为了能出现在赋值表达式的左右两边，重载的"[]"运算符应定义为：

- A. A operator [ ] (int);
- B. A& operator [ ] (int);
- C. const A operator [ ] (int);
- D. 以上答案都不对

```
#include <iostream>
#include <cassert>

class IntArray {
private:
    int* data;
    int size;

public:
    // 构造函数
    IntArray(int size) : size(size) {
        data = new int[size];
    }

    // 析构函数
    ~IntArray() {
        delete[] data;
    }

    // 重载[]运算符，返回引用
    int& operator[](int index) {
        assert(index >= 0 && index < size); // 添加基本的边界检查
        return data[index];
    }

```

```
    // const版本的重载[]运算符，用于const对象
    const int& operator[](int index) const {
        assert(index >= 0 && index < size);
        return data[index];
    }
};

int main() {
    IntArray arr(10); // 创建一个长度为10的整数数组

    // 使用重载的[]运算符设置数组元素
    for (int i = 0; i < 10; i++) {
        arr[i] = i * 10;
    }

    // 使用重载的[]运算符获取并打印数组元素
    for (int i = 0; i < 10; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;

    return 0;
}

```

友元函数（仅）实现<<重构

```
#include <iostream>

class Point {
public:
    int x, y;
    Point(int x, int y) : x(x), y(y) {}

    // 用友元函数重载<<运算符
    friend std::ostream& operator<<(std::ostream& os, const Point& pt) {
        os << "(" << pt.x << ", " << pt.y << ")";
        return os;
    }

    // 用友元函数重载>>运算符
    friend std::istream& operator>>(std::istream& is, Point& pt) {
        is >> pt.x >> pt.y;
        return is;
    }
```

```
};

```

前置自增和后置自增

```cpp
#include <iostream>

class Counter {
private:
    int value;

public:
    Counter(int value) : value(value) {}

    // 前置自增
    Counter& operator++() {
        value += 1;
        return *this;
    }

    // 后置自增
    Counter operator++(int) {
        Counter temp = *this;
        value += 1;
        return temp;
    }

    int getValue() const {
        return value;
    }
};

int main() {
    Counter c(5);

    std::cout << "Original: " << c.getValue() << std::endl;
    ++c; // Calls the prefix increment
    std::cout << "After prefix ++: " << c.getValue() << std::endl;
    c++; // Calls the postfix increment
    std::cout << "After postfix ++: " << c.getValue() << std::endl;

    return 0;
}
```

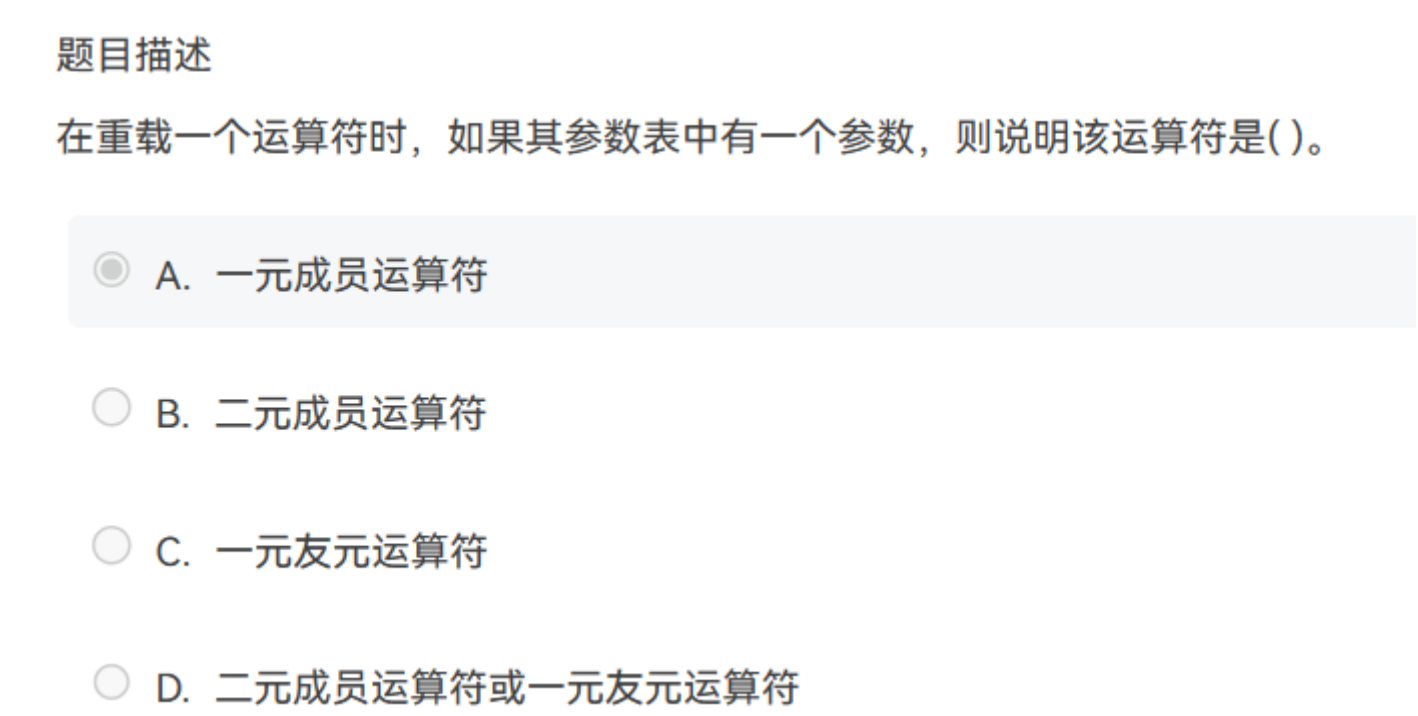
```
39
```

在这个示例中，`operator++()` 实现了前置自增，它直接增加 `value` 并返回对象的引用。而 `operator++(int)` 实现了后置自增，它首先保存当前对象的副本，然后增加 `value`，最后返回原始对象的副本

1. `this` 指针自动可用于所有非静态成员函数中，它指向调用函数的对象。`*this` 是通过 `this` 指针对当前对象进行解引用，因此它代表一个完整的当前对象。
2. 返回自身的引用：
   在类的成员函数中，如果需要返回当前对象的引用，可以使用 `return *this;`。这常见于运算符重载中，如赋值运算符（`operator=`）、前置自增（`++`）和前置自减（`--`）等，以允许链式调用。

成员函数与友元函数

D



掌握什么时候用成员函数，什么时候用友元函数

```
1 #include <iostream>
2
3 class Integer {
4 private:
5     int value;
6
7 public:
```

```
    // 构造函数
    Integer(int val) : value(val) {}

    // 二元成员运算符：operator+，需要一个参数
    Integer operator+(const Integer& other) {
        return Integer(value + other.value);
    }

    // 显示当前值的成员函数
    void display() const {
        std::cout << "Value: " << value << std::endl;
    }

    // 一元友元运算符：operator~，需要一个参数
    friend Integer operator~(const Integer& a) {
        return Integer(~a.value);
    }
};
```

生成类的示例还是指向类的数组

在下面类声明中，关于生成对象不正确的是（ ）。

```
class point
{ public:
int x;
int y;
point(int a,int b) {x=a;y=b;}
};
```

A. point p(10,2);

B. point *p=new point(1,2);

C. point *p=new point[2];

D. point *p[2]={new point(1,2), new point(3,4)};

赋值和拷贝

题目描述

假设A是一个类的名字,下面哪段程序不会用到A的拷贝构造函数?

A. A a1,a2; a1=a2;

B. void func(A a) { cout<<"good"<< endl; }

C. A func() { A tmp; return tmp;}

D. A a1; A a2(a1);

A是赋值构造