运算符的重载

运算符重载的方法

模板如下

```
#include <iostream>

// 假设我们有一个简单的Point类
template <typename T>
class Point {
public:
    T x, y;

    // 构造函数
    Point(T x, T y) : x(x), y(y) {}

    // 运算符重载模板：加法
    Point operator+(const Point& rhs) const {
        return Point(x + rhs.x, y + rhs.y);
    }

    // 运算符重载模板：减法
    Point operator-(const Point& rhs) const {
        return Point(x - rhs.x, y - rhs.y);
    }

    // 运算符重载模板：乘法（标量乘法）
    friend Point operator*(const Point& lhs, T scalar) {
        return Point(lhs.x * scalar, lhs.y * scalar);
    }

    // 运算符重载模板：除法（标量除法）
    friend Point operator/(const Point& lhs, T scalar) {
        return Point(lhs.x / scalar, lhs.y / scalar);
    }

    // 运算符重载模板：赋值
    Point& operator=(const Point& rhs) {
        if (this != &rhs) {
            x = rhs.x;
            y = rhs.y;
        }
        return *this;
    }

    // 运算符重载模板：输出流
    friend std::ostream& operator<<(std::ostream& os, const Point& p) {
        os << "(" << p.x << ", " << p.y << ")";
        return os;
    }
};

int main() {
```

```
    Point<int> p1(1, 2), p2(3, 4), p3;
    p3 = p1 + p2;  // 使用+运算符
    std::cout << "p3 = " << p3 << std::endl;

    p3 = p1 - p2;  // 使用-运算符
    std::cout << "p3 = " << p3 << std::endl;

    p3 = p1 * 2;   // 使用*运算符（标量乘法）
    std::cout << "p3 = " << p3 << std::endl;

    p3 = p1 / 2;   // 使用/运算符（标量除法）
    std::cout << "p3 = " << p3 << std::endl;

    p3 = p1;       // 使用=运算符
    std::cout << "p3 = " << p3 << std::endl;

    return 0;
}
```

重载运算符的规则

不能被重载的运算符:

> .(成员访问)::(域运算符)*(成员指针访问运算符)
> ?:(条件运算符)sizeof

注意

> 1. 重载不能改变运算符所需操作数的数目

比如

```
#include <iostream>

class Vector {
public:
    double x, y;

    Vector(double x, double y) : x(x), y(y) {}

    // 正确的加法运算符重载：需要两个操作数
    Vector operator+(const Vector& other) const {
        return Vector(x + other.x, y + other.y);
    }
};

int main() {
    Vector v1(1, 2), v2(3, 4), v3;
    v3 = v1 + v2; // 正确的用法：两个操作数
    std::cout << "v3: (" << v3.x << ", " << v3.y << ")" << std::endl;
    return 0;
}

// 错误：尝试重载+运算符为只需要一个操作数
Vector operator+() const; // 这是错误的，不能这么做
```

```
// 错误：尝试重载+运算符为需要三个操作数
Vector operator+(const Vector& first, const Vector& second, const Vector& third);
// 这是错误的，不能这么做
```

> 2. 重载运算符不能改变优先级
> 3. 不能改变运算符的结合性
> 4. 不能带有默认参数

```
#include <iostream>

class Point {
public:
    double x, y;

    Point(double x = 0, double y = 0) : x(x), y(y) {} // 默认构造函数

    // 错误的加法运算符重载：尝试带有默认参数
    Point operator+(const Point& rhs = *this) const {
        return Point(x + rhs.x, y + rhs.y);
    }
};

int main() {
    Point p1(1, 2);
    Point p2(3, 4);
    Point p3 = p1 + p2; // 正确调用
    Point p4 = p1 + ;    // 错误：编译器无法解析使用哪个默认参数

    return 0;
}
```

> 4. 重载运算符的参数至少有一个是类对象(如果不是的话还有啥意义么)

运算符重载函数作为类成员函数和友元函数

作为类成员函数

例子如下

```
#include <iostream>

// 定义Vector类
class Vector {
public:
    int x, y;  // 向量的x和y分量

    // 构造函数
    Vector(int x = 0, int y = 0) : x(x), y(y) {}

    // 重载加法运算符（在类内定义）
    Vector operator+(const Vector& other) const {
        return Vector(x + other.x, y + other.y);
    }
```

```
};


int main() {
    Vector v1(1, 2), v2(3, 4), v3;

    // 使用重载的加法运算符
    v3 = v1 + v2;
    std::cout << "v3 = (" << v3.x << ", " << v3.y << ")" << std::endl;
    return 0;
}
```

重载运算符作为友元函数

```
#include <iostream>

class Point {
public:
    int x, y;  // 点的x和y坐标

    // 构造函数
    Point(int x = 0, int y = 0) : x(x), y(y) {}

    // 声明友元函数，用于重载加法运算符
    friend Point operator+(const Point& a, const Point& b);

    // 声明友元函数，用于重载输出运算符
    friend std::ostream& operator<<(std::ostream& os, const Point& p);
};

// 重载加法运算符（友元函数）
Point operator+(const Point& a, const Point& b) {
    return Point(a.x + b.x, a.y + b.y);
}

// 重载输出运算符（友元函数）
std::ostream& operator<<(std::ostream& os, const Point& p) {
    os << "(" << p.x << ", " << p.y << ")";
    return os;
}

int main() {
    Point p1(1, 2), p2(3, 4), p3;

    // 使用重载的加法运算符
    p3 = p1 + p2;
    std::cout << "p3 = " << p3 << std::endl; // 使用重载的输出运算符

    return 0;
}
```

注意

1. C++规定，赋值运算符`=`、下标运算符`[]`、函数调用运算符`()`、成员运算符`->`必须作为成员函数。

2. 流插入`<<`和流提取运算符`>>`、类型转换运算符不能定义为类的成员函数，只能作为友元函数。

3. 一般将单目运算符和复合运算符`(+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=)`重载为成员函数。

4. 一般将双目运算符重载为友元函数。

重载单目运算符

例子如下

```
#include <iostream>
#include <stdexcept> // For std::out_of_range

class Number {
private:
    int value;

public:
    // 构造函数
    Number(int val = 0) : value(val) {}

    // 重载前缀递增运算符
    Number& operator++() {
        ++value;
        return *this;
    }

    // 重载前缀递减运算符
    Number& operator--() {
        --value;
        return *this;
    }

    // 重载正号运算符
    Number operator+() const {
        return *this;
    }

    // 重载负号运算符
    Number operator-() const {
        return Number(-value);
    }

    // 重载下标运算符
    int& operator[](int index) {
        if (index != 0) {
            throw std::out_of_range("Number class only supports index 0");
        }
        return value;
    }
```

```
    // 重载函数调用运算符
    int operator()(int times) const {
        return value * times;
    }

    // 输出运算符重载，用于打印Number对象
    friend std::ostream& operator<<(std::ostream& os, const Number& num) {
        os << num.value;
        return os;
    }

    // 获取值的成员函数，用于测试
    int getValue() const {
        return value;
    }
};

int main() {
    Number n(10);

    std::cout << "Original value: " << n << std::endl;

    ++n; // 使用++运算符
    std::cout << "After prefix ++: " << n << std::endl;

    n[0] = 20; // 使用[]运算符
    std::cout << "After setting value at index 0: " << n << std::endl;

    std::cout << "Value at index 0: " << n[0] << std::endl;

    std::cout << "Calling number as a function: " << n(3) << std::endl; // 使用()
运算符

    return 0;
}
```

输出结果

```
Original value: 10
After prefix ++: 11
After setting value at index 0: 20
Value at index 0: 20
Calling number as a function: 60
```

重载流插入运算符和提取流运算符

对"<<"和">>"重载的函数形式如下：

```
istream& operator>>(istream&, 自定义类&);
ostream& operator<<(ostream&, 自定义类&);
```

重载插入运算符

```
#include <iostream>

class Rectangle {
private:
    double width;
    double height;

public:
    // 构造函数
    Rectangle(double w, double h) : width(w), height(h) {}

    // 重载插入运算符，使其能够输出Rectangle对象的信息
    friend std::ostream& operator<<(std::ostream& os, const Rectangle& rect) {
        os << "Rectangle: width = " << rect.width << ", height = " <<
rect.height;
        return os;
    }
};

int main() {
    Rectangle rect(10.0, 20.0);

    // 使用重载的插入运算符输出Rectangle对象
    std::cout << rect << std::endl;

    return 0;
}
```

提取流运算符

```
#include <iostream>

class Rectangle {
private:
    double width;
    double height;

public:
    // 默认构造函数
    Rectangle() : width(0), height(0) {}

    // 重载提取运算符>>，使其能够从istream中提取Rectangle对象的数据
    friend std::istream& operator>>(std::istream& is, Rectangle& rect) {
        std::cout << "Enter width: ";
        is >> rect.width;
        std::cout << "Enter height: ";
        is >> rect.height;
        return is;
    }

    // 显示Rectangle信息的成员函数
    void display() const {
```

```
        std::cout << "Rectangle: width = " << width << ", height = " << height <<
std::endl;
    }
};

int main() {
    Rectangle rect;
    std::cout << "Enter the dimensions of the rectangle:" << std::endl;
    std::cin >> rect; // 使用重载的提取运算符

    rect.display(); // 显示Rectangle对象的信息

    return 0;
}
```

1. 有了运算符重载，在声明了类之后，人们就可把用于标准类型的运算符用于自己声明的类。
2. 使用运算符重载的**具体做法**是：

   I. 先确定要重载的是哪个运算符，想把它用于哪个类，重载运算符只能把一个运算符用于一个类。

   II. 设计运算符重载函数和有关的类。

   III. 一般是有人编好一批运算符重载函数，集中放在一个头文件，放在指定的目录中，提供给有关的人使用。

   IV. 使用者需要了解该头文件有哪些运算符重载，适用于什么类，函数原型。

   V. 没有现成的重载运算符可用，需要自己设计。
3. 注意在运算符重载中使用**引用**的重要性。
4. C++中**大多数**运算符都可以重载。

不同类型数据间的转换

标准类型数据间的转换

1. 标准类型转换

   因为它是**隐式**的，在书写时没有什么痕迹，要记住它的转换规则：

   1. 当 `char` 或 `short` 类型对象与 `int` 类型对象进行运算时，将 `char` 或 `short` 转换为 `int` 类型。
   2. 当两个操作数对象类型不一致时，在算术运算前，级别低的自动转换为级别高的类型。
   3. 在赋值表达式 `E1 = E2` 的情况下，赋值运算符右端的结果值需转换为 `E1` 类型后进行赋值。
2. 显式类型转换

   1. **强制转换法** -- C 语言中学过

      `(类型名)表达式`

      例如：

      ```
      int i, j;
      cout << (float)(i + j);
      ```

   2. **函数法**

      `类型名(表达式)`

例如：

```
int i, j;
cout << float(i + j);
```

此时 `i + j` 作为 `float` 的参数

注意

> 用户自己定义的类,系统不知道该如何转换,需要定义专门的转化函数

转换构造函数

例子如下

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

class Complex {
public:
    double real;
    double imag;

    // 转化构造函数
    Complex(double r) : real(r), imag(0) {
        cout << "Conversion constructor called." << endl;
    }

    void display() const {
        cout << "Real: " << real << ", Imaginary: " << imag << endl;
    }
};

int main() {
    Complex c = 5.0;  // 使用转化构造函数将 double 转换为 Complex
    c.display();

    return 0;
}
```

归纳起来，使用转换构造函数将一个指定的数据转换为类对象的方法如下：

1. 先声明一个类
2. 在这个类中定义一个只有一个参数的构造函数，参数的类型是需要转换的类型，在函数体中指定转换的方法。
3. 在该类的作用域内可以用以下形式进行类型转换：

   类名(指定类型的数据)

就可以将指定类型的数据转换为此类的对象。

类型转换函数

类型转换函数是类的成员函数，用于将对象转换为其他类型。其一般形式如下：

```
operator type() const {
    // 转换逻辑
}
```

例子如下

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

class Complex {
public:
    double real;
    double imag;

    Complex(double r, double i) : real(r), imag(i) {}

    // 类型转换函数，将 Complex 转换为 double
    operator double() const {
        return real;  // 仅返回实部作为 double
    }
};

int main() {
    Complex c(3.5, 2.5);
    double realPart = c;  // 使用类型转换函数
    cout << "Real part: " << realPart << endl;

    return 0;
}
```