目录

[一、 Trivial details 2](#_Toc27963)

[1.1初始化列表的使用 2](#_Toc26483)

[1.2 两类赋值的返回值 2](#_Toc7955)

[1.3 Vector.cpp函数定义 3](#_Toc8262)

[1.4 delete的使用 3](#_Toc5032)

[1.4.1 Delete的意义 3](#_Toc18178)

[1.4.2 Delete的检测 3](#_Toc27933)

[1.4.3 赋值的检测 3](#_Toc16451)

[1.4 move与类型转换 4](#_Toc9374)

[1.4.1 move的意义 4](#_Toc5010)

[1.4.2 例子 4](#_Toc4699)

[1.4.3 进一步讨论 4](#_Toc5124)

[二、 置空性讨论 5](#_Toc12843)

[2.0 析构置空 5](#_Toc22216)

[2.1 移动置空 5](#_Toc29290)

[2.1.0每次移动时，我们将B完全置空但是不delete 5](#_Toc12036)

[2.1.1 移动和拷贝的语义区别 5](#_Toc1763)

[2.1.2 移动和拷贝的原理区别 5](#_Toc25011)

[2.1.3 移动过程中B指针置空 5](#_Toc18351)

[2.1.4 移动过程中B完全置空 6](#_Toc6500)

[2.2赋值置空 6](#_Toc9337)

[2.2.0 Def每次赋值时，我们需要通过delete将A置空 6](#_Toc8461)

[2.2.1拷贝赋值 6](#_Toc13965)

[2.2.2移动赋值 6](#_Toc26184)

[三、 题意分析 6](#_Toc4240)

[3.1 oop的思想 6](#_Toc1725)

[3.2 Node的函数意义 7](#_Toc8411)

[3.3题意 7](#_Toc2431)

[四、技术细节 7](#_Toc25550)

[4.1拷贝与移动的匹配性 8](#_Toc11425)

[4.2 原对象完全置空 8](#_Toc19861)

[4.3赋值置空 9](#_Toc12386)

[4.4 std::move 9](#_Toc20274)

[4.5建立内存指针和移动的思想 10](#_Toc5259)

oop第三次作业第三题

By zhaochen20

1. Trivial details

没有太多思考价值，故而建议直接阅读

1.1初始化列表的使用

Vector::Vector(Vector&& other):array(other.array),capacity(other.capacity),len(other.len)

{other.array = nullptr;}

1.1.1初始化列表需要加冒号

1.1.2只有构造函数可以用初始化列表，中间有一系列的赋值运算符当然不能用初始化列表

1.2 两类赋值的返回值

· 所有成员函数的参数中，隐含着一个指向当前对象的变量，其名称为this。

· 比如下面这段代码（代码可能有误）

Vector&Vector::operator=(Vector&& other)

{

    if (this != &other) {

        if (this->array != nullptr)

        {

            delete[]array;

        }

        this->array = other.array;

        this->capacity = other.capacity;

        this->len = other.len;

        for (int i = 0; i < len; i++)

        {

            other.array[i] = 0;

        }

        0ther.array = nullptr;

        other.len = 0;}

    return \*this;

}

· 这段代码，除了return部分的this，别的都没有实际意义。

· 而return这一段，注意到此处的返回值是一个Vector的左值引用，这当然是有意义的。移动赋值运算符的返回值一般约定俗成为Vector&。实际上，仅仅赋值而言，你的返回值甚至可以是void，并不需要规定返回值为Vector的引用。的程序在读取A=B时，实际上执行了A.operator(B)。假设你的返回值是void，你不需要写return，也就对A完成了所有的赋值操作。规定为Vector&是为了进行更加复杂的操作。比如我们同时去重载了++，那么(A=B)++，就会先对A进行赋值，接下来返回\*this（指针解引用，返回指针指向地址的内容），返回的内容（也就是A本身）马上被引用（也就是Vector&），A的引用又去执行了++。整套操作就完成了对A先赋值，后++。

1.3 Vector.cpp函数定义

· 函数定义的写法是：返回值类型+命名空间：：函数名+参数。比如：

Node& Vector::operator [] (int pos)//正确

Vector::Node& operator [] (int pos)//错误

1.4 delete的使用

1.4.1 Delete的意义

       delete[]array;//delete：释放指针指向的内存区

· 故而delete之后，内存区的数据就清除了。

· delete释放的是指针指向的内存空间，指针变量本身仍然存在可以使用（用于赋值etc）。但是调用array[i]会产生运行时错误segmentation fault，因为它没有指向一块内存空间用于存储数据。

· 清楚不等同于设为0。而且实际上，在这道题里面，如果delete之后还把array[i]赋值为0，不仅没有意义（因为内存已经被清除了），而且会因为调用了多次Node的拷贝构造函数而浪费了效率。比如下面这个写法纯属浪费效率，还有错误。

Vector&Vector::operator=(const Vector& other)

{ if (this != &other) {

        if (this->array != nullptr)

        {delete[]array;}

        for (int i = 0; i < capacity; i++)

        {array[i] = 0;}......}

1.4.2 Delete的检测

建议在delete前检测指针是否为空指针，因为空指针delete会报错。

Vector::~Vector()

{if (array != nullptr) { delete[]array; }}

1.4.3 赋值的检测

和delete相似，建议赋值也要先进行检测，避免浪费效率

Vector&Vector::operator=(const Vector& other)

{    if (this != &other)

        //建议赋值的时候检测是否相同，检测的原理本质上就是检测地址是否相同

        if (this->array != nullptr)

        { delete[]array; }......

}

1.4 move与类型转换

1.4.1 move的意义

move的作用就是仅仅把左右值这个属性改为右值，其余的属性不变。

1.4.2 例子

Test F(const Test& a){

Test b = std::move(a);

return b;}

int main(){

Test A = F(1);

return 0;}

这个地方极其有趣的是F函数体内对b的构造是采用了拷贝构造而非移动构造。

这是由于move和const的双重作用。如同前文所述，move一定可以把任意对象调整为右值，不管是不是const。

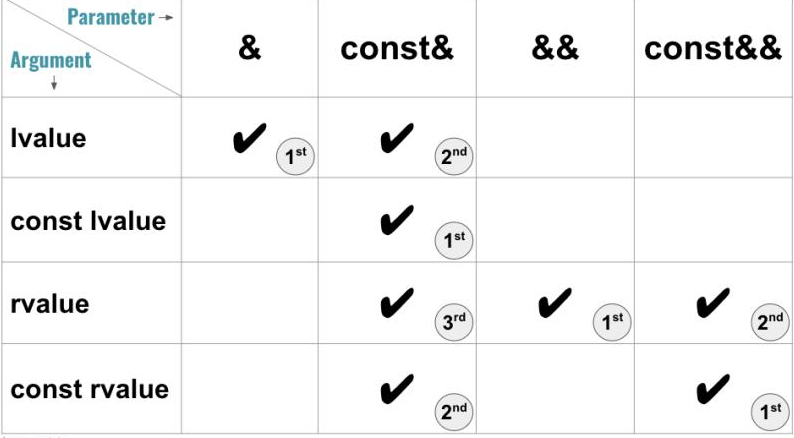
深层次地说，变量储存在内存里的时候存的都是它的数据，而不会专门开辟一块空间来说明它是不是const，是不是&等，const和&这些“类型”是由编译器来处理的。

我们说类型转换，指的就是改变类型，而不变动它在内存里的储存形式。比如一个指针，它在内存里存的就是一串数字表示地址，我们看它的内存情况时，完全可以把这块内存当做存的是一个整型变量。我们强行把指针当做整型变量来看的时候就是进行了一个类型转换。

所以std::move(a)这个表达式的类型就是const&&，因为它发生了类型转换。

而转换之后，a转为const&&，这是一个常量右值引用，C++固定其只能够被const&绑定。故而在构造函数重载的情况下调用了相应const&的拷贝构造函数。

1.4.3 进一步讨论



这张表给出了关于绑定和左右值的关系。左边的一列是参数的属性，Lvalue是指左值，const是指常量。上方一行给出了能够绑定的类型。而表中的次序代表多种绑定时的优先级。从而可见，常量右值优先在我们的两个构造函数当中只能绑定const&。（因为没有const&&类型的重载，故而绑定的是）从而调用了拷贝构造函数。

1. 置空性讨论

2.0 析构置空

析构函数需要将成员数据中的指针指向的内存空间置空（delete）

Vector::~Vector()

{if (array != nullptr) { delete[]array; }}

下文以A=B为例，讨论赋值过程中的置空性

2.1 移动置空

2.1.0每次移动时，我们将B完全置空但是不delete

2.1.1 移动和拷贝的语义区别

移动和拷贝的语义区别在于，被拷贝的对象之后还可能会被使用的，而被移动的对象必须置空（但不是delete，delete出现在赋值置空和析构置空当中）。因为移动的本质原理是把对同一块内存空间更换为新指针，并且将原指针置空。

2.1.2 移动和拷贝的原理区别

2.1.2.1拷贝的原理

A与B各自有一个指针，如果A！=B时，两个指针指向两块独立的内存空间。（A=B时无需拷贝）现将A内存空间里所有数据清除（为什么要清除在2.2.1进行了解释），接着把B的内存空间内所有数据传递给A的内存空间。这一传递过程不会破坏B的数据，但是效率较低。

2.2.2.2移动的原理

A与B各自有一个指针。如果A！=B时，两个指针指向两块独立的内存空间。（A=B时无需移动）如果此时A的指针指向的内存空间不为空，也就是A指针不为空指针，那么将A的指针指向的内存空间通过delete清除，A指针即转为空指针。将B的指针赋值给A的指针，并将B指针置为空指针。这一过程实际上是改变了内存空间的指针但是没有改变内存空间。

2.1.3 移动过程中B指针置空

在析构函数里，我们会调用delete。

Vector::~Vector()

{if (array != nullptr) { delete[]array; }}

倘若不把B指针置空，那么我们会析构delete一次B指针指向的内存空间，然而A指针不为空。在析构时，又将对同一个内存空间delete一次。Delete一空的空间是不允许的。故而移动构造必然要将B指针置空以避免多次delete同一空间。这样也导致了B指针不能再次使用（整个B无法再去赋值）。

同样的，为了模拟这种对B指针的破坏性，在第三部分技术细节里提及了Node的移动构造机制。这一机制也将“B指针”（实际上是原来的int）置空，故而被移动了的Node也是没法再次利用的。

2.1.4 移动过程中B完全置空

我们在上文已经叙述了为什么需要将B指针置空。实际上，由于B对象被移动之后已经不再次使用，需要将其完全置空（所有的成员数据都要置空），从而避免内存泄漏。（这里点到为止，具体的细节参见第三部分）

2.2赋值置空

2.2.0 Def每次赋值时，我们需要通过delete将A置空

2.2.1拷贝赋值

拷贝赋值时，检测完A==B后，我们需要使用delete将A置空。

这里为什么要先清除A的原内存空间？

首先，我们的移动赋值需要二者赋值完后完全相同，如果A原内存空间超出B的那部分容积有多于内容，不清除将A则无法使得A与B完全相同。其次，不清除更大的问题是内存泄露严重。

注意到拷贝赋值现将A的内存空间delete之后，A的指针需要指向一块内存空间才能使A接受B的拷贝赋值。故需要申请新的内存空间。（与B的内存空间大小完全相同）

2.2.2移动赋值

综合前文所述，移动需要将B完全置空但是不delete，而赋值需要将A通过delete置空之后再次申请新的空间。所以移动赋值需要两次置空，一次delete而另一次不delete。

1. 题意分析

3.1 oop的思想

oop的核心思想就是你只需要写好自己处理的程序，完善接口和函数，然后坐等别人来调用，故而本题理想状态下是不需要读太多其他部分的，直接看懂Vector.h然后写Vector.cpp。（碎碎念：这个Vector他妈的是大写的！）

然而，实际上由于这道题的评判标准，还是需要理清楚Node的意义。这里简要说明下Node的函数意义。

3.2 Node的函数意义

class Node

{

private:

    static long long int num\_createfunc\_default, num\_createfunc, num\_copyfunc, num\_movefunc, num\_copyassign, num\_moveassign, num\_delfunc;

    int val;//只有value是我们要用到的，别的都不需要

public:

    Node(int v);

    Node();

    ~Node();

    Node(const Node &y);//node的拷贝构造函数。结合题意，Vector的拷贝构造需要保留原来的Node，故而要用Node的拷贝构造

    Node(Node &&y);//node的移动构造。查看下Node.cpp会发现，对node而言，因为node内部没有数组，copyfunc和movefucn没有显著效率区别；

//但是出题人为了强调一般拷贝比移动效率低，所以强行把拷贝的cost设为10，move设为1；

//故而尽量多用移动而少用拷贝

    Node& operator=(const Node &y);

    Node& operator=(Node &&y);

    bool operator!=(const Node &y);

    friend std::istream& operator>>(std::istream& in, Node& x);

    friend std::ostream& operator<<(std::ostream& out, const Node& x);

    static void outputResult(int ref\_ans);

};

3.3题意

很直白，先利用已经提供的Node的两种构造，两种赋值完成Vector的两种构造，两种赋值。接下来完成几个需要实现的功能即可。

四、技术细节

涉及大量源码，高度剧透，建议先自己写一写，之后再阅读，以此加深印象。

并且，请妥善利用，避免查重。

4.1拷贝与移动的匹配性

Vector::Vector(const Vector& other)//拷贝构造函数如何拷贝一个指针，老实写for循环即可

{ ....

    array = new Node[other.capacity];//注意到这里的大小是capacity而非len。array的大小实际上是容量，而len是目前储存的量。关键是，你在赋值容器的时候不可能减少容积啊，所以这里的参数是capcity而非len。

    for (int i = 0; i < len; i++)

    {array[i] = other.array[i];}

}//拷贝了之后是否还需要原来的vector？当然需要！如果我不需要的话我就会用移动了。

注意到我的for循环里面的赋值操作。查看下Node的移动赋值与拷贝赋值，我们会发现二者实现的原理非常相近，单独观察这一段Vector的拷贝赋值，哪怕我用移动赋值清除了Node的数据，但是并没有影响到我给这个Vector赋值的正确性。

Node& Node::operator=(const Node &y) {

    val = y.val;

    num\_copyassign++;

    return \*this;

}

Node& Node::operator=(Node &&y) {

    val = y.val;

    y.val = 0;

    num\_moveassign++;

    return \*this;

}//对node的拷贝构造赋值与移动赋值没有本质的效率区别

那么，这里为什么不用移动赋值而用了拷贝赋值？

实际上，单独看这一段当然没有问题。

移动和拷贝的语义区别在于，被拷贝的对象之后还可能会被使用的。助教应该是注意到了这一点，因此被拷贝的Node被人为地多次使用。我第一次利用Node的移动赋值实现了Vector的拷贝赋值之后，被拷贝的Node的val就被设为0了。之后再次调用这个Vector来进行拷贝复制，就会出问题。所以说，Vector的移动应该使用Node的移动，而Vector的拷贝则调用Node的拷贝。

4.2 原对象完全置空

Vector::Vector(Vector&& other):array(other.array),capacity(other.capacity),len(other.len){

other.array = nullptr;

    other.len = 0;}

如1.5.3所述，这里不仅将原指针置空，还要将len置空。（所有成员数据都要置空）

4.3赋值置空

Vector&Vector::operator=(const Vector& other){

    if (this != &other)

              if (this->array != nullptr)

        {

            delete[]array;

        }

    this->capacity = other.capacity;

    this->len = other.len;

    array = new Node[capacity];//之前的内存区域已经清除了，这里必须重新申请

        for (int i = 0; i < len; i++)//这里是len还是capacity貌似没有影响

        {

            this->array[i] = other.array[i];

        }

        return \*this;

}

给出了拷贝赋值的代码，注意其中的置空使用。

Vector&Vector::operator=(Vector&& other)

{

    if (this != &other) {

        if (this->array != nullptr)//this是vector的指针，我要判断的是array是否为空

        {

            delete[]array;//delete的意义就是释放指针指向的内存区域

        }

        this->array = other.array;

        this->capacity = other.capacity;

        this->len = other.len;

        other.array = nullptr;

        other.len = 0;//系统会检查是否清0；

    }

    return \*this;

}

给出了移动赋值的代码，注意其中两次置空，一次通过delete，一次仅仅置为空指针而不delete。

4.4 std::move

综合前文所述，我们在原Node无需使用的情况下，应该尽量使用移动而非拷贝。故而在insert和swap里面应该尽量使用移动。然而移动只能对右值使用，故而必须要用std::move

void Vector::insert(int pos, int value)

{

    for (int i = len; i>pos; i--)/

        //注意到这里虽然用左值的拷贝赋值也没有问题，但是题目强调了拷贝和移动的效率区别，故而应该使用移动赋值

        //而移动赋值只能对右值进行，所以需要把左值手动转化为右值，也就是用std::move函数

    {

        array[i] = std::move(array[i - 1]);//在这一段的赋值逻辑，应该从从后往前覆盖，不然前面的数据会被抹除

    }

    array[pos] = value;

    len++;

    return;

}

void Vector::swap(int pos1, int pos2)//善用move,这里用move同样可以把左值转为右值

{

    Node tmp = std::move(array[pos1]);

    array[pos1] = std::move(array[pos2]);

    array[pos2] = std::move(tmp);

}

4.5建立内存指针和移动的思想

void Vector:: dilatation()

{

    this->capacity \*= 2;

    Node\* tmp = new Node[this->capacity];

    for (int i = 0; i < len; i++)

    {

        tmp[i] = std::move(array[i]);

    }

    if (array != nullptr) { delete[]array; }//把array这个地址指向的内存给释放了，但是array还存着那个地址，只不过不能用了也不应该再用了

    this->array = tmp;//我现在把array指向的地址改为tmp的地址

}

如何理解这一过程：

首先，this的容量加倍。

1. 添加临时指针，申请出一块两倍的内存空间。
2. 第三，把原内存空间的每个数据移动赋值到新内存空间上。（原内存空间必然不再使用，故而采用移动而非拷贝）
3. 将原内存空间置空。
4. 将this的指针指向tmp。

最后一个问题，为什么最终没有把tmp置空？

表面上，临时指针会在函数结束后自动变为0指针但是不会delete掉临时指针指向的内存。

delete 之后，指针仍然保存原先的地址，并不会自动转变为nullptr；但是这个地址对应的内存空间已被释放，如果试图访问会产生异常。

深层的原因是，tmp本身就是这个函数里的局部变量，它自己在栈区，new出来的内存在堆区，然后让array（在堆区）指向那部分内存，tmp就没用了，函数结束后自动出函数栈，以后都用不到了。