成员变量的读写封装成成员函数

在下面的例子中，当你修改了学生的姓名或班级时，学生的id也需要被修改，所以你需要将这三个变量设为私有，然后封装getter和setter用以访问和修改变量

```
class Student {
public:
    Student(std::string name, int class) : name_(name), class_num_(class) {
        id = std::to_string(class)+name;
    }

    void setName(std::string name) {
        name_ = name;
        id = std::to_string(class_num_)+name_;
    }

    void setClass(int class_num) {
        class_num_ = class_num;
        id = std::to_string(class_num_)+name_;
    }

    std::string getName() {
        return name_;
```

```
    }

    int getClass() {
        return class_num_;
    }

    std::string getId() {
        return id_;
    }

private:
    std::string id_;
    int class_num_;
    std::string name_;
}
```

1.3 RAII

RAII（Resource Acquisition Is Initialization）是C++一个重要的编程思想，作为一个没有垃圾回收（GC）机制的语言，程序员需要自己管理程序中分配的内存。RAII是一个避免内存泄露的一个很好的方法，简单说，它的思想就是**在构造函数中申请(new)资源，在析构函数中释放(delete)资源**。

```
class Array {
public:
    // 在构造函数中申请资源
    Array(int size) : size_(size) {
        data_ = new int[size];
    }
    // 在析构函数中释放资源
    ~Array() {
        delete[] data_;
    }
private:
    int size_;
    int *data_;
};
```

由于当对象销毁的时候，析构函数会自动调用，所以你就不需要担心new了忘记delete了。

2. STL库

STL就是Standar Template Library，标准模板库，是C++官方提供的功能非常强大的一系列库。STL分为六大组件

组件	功能
容器Container	各种数据结构，例如vector，list，set，map等
算法Algorithm	各种常用算法，例如sort，find，copy等

组件	功能
迭代器Iterator	用于遍历容器
仿函数Function object	类似于函数的类，可以作为算法的参数
适配器Adaptor	用于修饰容器或仿函数，例如stack，queue，priority_queue等
空间配置器Allocator	负责空间的配置与管理

这里就简单介绍下容器的使用，其他的组件可以自行查阅资料

2.1 vector

vector是最常用的容器，下面就以vector为例介绍STL的使用

vector你可以理解为动态数组，跟Java中的ArrayList类似

vector的初始化

```
#include <vector>

std::vector<int> v1; // 空的int类型vector
std::vector<int> v2(10); // 10个元素的vector，每个元素都是0
std::vector<int> v3(10, 1); // 10个元素的vector，每个元素都是1
std::vector<int> v4(v3); // v4是v3的拷贝
std::vector<int> v5 = v3; // v5也是v3的拷贝
std::vector<int> v6 = {1,2,3,4,5}; // 列表初始化
```

vector的基本操作

```
#include <vector>

srd::vector<int> v = {1,2,3,4,5};
// 可以像数组一样访问
std::cout<<v[1]<<"\n";
int a = v[2];
v[0] = 3;
// 往vector的末尾添加元素
v.push_back(6);
// 删除vector的末尾元素
v.pop_back();
// 获取vector的大小
std::cout<<v.size()<<"\n";
// 判断vector是否为空
if (v.empty()) {
    std::cout<<"v is empty\n";
}
```

配合迭代器使用

```
#include <vector>

std::vector<int> v = {1,2,3,4,5};
// 迭代器是一种类似于指针的对象，用于遍历容器
// 迭代器的类型是vector<int>::iterator
for (std::vector<int>::iterator it = v.begin(); it != v.end(); it++) {
    std::cout<<*it<<" ";
}
// 特殊的迭代器
v.begin(); //返回指向第一个元素的迭代器
v.end(); // 返回指向最后一个元素的下一个位置的迭代器
```

遍历vector

```
// 1.使用下标遍历
for (int i = 0; i < v.size(); i++) {
    std::cout<<v[i]<<" ";
}
// 2.使用迭代器遍历
for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
    std::cout<<*it<<" ";
}
// 3.基于范围的for循环(C++11)
for (int i : v) {
    std::cout<<i<<" ";
}
```

一些常用的algorithm

```
#include <vector>
#include <algorithm>

std::vector<int> v = {6,1,3,5,2,4};
// 排序
std::sort(v.begin(), v.end()); //执行完后v={1,2,3,4,5,6}
// 查找
std::vector<int>::iterator it = std::find(v.begin(), v.end(), 3); //返回指向3的迭代器
it = std::find(v.begin(), v.end(), 7); //返回v.end()，表示没找到
```

3. C++内存管理

C++程序在运行时会将内存分为四个分区，分别是堆区，栈区，全局/静态存储区和代码区，程序员需要关注的是堆区和栈区。

3.1 栈和堆

栈

我们知道，计算机程序都是写在内存里的。当运行中的程序进入一个函数时，计算机会自动为这个函数分配一块内存，用于存放这个函数的局部变量，这块内存就是栈区，也叫**栈帧**。当函数运行返回时，计算机会自动释放这块内存。

栈区的特点是**自动分配，自动释放**，这是由操作系统完成的。由于函数的调用非常频繁，而且函数会层层嵌套甚至递归调用，所以栈区是非常有限的，一般只有几MB。

> 想想也是，如果函数一个栈帧就几百MB，那我递归个几层，不就把内存炸了？

所以，如果你在一个函数中，定义一个非常大的数组，超过了函数的栈帧大小，那么程序就会崩溃，也就是所谓的**栈溢出**。

堆

与栈不同，堆区是属于整个程序而不是属于某个函数的。所以堆区的大小非常大。

所以在C/C++中，如果你想申请一个大数组，你就需要使用**new**关键字在堆区申请内存，以免爆栈。当你不再需要这块内存时，使用**delete**关键字释放内存。

```
int *array = new int[100000]; // 申请十万个int类型的内存
delete [] array; // 释放内存
```

3.2 智能指针

在C++11之前，程序员需要手动管理内存，这是非常容易出错的，例如忘记释放内存，或者释放了内存但后面又使用了这块内存等等。前者会导致程序占用内存越来越大造成内存泄漏，后者更是直接导致程序崩溃。

C++11引入了智能指针，可以帮助程序员自动管理内存，避免内存泄漏和野指针。

简单理解，智能指针就是在构造的时候帮你new，再对象销毁的时候自动帮你delete

```
#include <memory>

std::shared_ptr<int> p1(new int(1)); // p1是一个指向int类型的智能指针
std::cout<<*p1<<"\n"; // 输出1，在大多数情况下，智能指针的使用和普通指针没什么区别
```

智能指针分为三种

- shared_ptr：采用引用技术，可以有多个shared_ptr指向一个对象，当最后一个shared_ptr销毁时，对象也会被销毁
- unique_ptr：独占所有权，只有一个unique_ptr指向一个对象，当unique_ptr销毁时，对象也会被销毁
- weak_ptr：弱引用，通常与shared_ptr搭配使用，不会增加对象的引用计数，当最后一个shared_ptr销毁时，对象也会被销毁

```
#include <iostream>
#include <memory>
```

```
class Object {
public:
    Object(int value): value(value) {
        std::cout<<"构造函数\n";
    }
    ~Object() {
        std::cout<<"析构函数\n";
    }

    int value;
};

int main() {
    std::shared_ptr<Object> p1 = std::make_shared<Object>(1);
    // 输出 构造函数
    std::shared_ptr<Object> p2 = p1;
    std::cout<<p1->value<<" "<<p2->value<<"\n"; // 输出1 1
    p2->value = 2;
    std::cout<<p1->value<<" "<<p2->value<<"\n"; // 输出2 2
    std::cout<<p1.use_count()<<"\n"; // 输出2，表示有两个shared_ptr指向Object
    p1.reset(); // p1不再指向Object，但Object不会被销毁
    std::cout<<p1.use_count()<<" "<<p2.use_count()<<"\n"; // 输出0 1
    p1 == nullptr; // true
    std::weak_ptr<Object> p3 = p2;
    std::cout<<p2.use_count()<<"\n"; // 输出1, weak_ptr不会增加引用计数
    // 程序退出
    // 输出 析构函数
}
```

unique_ptr比较特殊，它不允许拷贝，也就是说不能同时有多个unique_ptr指向一个对象，但可以移动，也就是说可以将一个unique_ptr指向的对象转移到另一个unique_ptr，移动语义这里不做过多介绍，有兴趣自己去了解

4. 常用的一些C++新特性

4.1 auto关键字

auto可以自动推导变量的类型，例如

```
auto a = 1; // a是int类型
double b = 1.0;
auto c = b; // c是double类型

std::vector<int> v = {1,2,3,4,5};
auto it = v.begin(); // it是vector<int>::iterator类型
```

合理的使用auto能节省大量的时间，例如你想遍历一个vector，你只需要这么写

```
std::vector<int> v = {1,2,3,4,5};
for (auto it = v.begin(); it != v.end(); it++) {
    std::cout<<*it<<" ";
}
```

4.2 基于范围的for循环

基于范围的for循环是C++11引入的新特性，它可以遍历数组和容器，例如

```
std::vector<int> v = {1,2,3,4,5};
for (int i : v) {
    std::cout<<i<<" ";
}
// 输出 1 2 3 4 5
```

如果你自己实现的类想要支持基于范围的for循环，你需要实现一下你的类的迭代器，这里不做过多介绍，有兴趣自己去了解

4.3 constexpr关键字

constexpr关键字修饰的对象和表达式表示，这个表达式能在编译期就能得到计算结果，在用来提高性能和检查错误时会有帮助。例如，我们可以使用constexpr代替#define定义常量

```
#define PI 3.1415926 //传统的宏定义
constexpr double pi = 3.1415926; // 使用constexpr定义常量
```

这样做的区别是，pi是一个真正的变量，而PI只是一个文本替换，编译器不会检查PI的值是否正确，例如一个函数的参数是double类型，你传入PI，编译器不会报错，但如果你传入pi，编译器会报错。

4.4 lambda表达式

lambda表达式是C++11引入的一个新特性，它可以用来创建匿名函数（有的语言叫做闭包），例如

```
auto func = [](int a, int b) -> int {
    return a+b;
};
// func可以像函数一样调用

std::cout<<func(1,2)<<"\n"; // 输出3
```

lambda表达式的基本语法是

```
[捕获列表] (参数) -> 返回类型 { 函数体 };
```

- 捕获列表：捕获外部变量

```
int a = 1;
auto func = [a](int b) -> int {
```

```
    // a是从外部捕获的变量，可以在函数体中使用
    // a = 1
    return a+b;
};
std::cout<<func(2)<<"\n"; // 输出3

// 除了变量名，还可以使用&和=，表示按引用和按值捕获
auto func1 = [&a](int b) -> int {
    a = 2; //因为是按引用捕获，所以可以修改a的值
    return a+b;
};
// 如果不指定捕获列表，默认捕获外部所有变量
auto func2 = [&](int b) -> int {
    return a+b;
};
```

- 参数：函数的参数
- 返回类型：函数的返回类型
- 函数体：函数的实现

lambda表达式可以配合STL的算法，实现很酷的功能，例如

```
std::vector<int> v = {1,2,3,4,5};
// 将v中的每个元素都加1
std::for_each(v.begin(), v.end(), [](int &i) {
    i++;
});
// 按照自己定的规则排序
std::sort(v.begin(), v.end(), [](int a, int b) {
    // 排序后，所有元素都满足v[i] > v[i+1]，也就是按lambda表达式返回为true的顺序排序
    return a > b;
});
std::erase(v.begin(), v.end(), [](int i) {
    // 删除所有奇数
    return i % 2 == 1;
});
```

4.5 多线程编程

这里内容太多啦，就不在这里介绍了，有兴趣的同学可以自己去了解一下

```
#include <thread>
#include <mutex>

std::mutex mtx; // 互斥锁，用于保护共享资源
int count = 0;

void consumer() {
    while (true) {
        mtx.lock(); // 加锁
```

```
        if (count > 0) {
            std::cout<<"消费者消费了一个产品\n";
            count--;
        }
        mtx.unlock(); // 解锁
    }
}

void producer() {
    while (true) {
        mtx.lock();
        std::cout<<"生产者生产了一个产品\n";
        count++;
        mtx.unlock();
    }
}

int main() {
    std::thread t1(consumer);
    std::thread t2(producer);
    t1.join();
    t2.join();
    // 两个线程会同时运行
}
```

作业

2023.10.20之前提交到gitee仓库自己的分支下

实现一个自己的动态数组，类名叫Vector，为模板类（要求大家自己去学习），要求实现以下功能

- 支持内置的数据类型，例如int，double，char等
- 动态分配内存，可以自动扩容
- 实现push_back(T data)函数，将data添加到数组末尾，数组满了自动扩容
- 实现pop_back()函数，删除数组末尾的元素
- 实现size()函数，返回数组的大小（数组的大小与你内存分配大小不一定相同）
- 实现operator[]函数，可以像数组一样访问数组元素
- 实现empty()函数，判断数组是否为空

使用下面这段代码作为main函数测试你的代码，并将结果输出截图一并提交

```
#include <iostream>
#include <chrono>
#include <vector>
#include <functional>

#include "Vector.hpp"

constexpr int numIterations = 1000000;
```

```
void runBenchmark(const std::string& operation, const
std::function<void(Vector<int>&)>& func)
{
    std::cout << "Running benchmark for " << operation << "..." << std::endl;

    Vector<int> vec;
    auto startTime = std::chrono::high_resolution_clock::now();

    for (int i = 0; i < numIterations; ++i) {
        func(vec);
    }

    auto endTime = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(endTime -
startTime).count();

    std::cout << "Elapsed time: " << duration << " milliseconds" << std::endl;
}

int main()
{
    // Check accuracy
    Vector<int> vec;
    std::vector<int> stdVec;
    for (int i = 0; i < numIterations; i++) {
        vec.push_back(i);
        stdVec.push_back(i);
    }
    for (int i = 0; i < numIterations; i++) {
        if (vec[i] != stdVec[i]) {
            std::cout << "Error: vec[" << i << "] != stdVec[" << i << "]" <<
std::endl;
            return 1;
        }
    }


    // Benchmark insertions
    runBenchmark("insertions", [](Vector<int>& vec) {
        vec.push_back(42);
    });

    // Benchmark deletions
    runBenchmark("deletions", [](Vector<int>& vec) {
        if (!vec.empty()) {
            vec.pop_back();
        }
    });

    // Benchmark access
    runBenchmark("access", [](Vector<int>& vec) {
```

```
        if (!vec.empty()) {
            int value = vec[vec.size() / 2];
            ++value;
        }
    });

    return 0;
}
```