- (32条消息) 西电面向对象程序设计核心考点汇总（期末真题） oax_knud的博客-CSDN博客
- 看完题不要忽略题
-

声明引用返回值

- C++标识符通常是用来表示变量、函数、类等名称的字符序列。在C++中，标识符的第一个字符必须是字母、下划线或者字符 '$'，而不能是数字。这是因为数字开头的标识符在C++中被用来表示数字常量，编译器会将其解释为数字而不是标识符，从而导致出错。
-
```
//数据类型后加&，相当于用引用的方式返回
int& test02() {
  // 必须使用静态变量,需加 static 关键字
static int a = 20;
return a;
}
//使用 static 关键字来修饰变量 a，使其在函数执行结束之后仍然存在
```

- 在类外定义虚函数时，不必再加 virtual 关键字，编译器会自动将其识别为虚函数，并将其加入到虚函数表中。
- b.num=a.num++。***（特别注意这种表达式赋值和自增顺序，++在变量右时，先赋值，后自增；++在变量左时，先自增后赋值）***所以b.num=1, a.num自增后为2（不能忘记）。输出结果为21。
- 在 C++ 中，对于通过 `new` 运算符生成的对象，正确的说法是：

  B. 执行 `delete` 操作时才能析构

  当你使用 `new` 运算符创建一个对象时，它会在堆（而非栈）上为该对象分配内存。堆上的内存不会在函数返回时自动释放，也不会在程序结束时自动释放。你需要显式地使用 `delete` 运算符来释放这些内存并调用对象的析构函数。如果你忘记这样做，就会导致内存泄漏，这是一种常见的编程错误。

  因此，选项A, C, D都是错误的。A和C都错误地声明在某些条件下会自动释放 `new` 分配的内存，而D错误地声明在程序结束时会自动释放未释放的 `new` 分配的内存。

  所以，在使用 `new` 运算符分配的内存，你必须始终确保也使用 `delete` 运算符来释放内存，以防止内存泄漏。同时，这也会调用对象的析构函数，执行任何必要的清理工作。
- 这个错误出现在尝试使用命名空间的一个成员（而非整个命名空间）。使用 `using` 关键字可以导入整个命名空间，但不能导入命名空间的单个成员。正确的做法是直接使用全名。另一个问题是匿名命名空间不能被其他文件访问，因此使用 `using namespace myspace` 不会导入任何内容。正确的使用方法是：

  ```
  namespace myspace {
    void doSomething() { /* ..... */ }
    int temp;
  }
  using myspace::doSomething;
  using myspace::temp;
  temp = 1;//只能这么声明
  ```

是的，这个程序有错误。在 C++ 中，`using` 语句只是将命名空间中的名称引入到当前的命名空间，而不是为这个名称赋值。所以 `using mySpace2::x = 1;` 是无效的。如果你想要为 `mySpace2::x` 赋值，你需要单独进行，就像下面这样：

构造函数

- 如果用户定义有参构造函数，c++不再提供默认无参构造，但是会提供默认拷贝构造函数。对于拷贝构造函数，情况不同。由于拷贝构造函数的作用是用已有对象来初始化一个新的对象

```
class MyClass {
public:
    MyClass(const MyClass& other);  // 拷贝构造函数
};

```

- 除了const成员变量外，还有一些类型的成员变量也必须使用成员初始化列表进行初始化，否则会导致编译或运行时错误。这些类型包括：

1. 引用类型（Reference Type）：引用类型的成员变量必须在构造函数的成员初始化列表中进行初始化。
2. 常量成员对象（constant member object）：常量成员对象和const成员变量类似，也必须在构造函数的成员初始化列表中进行初始化。
3. 类类型的成员变量（Class Type Member Variables）：如果类类型的成员变量没有提供默认构造函数，那么它们也必须在构造函数的成员初始化列表中进行初始化。

例如，假设存在如下类定义：

```
class MyClass {
public:
 int& numRef;
 const int numConst;
 std::string str;
 MyClass(int n): numRef(n), numConst(1), str("Hello") { }
};
```

在上述例子中，numRef是一个int类型的引用成员变量，numConst是一个const int类型的常量成员对象，str是一个std::string类型的成员变量。在MyClass的构造函数中，我们必须使用成员初始化列表对它们进行初始化。

总之，在类中使用const、引用、常量成员对象、类类型的成员变量时，需要注意必须使用成员初始化列表进行初始化。

- 在C++中，关于类的构造函数的描述，下面是每个选项的解析：

  A. 类的构造函数可以重载 - 正确。构造函数可以有多个，并且可以根据传入的参数类型和数量不同进行重载。

  B. 类可以没有构造函数 - 错误。每个类都至少有一个构造函数。如果程序员没有明确地定义一个构造函数，那么编译器会自动为该类生成一个默认的无参构造函数。

  C. 类的构造函数可以缺省 - 正确。构造函数可以是默认的（无参），也可以带有参数。如果未定义构造函数，C++编译器将自动提供一个默认的无参构造函数。

  D. 类的构造函数可以作为其它类型向本类类型进行转换的函数 - 正确。这种构造函数被称为转换构造函数，它接受一个参数，并用这个参数的类型将其他类型转换为类的类型。例如，如果你有一个接受整型参数的构造函数，你可以将一个整型变量转换为你的类的实例。

因此，选项B“类可以没有构造函数”是错误的。

- class_name (const class_name &old_obj);

  关于拷贝构造函数，下面是你所问问题的答案：

  1. 调用拷贝构造函数的方式：
     - 当一个对象作为值传递给函数时，会调用拷贝构造函数。
     - 当一个函数返回对象时，也会调用拷贝构造函数。
     - 当使用一个对象初始化另一个对象时，会调用拷贝构造函数。
     - 当你创建一个对象的数组并初始化它时，如果该对象有可用的拷贝构造函数，会调用它。
  2. 深拷贝与浅拷贝：
     - 浅拷贝：默认的拷贝构造函数实现的是浅拷贝，也就是仅仅复制对象的值。当对象中包含指针时，这种复制方式会导致新旧对象共享同一片内存空间，这就会引发问题。例如，当其中一个对象被销毁时，另一个对象的指针就会变为悬空指针。
     - 深拷贝：对于含有动态分配内存的类，我们需要重写拷贝构造函数，执行深拷贝，即不仅复制对象的值，还会复制它引用的动态内存。这样，新旧对象就不会共享同一片内存空间，销毁其中一个对象时也不会影响另一个对象。

  例如：

```
class Deep {
   int *data;
public:
   Deep(int d) { data = new int(d); }
   // 深拷贝
   Deep(const Deep &source) {
      data = new int(*(source.data));
   }
   ~Deep() { delete data; }
};

class Shallow {
   int *data;
public:
   Shallow(int d) { data = new int(d); }
   // 浅拷贝（默认的拷贝构造函数）
   Shallow(const Shallow &source) = default;
   ~Shallow() { delete data; }
};
```

在上述例子中，如果我们复制一个 `Shallow` 对象并删除原始对象，复制的对象将包含一个指向已删除内存的悬空指针。相反，如果我们复制一个 `Deep` 对象并删除原始对象，复制的对象将包含一个指向其自己内存的指针，这块内存的内容是从原始对象复制过来的，所以它是安全的。

-
```
/**
   * Derived1 derived1_obj；在堆栈上创建一个Derived1类型的对象，
   * 并使用默认构造函数对其进行初始化。这是创建没有参数化构造函数的类的对象的正确语法。
   */
  Derived1 derived1_obj;
```

数组指针

- 数组名 arr 相当于指向数组第一个元素的指针，即 arr 等价于 &arr[0]

> 在 C/C++ 中，数组指针和指针数组都涉及到数组和指针的概念，但含义不同。下面分别介绍一下它们的定义和区别。
>
> 1. 数组指针
>
> 数组指针是指针类型，它指向一个数组，可以通过指针访问该数组中的元素。数组指针的定义方式为：`type (*p)[n]`，其中 type 表示所指向的数组中元素的类型，`p` 是一个指向数组的指针，`n` 表示数组的长度。
>
> 举个例子，如果要声明一个指向整型数组的指针，可以这样写：
>
> ```
> int arr[3] = {1, 2, 3};
> int (*p)[3] = &arr; // 定义一个指向长度为 3 的整型数组的指针 p，将其赋值为数组 arr 的地址
> ```
>
> 使用 `(*p)[i]` 或 `*(*(p+i)+j)`（与二级指针类似）的方式，可以访问数组 arr 中的元素。
>
> 2. 指针数组
>
> 指针数组是一种数组类型，它的每个元素都是一个指针，可以用来存储多个指针值。指针数组的定义方式为：`type *p[n]`，其中 type 表示指针指向的数据类型，`p` 是一个包含 n 个指针变量的数组。
>
> 举个例子，如果要声明一个指针数组，可以这样写：
>
> ```
> int a = 1, b = 2, c = 3;
> int *p[3] = {&a, &b, &c}; // 定义一个包含三个指针变量的数组 p，每个元素都是一个指向整型变量的指针
> ```
>
> 使用 `p[i]` 的方式，可以访问其中的某个指针变量。
>
> 综上所述，数组指针与指针数组的区别在于：数组指针是一个指向数组的指针类型，可以通过指针访问数组中的元素；而指针数组是一个数组类型，其中每个元素都是一个指针，可以用来存储多个指针值。

- 加了[]和解引用*是一样的效果取的都是内容。

继承

- 在 C++ 中，如果一个类继承了其它类，那么子类在构造时必须先构造其父类的成员变量。

  如果在子类的构造函数中没有显式地调用其父类的构造函数，编译器会自动调用其父类的默认构造函数，从而初始化其父类成员变量。但是如果父类没有默认构造函数，或者默认构造函数不满足要求，那么**编译器就无法为父类成员变量赋初值，即出现编译错误。**

  因此，在子类中定义构造函数时，需要显式地调用父类的构造函数，保证父类中的成员变量能够正确被初始化。可以使用子类初始化列表来完成这个任务。例如：

  ```
  DC(int a) : BC(a) {
      // 子类的构造函数体
  }
  ```

这样就可以在子类的构造函数中调用父类的构造函数，并提供相应的参数来初始化父类成员变量。这样就能够使程序正确执行了。

- 在这个例子中，错误在于 `DC` 类试图调用被自己覆盖掉的基类 `BC` 的 `f` 函数。`DC` 类中的 `f` 函数已经覆盖了 `BC` 中的 `f` 函数，所以在 `DC` 类的对象上调用 `f()` 函数会调用 `DC` 的版本，而不是 `BC` 的版本。这里由于 `DC` 的 `f` 函数需要一个 `int` 参数，而我们并没有提供，所以这个调用是错误的。如果我们希望调用 `BC` 的 `f` 函数，可以在 `DC` 中添加一个无参数的 `f` 函数，或者使用 `BC::f` 来显式调用 `BC` 的版本。

  当子类与父类拥有同名的成员函数，子类会隐藏父类中同名成员函数，加作用域可以访问到父类中同名函数

运算符重载

- 大部分运算符都可以使用非成员函数来进行重载，但以下几种情况需要**使用成员函数来进行重载**：

1. 赋值运算符（=）：赋值运算符必须用成员函数进行重载，并且返回类型应该是指向自身的引用。
2. 下标运算符（[]）：下标运算符必须使用成员函数进行重载，其参数列表中应该有一个整型索引，并且返回类型应该是元素类型的引用。
3. 函数调用运算符（()）：函数调用运算符必须使用成员函数进行重载，并且不允许多次重载。

以上三种运算符是仅能使用成员函数进行重载的情况，因为它们需要操作类的私有成员变量或者将类的对象视作函数或数组使用，只有成员函数才能够访问类的私有成员变量和成员函数。除此之外，其他的运算符都可以使用非成员函数或成员函数进行重载。

- 这个错误出现在试图将一个非成员函数定义为成员函数。在这个例子中，`operator+` 被声明为 `C` 类的一个友元函数，而不是成员函数。友元函数在实现时不需要使用 "类名::"。正确的实现方式是：

```
#include <iostream>
using namespace std;

class complex {
    double re, im;
public:
    complex(double r=0, double i=0) : re(r), im(i) {  }
    complex(const complex& a):re(a.re),im(a.im) {  }
    operator double() const {return re;}
    complex& operator+= (complex a);
    complex operator+ (complex a);
    friend ostream& operator<<(ostream& oo, const complex &a);
};

complex& complex::operator+= (complex a) {
    re += a.re;
    im += a.im;
    return *this;
}

complex complex::operator+ (complex a) {
    return complex(re + a.re, im + a.im);
}

ostream& operator<<(ostream& oo, const complex &a) {
    oo << "(" << a.re << ", " << a.im << ")";
```

```
    return oo;
}
```

```
#include <iostream>
#include <vector>

using namespace std;

class Queue
{
public:
    Queue() {}

    Queue operator--()
    {
        if (data.empty())
        {
            cout << "The queue is empty!" << endl;
            return *this;
        }
        else
        {
            Queue newqueue;
            newqueue.data.push_back(data.front());
            data.erase(data.begin());
            return newqueue;
        }
    }

    Queue operator+(int x)
    {
        if (data.size() < capacity)
        {
            cout<<"no f"<<endl;
            data.push_back(x);
        }
        else
        {
            cout << "The queue is full!" << endl;
        }
        return *this;
    }

    friend Queue operator+(int x, const Queue &q)
    {
        cout<<"fff"<<endl;
        Queue newQueue = q;
        if (newQueue.data.size() < capacity)
        {
            newQueue.data.push_back(x);
        }
        else
        {
            cout << "The queue is full!" << endl;
        }
```

```
        return newQueue;
    }

    friend ostream &operator<<(ostream &os, const Queue &q)
    {
        if (q.data.empty())
        {
            os << "The queue is empty!" << endl;
        }
        else
        {
            os << "Display: ";
            for (int x : q.data)
            {
                os << x << "  ";
            }
            os << endl;
        }
        return os;
    }
    // friend ostream &operator<<(ostream& os,int num){
    //     os<<"Display: "<<num<<endl;
    //     return os;
    // }不能多次重载


private:
    static const int capacity = 3;
    vector<int> data;
};
int main()
{
    Queue q;
    --q; // Display: The queue is empty!
    q = q + 5 + 6 + 3;//no f x 3
    cout << q;   // Display: 5   6   3
    cout << --q; // Display: 5
    cout << q;   // Display: 6   3
    q = 9 + q;
    cout << q; // Display: 6   3   9fff
    q = 3 + q; // Display: The queue is full!
    cout << q; // Display: 6   3   9fff
    return 0;
}
```

- ```
  #include <iostream>
  using namespace std;

  class Integer
  {
  public:
      int value;

  ```
  ```

```
    Integer()
    {
        value = 0;
    }

    Integer(int v)
    {
        value = v;
    }

    Integer(const Integer &other)
    {
        value = other.value;
    }

    Integer &operator=(const Integer &other)
    {
        value = other.value;
        return *this;
    }

    Integer operator+(const Integer &other) const
    {
        return Integer(value + other.value);
    }

    Integer operator-(const Integer &other) const
    {
        return Integer(value - other.value);
    }

    Integer operator*=(const Integer &other)
    {
        value *= other.value;
        return *this;
    }

    Integer operator/(const Integer &other) const
    {
        return Integer(value / other.value);
    }

    friend ostream &operator<<(ostream &out, const Integer &I)
    {
        out << I.value;
        return out;
    }

    friend istream &operator>>(istream &in, Integer &I)
    {
        in >> I.value;
        return in;
    }
};

```

```
int main()
{
    Integer a, b = 10, c(b);
    cout << "a=" << a << endl;
    cout << "b=" << b << endl;
    cout << "c=" << c << endl;
    cin >> c;
    cout << "c=" << c << endl;
    c = b + 90;
    cout << "b=" << b << " c=" << c << endl;
    a = b - 100;
    cout << "a=" << a << " b=" << b << endl;
    c = a / b;
    cout << "a=" << a << " b=" << b << " c=" << c << endl;
    c = b *= a;
    cout << "a=" << a << " b=" << b << " c=" << c << endl;
    return 0;
}

```

-

类型

- string

```
// Return a string representation of the wrapped double value
std::string toString() const {
  return std::to_string(value);
}
// Return the value of this DoubleValue as an int type
int toInt() const {
  return static_cast<int>(value);
}
```

多态

- 在这个例子中，错误在于试图创建一个抽象类的实例。抽象类是包含至少一个纯虚函数的类，不能被直接实例化。如果想要创建这个类的对象，你需要创建一个继承自这个抽象类的具体类（该类实现了所有的纯虚函数）。所以，这段代码中的 `Animal b;` 是错误的，应改为创建一个继承自 `Animal` 的具体类的实例
-

```
#include <iostream>
#include <cmath>
using namespace std;

class Sequence {
protected:
    int number;
public:
    Sequence(int num): number(num) {}
```

```
    virtual void print() const = 0;  // Pure virtual function
    virtual void update() = 0;  // Pure virtual function to update the number
};

class Increment : public Sequence {
public:
    Increment(int num): Sequence(num) {}
    void print() const override {
        cout << number << " ";
    }
    void update() override {
        number++;
    }
};

class Power : public Sequence {
public:
    Power(int num): Sequence(num) {}
    void print() const override {
        cout << number << " ";
    }
    void update() override {
        number *= number;
    }
};

class Decrement : public Sequence {
public:
    Decrement(int num): Sequence(num) {}
    void print() const override {
        cout << number << " ";
    }
    void update() override {
        number--;
    }
};

/*
在C++中，一旦父类中的某个方法被声明为`virtual`，在其子类中重写该方法时，`virtual`关键字是可以省略的。原因是，子类会继承父类的`virtual`状态。因此，对于那些想要重写父类的`virtual`方法的子类来说，添加`virtual`关键字是可选的。

至于`override`关键字，它在C++11及以后的版本中被引入。当你在子类中重写一个虚方法时，可以使用`override`关键字，这将告诉编译器你打算重写一个基类中的方法。如果基类中没有与之匹配的虚方法，编译器会报错。这样可以避免因误拼写或参数列表错误而未能正确重写基类方法的问题。

所以，虽然`override`关键字在语法上是可选的，但是为了编程的安全性，还是推荐在重写虚函数时使用`override`关键字。
*/
int main() {
    Sequence *spi = new Increment(2);
    Sequence *spp = new Power(3);
    Sequence *spd = new Decrement(4);
```

```
    for(int i = 0; i < 3; i++) {
        spi->print();
        spi->update();
        spp->print();
        spp->update();
        spd->print();
        spd->update();
        cout<<endl;
    }
    delete spi;
    delete spp;
    delete spd;
    return 0;
}

```

-

模板

```
#include <vector>
#include <algorithm>
#include <iostream>
template <typename T>
int partition(std::vector<T> &arr, int low, int high)
{
    T pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++)
    {
        if (arr[j] < pivot)
        {
            i++;
            std::swap(arr[i], arr[j]);
        }
    }
    std::swap(arr[i + 1], arr[high]);
    return (i + 1);
}

template <typename T>
void qsort(std::vector<T> &arr, int low, int high)
{
    if (low < high)
    {
        int pi = partition(arr, low, high);

        qsort(arr, low, pi - 1);
        qsort(arr, pi + 1, high);
    }
}

```

```cpp
template <typename T>
void quickSort(std::vector<T> &arr)
{
    qsort(arr, 0, arr.size() - 1);
}
int main(int argc, char const *argv[])
{
    std::vector<int> arr = {10, 7, 8, 9, 1, 5};
    quickSort(arr);
    for (int i = 0; i < arr.size(); i++)
    {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;

    return 0;
}

```

```cpp
#include<bits/stdc++.h>
#include<String>
/**
Please define a class DoubleValue that wraps(包装) a value of primitive type double and satisfies the following requirements:

(1) it has a default constructor which sets the value to 0.0;

(2) it has a constructor with one argument of type double that is wrapped;

(3) by overloading the operator “==”, it can compare this object against another specified DoubleValue object, and return true if and only if both DoubleValue represent the same double value;

(4) it can return a string representation of the wrapped double value;

(5) it can return the value of this DoubleValue as an int type after a narrowing primitive conversion.
 */
#include <string>

class DoubleValue {
private:
    double value;

public:
    // Default constructor
    DoubleValue() : value(0.0) {}

    // Constructor with one argument
    DoubleValue(double val) : value(val) {}

```

```
    // Overloading the operator "=="
    bool operator==(const DoubleValue& other) const {
        return value == other.value;
    }

    // Return a string representation of the wrapped double value
    std::string toString() const {
        return std::to_string(value);
    }

    // Return the value of this DoubleValue as an int type
    int toInt() const {
        return static_cast<int>(value);
    }
};

```

改错

OOP复习

本文适合针对考试内容的复习，旨在快速对考试知识点有大概掌握。本文会以讲解例题的形式呈现。相应题目来源于网络。

Part I There is one error in each code paragraph. Find out the error and write down the error statement on your answer sheet.

第一部分为找错题，10个，每个两分。每题一个错误。

常考的知识点有：

new开辟空间的释放(delete name; delete []name)

```
float * ptr=new float[20];
for (int i=0;i<20;i++)
    ptr[i]=i+2;
delete ptr;//---->delete []ptr;
```

命名空间(namespace)的使用

```
namespace{
    void do(){/* ......  */}
    int temp;
}
using namespace myspace;
using namespace myspace:temp;//---->using namespace myspace::temp;
```

构造函数(constructor)的使用

```
class Student{
//...
public:
    void Student();//---->Student();
    ~Student();
}
```

私有成员(private member)

```
class C{
    int x;//默认为private
    void setx(int a){
        /* .... */
}
};
void main(){
    C c1;
    c1.setx(3);//类外不能使用private成员
}
```

```
class BC{
    int x;
public:
    BC(int xx=0){x=xx;}
};
class DC:public BC{
    char c;
    DC(int x1,char c1){
        x=x1;//子类不能调用父类的私有成员，若使用protected则可以
        c=c1;
    }
};
```

```
class BC{
protected
    void set_x(int);
};

class DC:public BC{
public:
    void f(){set_x(66);}
};
void f(){
    BC c1;
    c1.set_x(77);//类外不可调用父类protected成员
}
```

运算符重载(operator overloading)

```
class C{
    int sz;
public:
    friend C operator+(const C&, const C&);
    //...
};

C C::operator+(const C& c1, const C& c2){//友元函数（friend function）实现时不需要 "类名::"
    cout<<c1.sz;
    //...
}
```

纯虚函数(pure virtual function)，抽象类（abstract class)

```
class Animal{
public:
    virtual void f()=0;//纯虚函数
};

void f():{
    Animal b;//抽象类不可实例化
}
```

默认参数(default argument)

```
class B{
public:
    B(int a=10, float y)//默认实参从后向前绑定，不能有空隙
    {   i=a;z=y;}
private:
    int i;
    float z;
};
```

成员函数的调用

```
class C{
public:
    void m(){/* ... */}
    static void s(){/* ... */}
};
void main(){
    C c1;
    c1.m();
    c1.s();
    C::s();//成员函数调用方式错误
}
```

const变量

```
int* f(const int y){
    int temp=3;
    temp+=++y;//y为const整形变量，其值不可直接改变。
    return &temp;
}
```

this指针

```
class S{
    //...
    friend int operator[](s,int);//重载成员函数时，this指针默认绑定到左侧运算对象
                                 //---->firend int operator[](int);即可
    int operator!(int);
    //...
};
```

函数重载

```
class BC{
public:
    virtual void f(){/*...*/}
    //...
};
class DC:public BC{
    //...
    void f(int x){/*...*/}
};
void f(){
    DC d;
    d.f();//此处候选函数(candidate function)只有void f(int x)，无可行函数(viabal funtion)
          //DC已经重载f()，此时 BC的f()对 d 不可见
}
```

答题

```
class Book{
    String title;
    vector<String> authors;
    int ISBN;
public:
    Book():{}
    Book(String title, int ISBN):title(title),ISBN(ISBN){}
    void addAurhor(String name){
        authors.push_back(name);
    }
    void setTitle(){

    }
    String getTItle(){

    }
}
```

```
class Integer{
    int value;
public:
    Integer():value(0);
    Integer(int number):value(number){};
    Integer(const Integer& other): value(other.value){}
    Integer operator+(int num){
        return Integer(value+num);
    }
    Integer operator-(Integer another){
        return Integer(value-another.value);
    }
    Integer operator *=(Integer another){
        value*=another.value;
        return *this;
    }
}
```

```
class Sequence{
protected:
    int value;
public:
    int getNumber(){
```

```
        return value;
    }
    virtual void action()=0;
}
class Increment: public Sequence{
public:
    Increment(int val):Sequence(val);
    void action(){
        value++;
    }
}
class Square: public Sequence{
public:
    Square(int val):Sequence(val);
    void action(){
        value= value*value;
    }
}
class Decrement: public Sequence{
public:
    Decrement(int val):Sequence(val);
    void action(){
        value--;
    }
}
```