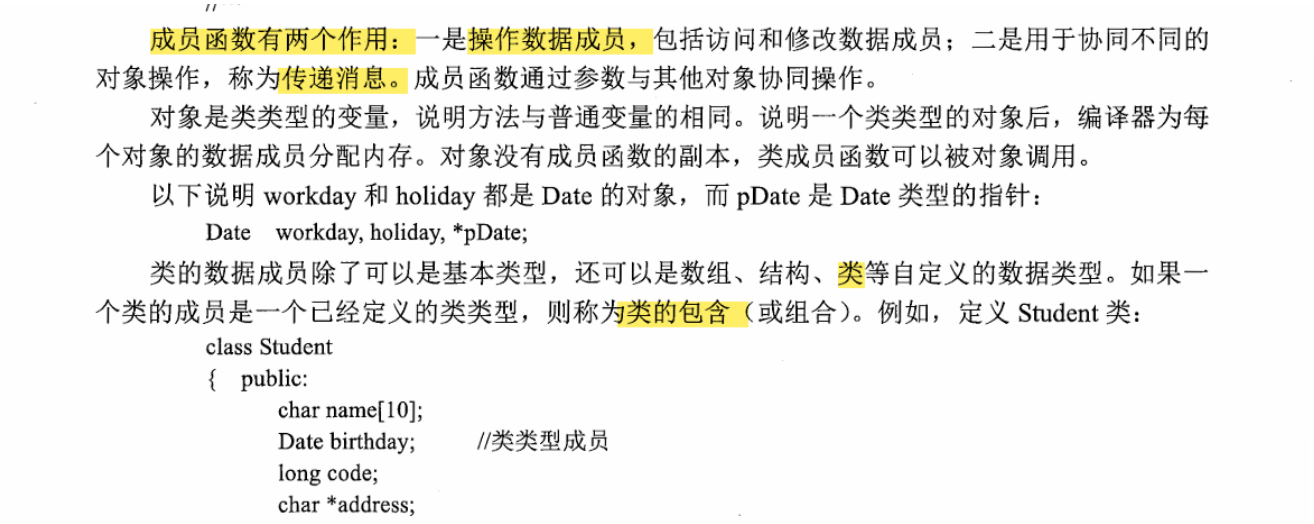
类与对象，运算符重载

类

数据和操作数据的方法的封装

成员函数的作用



成员函数有两个作用：一是操作数据成员，包括访问和修改数据成员；二是用于协同不同的对象操作，称为传递消息。成员函数通过参数与其他对象协同操作。

对象是类类型的变量，说明方法与普通变量的相同。说明一个类类型的对象后，编译器为每个对象的数据成员分配内存。对象没有成员函数的副本，类成员函数可以被对象调用。

以下说明 workday 和 holiday 都是 Date 的对象，而 pDate 是 Date 类型的指针：

```
Date   workday, holiday, *pDate;
```

类的数据成员除了可以是基本类型，还可以是数组、结构、类等自定义的数据类型。如果一个类的成员是一个已经定义的类类型，则称为类的包含（或组合）。例如，定义 Student 类：

```
class Student
{  public:
       char name[10];
       Date birthday;        //类类型成员
       long code;
       char *address;
```

内部

```
class Car {
public:
    void accelerate() { /* 直接定义 */ }
};
```

外部

```
class Car {
public:
void accelerate(); // 声明
};

// 外部定义需要作用域解析符 ::
void Car::accelerate() {
    speed += 10;
}
```

构造函数

作用

- 在对象创建时**自动调用**，用于初始化对象的成员变量。
- 可以重载（定义多个不同参数的构造函数）。

特点

- 函数名与类名**完全相同**。(案例：只能是cat)
- **没有返回值类型**（连 `void` 也没有）

析构函数

作用

- 在对象销毁时**自动调用**，用于释放对象占用的资源（如内存、文件句柄等）。
- 如果对象中有动态分配的内存（`new`），必须在析构函数中释放（`delete`），否则会导致内存泄漏！

特点

- 函数名是 `~类名`。
- **没有参数，也没有返回值类型**。
- 不能重载（每个类只有一个析构函数）。

初始化的三种情况

1.当类的成员变量是内置类型（如int、double等）时，如果不进行初始化，它们的值是未定义的（即垃圾值）。如果这些变量在后续代码中被直接使用，会导致不可预测的行为。

2.特殊背景条件

3.基类构造函数**场景描述**：当一个类继承自基类，并且派生类依赖于基类的某些初始化状态时，必须在派生类的构造函数中显式调用基类的构造函数，以确保基类部分被正确初始化。

```
class Person {
protected:
    string name;
```

```
    int age;

public:
    // 基类构造函数
    Person(const string& personName, int personAge) :
name(personName), age(personAge) {
        if (age < 0) {
            throw invalid_argument("年龄不能为负数");
        }
    }

    void Display() const {
        cout << "姓名: " << name << ", 年龄: " << age << endl;
    }
};

// 派生类
class Employee : public Person {
protected:
    double salary;

public:
    // 正确示例：显式调用基类构造函数
    Employee(const string& empName, int empAge, double empSalary)
        : Person(empName, empAge), salary(empSalary) {
        if (salary < 0.0) {
            throw
```

类的成员

4. 总结对比

访问权限	类内部	派生类	类外部	用途
public	✔	✔	✔	提供类的接口，允许外部访问。
private	✔	✘	✘	隐藏实现细节，保护数据。
protected	✔	✔	✘	在继承中使用，允许派生类访问。

记住这张图就行

隐式/显式转换

1. 隐式转换（偷偷摸摸的转换）

定义：编译器**自动**帮你把一种类型转成另一种类型，**不需要你手动写转换代码。**

特点：悄悄发生，你可能都没注意到！

例子1：基本数据类型的隐式转换

cpp 复制

```cpp
int a = 10;
double b = a;  // 隐式转换：int -> double
```

- a 是 int 类型，b 是 double 类型。
- 编译器**自动**把 int 转成 double，你不用写 (double)a。

例子2：自定义类的隐式转换（构造函数实现）

cpp 复制

```cpp
class MyString {
public:
    MyString(int size) { ... }  // 非 explicit 构造函数
};

void printString(MyString s) { ... }

int main() {
    printString(10);  // 隐式转换：int -> MyString
    return 0;
}
```

2. 显式转换（明明白白的转换）

定义：你必须**手动写代码**告诉编译器："我要把这个类型转成另一个类型"。

特点：强制、明确，一眼就能看出来！

例子1：基本数据类型的显式转换（C风格）

```cpp
int a = 10;
double b = (double)a;  // 显式转换：int -> double
```

- 你**明确写了** (double)a，告诉编译器："把 a 转成 double"。

例子2：C++的 static_cast（更安全的显式转换）

```cpp
int a = 10;
double b = static_cast<double>(a);  // 显式转换：int -> double
```

- static_cast 是 C++ 推荐的显式转换方式，比 C 风格的 (double)a 更安全。

例子3：自定义类的显式转换（explicit 构造函数）

```cpp
class MyString {
public:
    explicit MyString(int size) { ... }  // 加了 explicit 关键字
};

void printString(MyString s) { ... }

int main() {
    // printString(10);  // 错误！不能隐式转换
    printString(MyString(10));  // 正确！必须显式调用构造函数
    return 0;
}
```

拷贝构造函数

拷贝构造函数 是用来把一个对象的内容复制到另一个新对象中的。

类名(const 类名 &引用名);

它会在以下情况下被调用：

1. 用一个对象初始化另一个对象。
2. 函数传值时（传值参数）。

3. 函数返回值时

场景1：用一个对象初始化另一个对象

```
拷贝构造函数的定义：
class MyClass {
public:
    // 拷贝构造函数
    MyClass(const MyClass& other) {
        // 拷贝other的成员到当前对象
    }
};

int main() {
    Book book1("C++编程", 300); // 创建一本书 book1
    Book book2 = book1;          // 用 book1 初始化 book2
    cout << "book2 的书名: " << book2.title << ", 页数: " <<
book2.pages << endl;
}
拷贝构造函数被调用!
book2 的书名: C++编程, 页数: 300
```

场景2：函数传值时

```
void printBook(Book b) { // 参数 b 是传值参数
    cout << "书名: " << b.title << ", 页数: " << b.pages << endl;
}

int main() {
    Book book1("C++编程", 300); // 创建一本书 book1
    printBook(book1);           // 调用函数 printBook
}
```

浅复制

如果你没有为类定义拷贝构造函数，C++编译器会自动生成一个默认的拷贝构造函数。

如果你没有为类定义拷贝构造函数，C++编译器会自动生成一个默认的拷贝构造函数。

```
class MyClass {
public:
    int x;
    int y;
};

int main() {
    MyClass obj1;
    obj1.x = 10;
    obj1.y = 20;

    MyClass obj2 = obj1; // 使用默认拷贝构造函数
    // obj2.x == 10, obj2.y == 20
}
```

默认的拷贝构造函数执行的是浅拷贝（Shallow Copy），即只拷贝对象的成员变量的值。如果对象中有指针成员，浅拷贝会导致两个对象的指针指向同一块内存，这可能会导致问题（如双重释放内存）

问题

浅复制的问题在于，如果对象中有 **指针**，浅复制只会复制指针的值（即地址），而不会复制指针指向的内容。

只复制地址，不复制内容

深复制

```
class Person {
public:
    char *name; // 指针，指向名字

     // 构造函数
    Person(const char *n) {
```

```
        name = new char[strlen(n) + 1]; // 动态分配内存
        strcpy(name, n); // 复制名字
        cout << "构造函数被调用！名字：" << name << endl;
    }

    // 拷贝构造函数（深复制）
    Person(const Person &other) {
        name = new char[strlen(other.name) + 1]; // 重新分配内存
        strcpy(name, other.name); // 复制名字
        cout << "拷贝构造函数被调用！名字：" << name << endl;

    }

    // 析构函数
    ~Person() {
        cout << "析构函数被调用！名字：" << name << endl;
        delete[] name; // 释放内存
    }

    // 打印名字
    void printName() {
        cout << "名字：" << name << endl;
    }
};

int main() {
    // 创建对象 p1
    Person p1("Alice");
    p1.printName();

    // 使用深复制创建对象 p2
    Person p2 = p1;
    p2.printName();

    // 修改 p1 的名字
    strcpy(p1.name, "Bob");
    cout << "修改后的 p1 的名字：";
    p1.printName();

    // 再次打印 p2 的名字
    cout << "p2 的名字：";
    p2.printName();
```

```
    return 0;
}
构造函数被调用！ 名字： Alice
名字： Alice
拷贝构造函数被调用！ 名字： Alice
名字： Alice
修改后的 p1 的名字： 名字： Bob
p2 的名字： 名字： Alice
析构函数被调用！ 名字： Alice
析构函数被调用！ 名字： Bob
```

ee看不惯char，手动内存是吗 look at this,效果一样的

```
class Person {

public:

  string name;



  Person(string s):name(s){}



  Person (const Person & other){

    name=other.name;

    cout<<"拷贝构造函数被调用！ "<<endl;

  }



  ~Person(){

    cout<<"!!"<<endl;

  }
```

```
    // 打印名字
    void printName() {
        cout << "名字: " << name << endl;
    }
};
int main() {
    // 创建对象 p1
    Person p1("Alice");
    p1.printName();

    // 使用深复制创建对象 p2
    Person p2 = p1;
    p2.printName();

    // 修改 p1 的名字
    //strcpy(p1.name, "Bob");
    p1.name="Bob";
    cout << "修改后的 p1 的名字: ";
    p1.printName();
```

```
    // 再次打印 p2 的名字
    cout << "p2 的名字: ";
    p2.printName();

    return 0;
}
```

L：学生成绩

```
class Student{
    private:
    string name;
    vector<double> score;

    public:
    Student(string s,vector<double>& sco){
        name=s;
        score=sco;
    }

    void printGrade(){
        cout<<endl;
```

```
        cout<<name<<endl;
        for(auto a:score){
            cout<<a<<'\t';
        }
    }
    void modifscore(size_t index, double value){
        if(index<score.size()){
            score[index]=value;
        }
    }
};
int main(){
    string name1,name2;
    vector<double> vec1,vec2;
    cin>>name1;
    for(int i=0;i<3;i++){
        double temp;
        cin>>temp;
        vec1.push_back(temp);
    }


    Student s1(name1,vec1);
    Student s2=s1;
```

```
    s1.printGrade();

    s2.printGrade();



    s2.modifscore(0,32);

    s2.printGrade();
}
```

类的其他成员

常对象

若在定义对象的说明语句以 const 作为前缀，则称该对象为常对象。这个对象的全部数据成 员在作用域中约束为只读。

【例 6-13】常对象测试。

```
#include<iostream>
using namespace std;
class T_class
{   public:
        int a,b;
        T_class(int i, int j)
        {   a=i;   b=j;   }
};
int main()
{   const T_class t1(1,2);              //t1 是常对象
    T_class t2(3,4);
    //t1.a=5;                           //错误，不能修改常对象的数据成员
    //t1.b=6;                           //错误，不能修改常对象的数据成员
    t2.a=7;
    t2.b=8;
    cout<<"t1.a="<<t1.a<<'\t'<<"t1.b="<<t1.b<<endl;
    cout<<"t2.a="<<t2.a<<'\t'<<"t2.b="<<t2.b<<endl;
}
```


程序运行结果：

```
t1.a=1          t1.b=2
t2.a=7          t2.b=8
```

常成员函数

void fun()const{}

const写在前面：返回值是const

const写在后面：常成员函数

常成员与静态成员的对比

（4）使用场景

- 静态数据成员：用于存储类的所有对象共享的数据（如计数器、全局配置）。
- 静态成员函数：用于实现与类相关但不依赖于具体对象的工具函数。

---

3. 常成员与静态成员的对比

特性	常成员（Const Members）	静态成员（Static Members）
数据成员	常数据成员的值在对象创建后不能被修改。	静态数据成员属于类本身，所有对象共享同一个值。
成员函数	常成员函数不能修改类的数据成员。	静态成员函数不能访问非静态成员。
初始化	常数据成员必须在构造函数的初始化列表中初始化。	静态数据成员必须在类外定义和初始化。
对象依赖	常成员依赖于对象，每个对象有自己的常数据成员。	静态成员不依赖于对象，属于类本身。
调用方式	常成员函数通过对象调用。	静态成员函数可以直接通过类名调用。
使用场景	用于存储不可变的数据或实现不修改对象状态的函数。	用于存储共享数据或实现工具函数。

- **所有对象共享同一个静态成员变量。**
- **静态成员变量只有一份内存空间**，无论创建多少个对象。
- **静态成员变量需要在类外初始化。**

静态数据成员

- 在类内声明静态数据成员时，需要使用 `static` 关键字。
- 静态数据成员必须在类外定义和初始化。

静态数据成员要求在类中声明，在类外定义。尽管static数据成员从存储性质上是全局变量， 但其作用域是类。

若不指定初值，则系统自动将其 初始化为0。

静态成员函数

在类内声明静态成员函数时，需要使用 `static` 关键字。

静态成员函数可以直接通过类名调用，不需要创建对象。

```cpp
class MathUtils {
public:
    // 静态成员函数
    static int add(int a, int b) {
        return a + b;
    }

    static int multiply(int a, int b) {
        return a * b;
    }
};

int main() {
    // 直接通过类名调用静态成员函数
    int sum = MathUtils::add(10, 20);
    int product = MathUtils::multiply(10, 20);

    cout << "Sum: " << sum << endl;
    cout << "Product: " << product << endl;

    return 0;
}
```

L：统计对象数量

编写一个 `Counter` 类，包含一个静态成员变量 `count`，用于统计创建的对象数量。每次创建对象时，`count` 加 1；每次销毁对象时，`count` 减 1。

要求

1. 在构造函数中增加 `count`。
2. 在析构函数中减少 `count`。

3. 提供一个静态方法 `getCount()`，返回当前的对象数量。

统计对象数量 C++ | 自动换行 开启 | 复制

```cpp
class Counter {
public:
    static int count;  // 静态成员变量

    Counter() {
        count++;  // 创建对象时增加
    }

    ~Counter() {
        count--;  // 销毁对象时减少
    }

    static int getCount() {
        return count;
    }
};

// 初始化静态成员变量
int Counter::count = 0;

int main() {
    Counter c1;
    cout << "当前对象数量: " << Counter::getCount() << endl;  // 输出: 1

    {
        Counter c2;
        cout << "当前对象数量: " << Counter::getCount() << endl;  // 输出: 2
    }
 //注意到，这里有个花括号，这是它的作用域，出了作用域也就是销毁对象，所以2-1=1

    cout << "当前对象数量: " << Counter::getCount() << endl;  // 输出: 1
    return 0;
}

```

L：图书馆系统

```cpp
class Ebook {
public:
    string title;
    string author;
    float price;
    int pages;
```

```
    Ebook(string t, string a, float p, int pg)
        : title(t), author(a), price(p), pages(pg) {}
};

class Control {
public:
static Ebook book;
//: 声明了一个静态成员变量 book，类型为 Ebook。
static vector<Ebook> Lib;
//声明了一个静态成员变量 Lib，类型为 vector<Ebook>（即 Ebook 对象的动态数组）。

    static void addBook(const Ebook& eb) {
        Lib.push_back(eb);      // 向库中添加电子书
    }

    static void showLib() {
        for (const auto& eb : Lib) {
            cout << "Title: " << eb.title << ", Author: " << eb.author
                 << ", Price: " << eb.price << ", Pages: " << eb.pages << endl;
        }
    }
};


初始化
vector<Ebook> Control::Lib;
//初始化静态成员变量 Lib，类型为 vector<Ebook>。这里没有显式赋值，因此 Lib 会被默认初始化为一个空的 vector。
Ebook Control::book("", "", 0.0f, 0);
//初始化静态成员变量 book，类型为 Ebook。这里调用了 Ebook 类的构造函数，传递了空字符串、0.0f 和 0 作为参数。
int main() {
    // 添加电子书
    Control::addBook(Ebook("C++ Primer", "Stanley", 99.99f, 1024));
    Control::addBook(Ebook("Clean Code", "Robert", 49.99f, 464));

    // 显示电子书库
    Control::showLib();
```

```
    return 0;
}
```

L:学生管理系统

题目

编写一个 `Student` 类，包含以下成员：

1. 静态成员变量 `totalScore`，用于记录所有学生的总成绩。
2. 静态成员变量 `studentCount`，用于记录学生数量。
3. 普通成员变量 `name` 和 `score`，记录学生的姓名和成绩。
4. 方法 `addScore(amount)`：增加学生的成绩，并更新 `totalScore`。
5. 静态方法 `getAverageScore()`，返回所有学生的平均成绩。

要求

1. 在构造函数中更新 `studentCount` 和 `totalScore`。
2. 在析构函数中更新 `studentCount` 和 `totalScore`。
3. 提供方法计算平均成绩。

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

class Student {
public:
    static double totalScore;  // 静态成员变量：总成绩
    static int studentCount;   // 静态成员变量：学生数量
    string name;
    double score;

    // 带参构造函数
    Student(string n, double s) : name(n), score(s) {
        studentCount++;        // 学生数量加 1
        totalScore += score;   // 更新总成绩
    }

    // 默认构造函数（如果需要）
    Student() : name(""), score(0) {
        studentCount++;        // 学生数量加 1
        totalScore += score;   // 更新总成绩
    }

    // 增加成绩
    void addScore(double amount) {
        score += amount;
        totalScore += amount;
    }

    // 计算平均成绩
    static double getAverageScore() {
        if (studentCount == 0) return 0;  // 避免除以 0
        return totalScore / studentCount;
    }
};

// 初始化静态成员变量
double Student::totalScore = 0.0;
int Student::studentCount = 0;

int main() {
    string n1, n2;
    double s1, s2;
    cin >> n1 >> s1;
```

```
    cin >> n2 >> s2;

    Student stu1(n1, s1);  // 创建学生 1
    Student stu2(n2, s2);  // 创建学生 2

    stu1.addScore(23.5);   // 学生 1 增加成绩
    stu2.addScore(78.3);   // 学生 2 增加成绩

    cout << n1 << " " << stu1.score << endl;  // 输出学生 1 的成绩
    cout << n2 << " " << stu2.score << endl;  // 输出学生 2 的成绩
    cout << Student::getAverageScore() << endl;  // 输出平均成绩

```

友元

友元函数是一个 **非成员函数**，但它可以访问类的 **私有成员** 和 **保护成员**。友元函数通过在类中声明 `friend` 关键字来定义。

（1）**访问多个类的私有成员**

- 如果一个函数需要访问多个类的私有成员，可以将它声明为这些类的友元函数。

（2）**运算符重载**

- 在某些情况下，运算符重载函数需要访问类的私有成员，可以将它声明为友元函数。

内存对齐（Alignment）

成员字节大小

char: 1字节

short: 2字节

int: 4字节

long: 4或8字节（取决于平台）

float: 4字节

double: 8字节

指针: 4或8字节（32位系统为4字节，64位系统为8字节）

```
class Example {
    char c;     // 1字节
    int i;      // 4字节
};
尽管char和int共5字节，但由于对齐要求，Example类的大小可能是8字节。
char c占用1字节，假设它存储在地址0x0000。

int i需要4字节，并且需要对齐到4字节边界。这意味着int i的起始地址必须是4的倍数。

由于char c已经占用了地址0x0000，接下来的地址是0x0001。但是0x0001不是4的倍数，因此编译器会在
char c后面插入3字节的填充（padding），以确保int i从地址0x0004开始存储。

class Example2 {
    int i;      // 4字节
    char c;     // 1字节
};
int i需要4字节对齐，假设它从地址0x0000开始存储，占用地址0x0000-0x0003。

char c只需要1字节，可以紧跟在int i后面，存储在地址0x0004。

此时，类的总大小是4字节（int i） + 1字节（char c） = 5字节。
```

```
class MixedData {
    char c;       // 1字节
    short s;      // 2字节
    int i;        // 4字节
    double d;     // 8字节
};
我们把所有成员和填充字节加起来：

char c：1字节

填充：1字节

short s：2字节

int i：4字节

double d：8字节

总大小 = 1 + 1 + 2 + 4 + 8 = 16字节。
```

静态成员不属于类的实例，因此不计入类的大小。

运算符重载

不能重载的

以下运算符不能重载：

. .* :: ?: sizeof

重载运算符函数可以对运算符做出新的解释，即定义用户所需要的各种操作。但运算符重载后，原有的基本语义不变，包括：

- 不改变运算符的优先级；
- 不改变运算符的结合性；
- 不改变运算符所需要操作数的个数。

优先级和结合性主要体现在重载运算符的使用上，而操作数的个数不但体现在重载运算符的使用上，更关系到函数定义时的参数设定。例如，一元运算符重载函数不能有两个参数，调用时也不能作用于两个对象。

不能创建新的运算符，只有系统预定义的运算符才能被重载。

7.1.2 重载运算符的语法

运算符函数是一种特殊的成员函数或友元函数。成员函数的语法格式为：

```
类型 类名::operator op(参数表)
{
    //相对于该类定义的操作
}
```

其中，“类型”是函数的返回类型。“类名”是要重载该运算符的类。“op”表示要重载的运算符。函数名是“operator op”，由关键字 operator 和被重载的运算符 op 组成。“参数表”列出该运算符所需要的操作数。

定义

无论是**一元运算符**还是**二元运算符**，都可以在**类外定义**（通过友元函数或普通函数），但需满足以下条件：

1. 二元运算符：

- 如果左操作数**不是**当前类的对象（如 `5 + obj`），则**必须**在类外定义（通常为友元函数）。
- 示例：

```cpp
friend Complex operator+(int a, const Complex& obj); // 类外定义
```

2. 一元运算符：

- 也可在类外定义，但需通过**友元函数**（因为一元运算符的操作数通常是类对象）。
- 示例：

```cpp
friend Complex operator-(const Complex& obj); // 类外定义一元负号
```

记忆： 左翼，相信二元论，是异教徒，必须类外定义 一元（一神论）无所谓，是朋友

用成员函数重载

成员函数有this指针，友元函数没有

- **友元函数重载**：更灵活，支持左、右操作数类型不同，代码更简洁。
- **成员函数重载**：左操作数必须是类的对象，不支持隐式类型转换，代码稍显繁琐。

用友元函数重载

L：复数

```
class Complex {
private:
    int real, image;  // 实部和虚部

public:
    Complex(int a = 0, int b = 0) : real(a), image(b) {}  // 构造函数

    // 成员函数重载 + 运算符
```

```
    Complex operator+(const Complex &c) const {
        return Complex(real + c.real, image + c.image);
    }

    void print() const {  // 打印复数
        cout << "(" << real << ", " << image << ")" << endl;
    }
}

class Complex {
private:
    int real, image;  // 实部和虚部

public:
    Complex(int a = 0, int b = 0) : real(a), image(b) {}  // 构造函
数

    // 友元函数重载 + 运算符
    friend Complex operator+(const Complex &c1, const Complex &c2);

    void print() const {  // 打印复数
        cout << "(" << real << ", " << image << ")" << endl;
    }
};

// 定义 + 运算符的重载函数
Complex operator+(const Complex &c1, const Complex &c2) {
    return Complex(c1.real + c2.real, c1.image + c2.image);
}
```

关于c3还是c1+c2的小细节

很简单，但是有些小细节

Q：为什么 cout<<c3<<endl;可以输出而 cout<<c1+c2<<endl;却是错误的

A：

- `c1 + c2` 返回的临时对象在表达式结束后会被销毁。如果 `operator<<` 接受非常量引用，编译器会尝试延长临时对象的生命周期以匹配引用的生命周期，但这是不允许的

- 若将 `operator<<` 的参数改为 `const Complex&`，则可以接受临时对象，因为常量引用可以绑定到右值（临时对象

ostream& operator<<(ostream&os, const Complex&c)

重载赋值运算符

重载++和--

前置 `++` 和 `--` 运算符的作用是**先自增/自减，再返回修改后的值。**

后置 `++` 和 `--` 运算符的作用是**先返回当前值，再自增/自减。**

```
class Counter {
private:
    int count;

public:
    Counter(int c = 0) : count(c) {}

    // 前置++运算符重载
    Counter& operator++() {
        ++count;
        return *this;
    }
  friend Counter&operator++(Counter&c){
  }

    // 后置++运算符重载
    Counter operator++(int) {
        Counter temp = *this;  // 保存当前值
        ++(*this);  // 调用前置++
        return temp;  // 返回保存的值
    }
```

重载 [] 与 () 运算符

只能用成员函数，不能用友元函数重载

```
//[]重载
    int& operator[](int index){
        if(index<0||index>=size){
            throw out_of_range("索引越界");
        }
        return p[index];
    }
    //const 只读不能改
    int&operator[](int index)const{
        if(index<0||index>=size){
            throw out_of_range("索引越界");
        }
        return p[index];
    }

    //()重载
    int operator()(int index){
        if(index<0||index>=size){
            throw out_of_range("索引越界");
        }
        return p[index];
    }
```

要写在 public 里面，主要是写成友元不好看，破坏了封装

```
  Vector vec9(5,10);
    cout<<"vec9[0]="<<vec9[0]<<endl;
    cout<<"vec9(1)="<<vec9(1)<<endl;
```

两者的区别：

前者，像数组一样访问，后者像函数一样调用，可以实现一些运算

求区间和

```
// 函数调用运算符 (): 求区间和
    int operator()(int start, int end) const {
        if (start < 0 || end >= size || start > end) throw out_of_range("非法范围");
        int sum = 0;
        for (int i = start; i <= end; ++i) sum += p[i];
        return sum;
    }

    cout << "vec(1, 3)的和 = " << vec(1, 3) << endl; // 输出: 20+30+40=90
```

重载流插入与流提取运算符

输出流可以用友元和成员函数

```cpp
friend ostream &operator<<(ostream&os,Rational&r){
        os<<r.p<<"/"<<r.q<<endl;
        return os;
    }
    // ostream& operator<<(ostream &os)
    // {
    //     os <<this-> p << "/" <<this-> q << endl;
    //     return os;
    // }

        cout<<"r1= "<<r1;
```

但是，如果用成员函数的话，无法直接输出r1

成员函数版本的调用方式

当 operator<< 是成员函数时，它的第一个参数是隐式的 this 指针，指向当前对象。因此，正确的调用方式变成了：

```cpp
r1 << cout; // 成员函数版本的调用方式
```

这显然不符合我们的习惯用法，也与标准库中其他类型的输出方式不一致。

为什么友元函数版本更合适

友元函数版本的 operator<< 需要两个参数（ostream& 和 const Rational&），这使得我们可以按照习惯的方式使用它：

```cpp
cout << r1; // 友元函数版本的调用方式
```

bool

还是上面的Vector类

重载==和！=运算符用千判断向量是否相等。

```
bool operator==(const Vector&other)const{
        return data == other.data;
    }
 if(v==v3){
        cout<<"v=v3";
    }else{
        cout<<"v!=v3";
    }

```

类类型转换

类类型转换函数只能定义为一个类的成员函数，不能是友元函数

【例 7-9】简单串类与字符串之间的类型转换。

```
#include<iostream>
using namespace std;
class String
{       char *data;
        int size;
    public:
        String( char* s)
        {   size=strlen(s);
            data = new char(size+1);
            strcpy_s(data,size+1,s);
        }
        operator char* () const          //类型转换函数
        {   return data;   }
};
void main()
{   String sobj = "hell";
    char * svar = sobj;                  //把String型对象赋给字符串变量，进行了类型转换
    cout<<svar<<endl;
}
```

上述程序中，构造函数用一个字符串建立对象，即把 char*型转换成 String 对象。成员函数

```
String::operator char* () const;
```

把一个 String 型对象*this 转换成字符串 char*型变量，实际上返回了数据成员 this->data。注意，以下函数名中：

```
operator char*
```

类型标识符是 char*，表示字符指针，是由两个单词组成的复合类型符。

L：Vector

```
class Vector{
    private:
    vector<int>data;

    public:
    Vector(int n=0){
        data.resize(n);
        srand(time(nullptr));

        for(auto &a:data){
            a=rand()%100;
        }
    }

    void print(){
        cout<<"(";
        for(auto a:data){
            cout<<a<<",";
        }
        cout<<")"<<endl;
```

```
    }

    Vector(const Vector&other): data(other.data){}

    Vector&operator=(const Vector&other){
        if(this!=&other){
            data =other.data;
        }
        return *this;
    }

    friend Vector operator+(Vector&v1,Vector&v2);
    friend Vector operator+(Vector&v1,int value);
    friend Vector operator*(Vector&v1,Vector&v2);
    friend ostream &operator<<(ostream& os,const Vector&v);

    //前置++和后置++
    Vector& operator++(){
        for(auto& a:data){
            ++a;
        }
```

```
        return *this;
    }
    Vector&operator++(int){
        Vector temp(*this);
        for(auto &a: data){
            ++a;
        }
        return temp;
    }

    //[]和（）
    int& operator[](int index){
        if(index<0||index>=data.size()){
            throw out_of_range("索引越界");
        }
        return data[index];
    }

    int& operator()(int index){
        if(index<0||index>=data.size()){
            throw out_of_range("索引越界");
```

```
        }
        return data[index];
    }
};

    Vector operator+(Vector& v1,Vector& v2) {
        Vector result(max(v1.data.size(), v2.data.size()));
        for (int i = 0; i < result.data.size(); ++i) {
            int val1 = (i < v1.data.size()) ? v1.data[i] : 0;
            int val2 = (i < v2.data.size()) ? v2.data[i] : 0;
            result.data[i] = val1+val2;
        }
        return result;
    }

    Vector operator+(Vector&v1,int value){
        Vector result=v1;
        result.data.push_back(value);
        return result;
    }
```

```
Vector operator*(Vector&v1,Vector&v2){
    if(v1.data.size()!=v2.data.size()){
        cout<<"wrong!";
        return Vector(0);
    }

    Vector result(1);
    int sum;
    for(auto a: v1.data){
        for(auto b:v2.data){
          sum+=a*b;
        }
    }
    result[0]=sum;
    return result;
}
ostream& operator<<(ostream& os, const Vector&v){
    if(v.data.size()==1){
        os<<v.data[0];
    }else{
    os<<"(";
```

```
        for(auto a:v.data){
            os<<a<<",";
            }
        os<<")";
        }
        return os;
    }


    int main(){
        Vector v1(5);
        cout<<"v1: ";
        v1.print();
        Vector v2=v1;
        v2.operator++();
        cout<<"v2: ";
        v2.print();
        Vector v3=v1+v2;
        Vector v4=v1*v2;
        cout<<"v1+v2= "<<v3<<endl;
        cout<<"v1*v2="<<v4<<endl;
    }
```

transform大法好

```
//old
    Vector operator+(Vector& v1,Vector& v2) {

        Vector result(max(v1.data.size(), v2.data.size()));

        for (int i = 0; i < result.data.size(); ++i) {

            int val1 = (i < v1.data.size()) ? v1.data[i] : 0;

            int val2 = (i < v2.data.size()) ? v2.data[i] : 0;

            result.data[i] = val1+val2;

        }

        return result;

    }

//很好用啊
    friend Vector operator+(Vector v1, const Vector& v2) {

        v1.data.resize(max(v1.size(), v2.size()));

        transform(v2.data.begin(), v2.data.end(), v1.data.begin(),

                v1.data.begin(), plus<int>());

        return v1;

    }

    friend Vector operator-(Vector v1, const Vector& v2) {

        v1.data.resize(max(v1.size(), v2.size()));

        transform(v2.data.begin(), v2.data.end(), v1.data.begin(),

                v1.data.begin(), minus<int>());
```

```
        return v1;
    }
```

ostream容易写错

```
ostream& operator<<(ostream& os, const Vector&v)
容易写成Vector开头
```

不要返回临时变量

```
  Matrix operator*(int n) const
  {
    Matrix result(rows);
    for (int i = 0; i < rows; i++)
    {
      for (int j = 0; j < cols; j++)
      {
        result.data[i][j] = data[i][j] * n;
      }
    }
    return result;
  }
注意不是 Matrix & operator()
friend 也没有
```

除了几个特殊的都没有

L：Rational

GCD的数学原理

比如 **48 和 18**：

1. **用大数除以小数，取余数**：
   48 ÷ 18 = 2 余 **12**（余数就是 48 减两次 18 剩下的数）。
2. **把大数换成余数，重复步骤 1**：
   现在计算 **18 和 12**：
   18 ÷ 12 = 1 余 **6。**
3. **继续替换，直到余数为 0**：
   12 ÷ 6 = 2 余 **0。**
   当余数为 0 时，**除数 6 就是 GCD**！

```cpp
int gcd(int m, int n)
{
    while (n != 0)
    {
        int t = m % n;
        m = n;
        n = t;
    }
    return m;
}

  Rational(int a = 0, int b = 1) : p(a), q(b)
    {
        if (q == 0)
            throw invalid_argument("分母不能为0");
        if (q < 0)
        {
            p = -p;
            q = -q;
        }
        int temp = gcd(abs(p), abs(q));
        p /= temp;
        q /= temp;
    }
```

先保存余数，再更新 `m` 和 `n`。 交错更新

口诀：
“除数变余数，余数变除数，直到余数为零，除数就是答案！”

拆解记忆：

1. **循环条件**：`while(y != 0)` → 只要余数不为 0，就继续循环。
2. **关键三步**：
   - `t = y` → 把当前的除数 (y) 存起来。
   - `y = x % y` → 新的余数是原来的除数除以余数。
   - `x = t` → 新的除数是刚才存起来的旧除数。
3. **返回结果**：当余数为 0 时，最后的除数 (x) 就是 GCD。

为什么不能直接输出r1+r2

`operator+` 接受的是非常量引用

你定义的 `operator+` 函数接受的是 `Rational&`（非常量引用）作为参数：

```
friend Rational operator+(Rational&r1, Rational&r2);
 Rational r3=r1+r2; 正确
    cout<<r1+r2; 错误
```

然而，`r1 + r2` 这个表达式本身是一个 **右值**（临时对象），不能绑定到非常量引用上。因此，编译器会报错，指出无法将 `r1 + r2` 传递给接受非常量引用的 `operator+`。

建议**存储结果并输出**（推荐）

如果你非要直接输出临时变量呢?

1.加上const

friend ostream&operator<<(ostream&os,const Rational&r);

ostream&operator<<(ostream&os,const Rational&r){

2.加上（）

cout << (r1 + r2)

不能返回临时变量的引用和指针

在C++中，返回临时变量的引用或指针是一个常见的错误，因为临时变量在表达式结束后会被销毁，导致返回的引用或指针指向无效的内存。

返回临时变量引用的例子

cpp 复制

```cpp
#include <iostream>

// 错误示例：返回局部变量的引用
int& getTemporaryRef() {
    int temp = 42;  // 局部变量，函数结束后会被销毁
    return temp;    // 返回它的引用 - 危险！
}

int main() {
    int& ref = getTemporaryRef();  // ref现在指向一个无效的内存位置
    std::cout << ref << std::endl; // 未定义行为 - 可能输出42，也可能是垃圾值
    return 0;
}
```

返回临时变量指针的例子

cpp 复制

```cpp
#include <iostream>

// 错误示例：返回局部变量的指针
int* getTemporaryPtr() {
    int temp = 100;  // 局部变量，函数结束后会被销毁
    return &temp;    // 返回它的地址 - 危险！
}

int main() {
    int* ptr = getTemporaryPtr();  // ptr现在指向一个无效的内存位置
    std::cout << *ptr << std::endl; // 未定义行为 - 可能输出100，也可能是垃圾值
    return 0;
}
```

正确做法

如果需要返回一个值，应该：

1. 返回值本身（而不是引用或指针）
2. 使用动态分配的内存（并确保正确管理）
3. 使用静态或全局变量（但要注意线程安全问题）

4.
```cpp
#include <iostream>

// 正确示例1：返回值
int getValue() {
    int temp = 42;
    return temp;  // 返回值的拷贝
}

// 正确示例2：返回动态分配的内存（需要调用者释放）
int* getDynamicValue() {
    int* temp = new int(42);
    return temp;  // 调用者需要负责delete
}

// 正确示例3：返回静态变量（注意线程安全）
int& getStaticValue() {
    static int temp = 42;
    return temp;  // 返回静态变量的引用
}

int main() {
    int val1 = getValue();  // 安全
    std::cout << val1 << std::endl;

    int* val2 = getDynamicValue();
    std::cout << *val2 << std::endl;
    delete val2;  // 必须释放

    int& val3 = getStaticValue();
    std::cout << val3 << std::endl;

    return 0;
}
```

重载运算符

1. **运算符重载中返回引用/指针时**：

- 只能返回成员变量或静态变量的引用/指针
- 不能返回局部变量的引用/指针
- 对于新建的对象，通常应该返回值而不是引用

1. **常见可以返回引用的情况**：

- 返回 `*this`（如赋值运算符）

- 返回成员变量的引用（如 `operator[]` 返回数组元素）
- 返回静态成员变量的引用

1. **对于新建对象**：

- 通常应该返回值（如 `operator+`）

示例：重载 `+` 运算符返回引用

cpp 复制

```cpp
#include <iostream>

class MyClass {
public:
    int value;

    MyClass(int v) : value(v) {}

    // 错误：返回临时对象的引用
    MyClass& operator+(const MyClass& other) {
        MyClass temp(value + other.value);  // 临时对象
        return temp;  // 错误！temp会在函数结束时被销毁
    }
};

int main() {
    MyClass a(10), b(20);
    MyClass& c = a + b;  // c引用了一个已经被销毁的对象
    std::cout << c.value << std::endl;  // 未定义行为
    return 0;
}
```

正确做法：返回值而不是引用

cpp 复制

```cpp
MyClass operator+(const MyClass& other) const {
    return MyClass(value + other.value);  // 返回新对象
}
```

示例：重载 `=` 运算符

cpp 复制

```cpp
#include <iostream>

class MyClass {
    int* data;
public:
    MyClass(int val) { data = new int(val); }

    // 正确：返回当前对象的引用
    MyClass& operator=(const MyClass& other) {
        if (this != &other) {
            *data = *other.data;  // 复制值
        }
        return *this;  // 返回当前对象的引用
    }

    ~MyClass() { delete data; }

    int getValue() const { return *data; }
};

int main() {
    MyClass a(10), b(20);
    a = b;  // 正确使用赋值运算符
    std::cout << a.getValue() << std::endl;  // 输出20
    return 0;
}
```