Overloading

05 July 2021 14:53

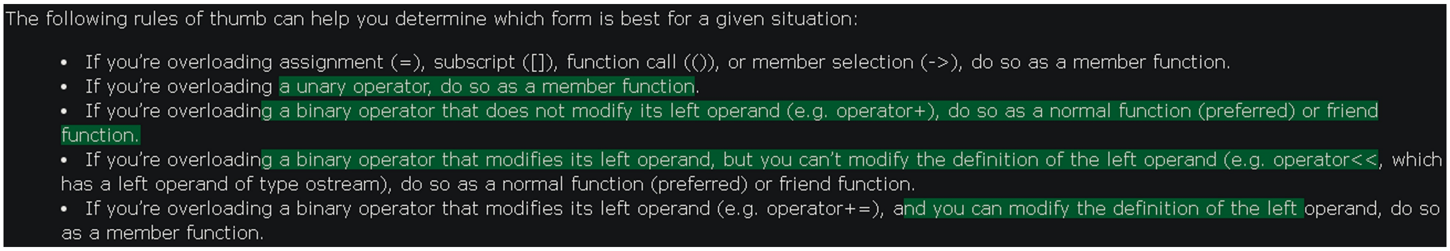
1. Compiler resolving overloaded operators

1. If operands are fundamental datatypes -> use built-in operators
2. If not, check if overloaded operators are present .. If not convert operands to fundamental types and use built-in operators

2. Operands of overloaded operators are the objects

3. Rules of operator overloading

- Operands which needs 'this' use, modify member overloads
  - Unary operator overload funcs has no operands - use member overloads
- If left operand cant be this use normal functions
- Other cases use normal functions



The following rules of thumb can help you determine which form is best for a given situation:

- If you're overloading assignment (=), subscript ([]), function call (()), or member selection (->), do so as a member function.
- If you're overloading a unary operator, do so as a member function.
- If you're overloading a binary operator that does not modify its left operand (e.g. operator+), do so as a normal function (preferred) or friend function.
- If you're overloading a binary operator that modifies its left operand, but you can't modify the definition of the left operand (e.g. operator<<, which has a left operand of type ostream), do so as a normal function (preferred) or friend function.
- If you're overloading a binary operator that modifies its left operand (e.g. operator+=), and you can modify the definition of the left operand, do so as a member function.

4. The exceptions are:

1. conditional (?:)
2. Sizeof
3. scope (::)
4. member selector (.)
5. member pointer selector (.*)
6. typeid
7. casting operators

5. Operator overloading

1. the member function way,
2. the friend function way, and
3. the normal function way

```
Class Cents{
friend Cents operator+(const Cents &c1, const Cents &c2);
};
Cents operator+(const Cents &c1, const Cents &c2) {..}

class Cents
{
  ..
  int getCents() const { return m_cents; }
};
// note: this function is not a member function nor a friend function!
Cents operator+(const Cents &c1, const Cents &c2)
{
  // use the Cents constructor and operator+(int, int)
  // we don't need direct access to private members here
  return Cents{ c1.getCents() + c2.getCents() };
}

class Cents
{
    Cents operator+(int value);
};
// Left operand is implicitly assumed as this -> due to this >> or << cannot be overloaded
Cents Cents::operator+(int value)
{
    return Cents(m_cents + value);
}
```

6. Overloading operator<< is similar to overloading operator+ (they are both binary operators), except that the parameter

types are different.

```
friend std::ostream& operator<< (std::ostream &out, const Point &point);
friend std::istream& operator>> (std::istream &in, Point &point)

std::ostream& operator<< (std::ostream &out, const Point &point)
{
    // Since operator<< is a friend of the Point class, we can access Point's members directly.
    out << "Point(" << point.m_x << ", " << point.m_y << ", " << point.m_z << ')'; // actual output done here

    return out; // return std::ostream so we can chain calls to operator<<
}
std::istream& operator>> (std::istream &in, Point &point)
{
    // Since operator>> is a friend of the Point class, we can access Point's members directly.
    // note that parameter point must be non-const so we can modify the class members with the input values
    in >> point.m_x;
    in >> point.m_y;
    in >> point.m_z;

    return in;
}

std::cout << point1 << '\n';
std::cin >> point;
```

If the operator is <<, what are the operands? The left operand is the std::cout object, and the right operand is your Point class object. std::cout is actually an object of type std::ostream

7. Overloading prefix & postfix increment

I

```
Digit& Digit::operator++()
{
    ++m_digit;
    return *this;
}
```

C++ uses a "dummy variable" or "dummy argument" for the postfix operators. This argument is a fake integer parameter that only serves to distinguish the postfix version of increment/decrement from the prefix version

```
// returns a copy of original this with variable not incremented while original obj is incrmented
Digit Digit::operator++(int)
{
    // Create a temporary variable with our current digit
    Digit temp{*this};

    // Use prefix operator to increment this digit
    ++(*this); // apply operator

    // return temporary result
    return temp; // return saved state
}
```

8. Overloading subscript[]

Make sure you're not trying to call an overloaded operator[] on a pointer to an object. Compiler thinks aptr[x] is pointer to Aclass object

```
Class Aclass{
private:
int m_array[10];

int& operator[] (int index)
{
     return this->m_array[index];
}
};
Aclass aobj{10, 12}

std::cout << aobj[0];
aobj[1] = 2;
```

9. Overloading typecasts

```
class Cents
{
private:
    int m_cents;
    ..
    // Overloaded int cast
    operator int() const { return m_cents; }
     // No arguments, <space> between operator keyword and operator int(), No return type--> returntype has to be converted type
    // Allow us to convert Dollars into Cents
    operator Cents() const { return Cents(m_dollars * 100); }
};
```

This has implicit this operator… this-> will be used inside.. Calling will be like int(object)…

10. Overloading = .. Compiler provides default copy assignment member -> shallow copy

```
class Fraction
{
    // Overloaded assignment
    Fraction& operator= (const Fraction &fraction);

};

// A simplistic implementation of operator= (see better implementation below)
Fraction& Fraction::operator= (const Fraction &fraction)
{
    // do the copy
    m_numerator = fraction.m_numerator;
    m_denominator = fraction.m_denominator;

    // return the existing object so we can chain this operator
    return *this;
}

int main()
{
    Fraction fiveThirds(5, 3);
    Fraction f;
    f = fiveThirds; // calls overloaded assignment
}
```

11. Overloading paranthesis -> functors - classes that can be used as **functions but with storage**

Better option than using function with static variables0

```
class Accumulator
{
private:
    int m_counter{ 0 };

public:
    int operator() (int i) { return (m_counter += i); }
};

int main()
{
    Accumulator acc{};
    std::cout << acc(10) << '\n'; // prints 10
    std::cout << acc(20) << '\n'; // prints 30

    return 0;
}
```

12. F

I

13. D

I

14. F

I

15. F

I

16. F

I

17. F

18. F

I

19. F

I