

Operator Overloading

Jonathan Windle

University of East Anglia

J.Windle@uea.ac.uk

May 29, 2017

Overview I

- 1 Intro
- 2 Operator overloading
 - Method Example
 - Function Example
 - More on operators
 - Conversions
- 3 Conversions - Extended
- 4 Relational Operators
- 5 Overloading Streams
 - Ostream
 - IStream
 - Stream Status Flags
- 6 Files
 - Output
 - Input

- **Redefine function of operators for specific types.**
- No more complicated than writing function version.
- Create methods with specific names.
- Compiler uses operand types to determine correct method/function.

Method Example

```
class Fraction
{
private:
    int numerator, denominator;
public:
    // Begin with operator, redefine behaviour of *
    // Redefine behaviour for two Fraction operands
    Fraction operator*(Fraction& rf) {
        Fraction result(0,1);

        result.numerator = numerator*rf.numerator;
        result.denominator = denominator*rf.
            denominator;

        return result;
    }
}
```

- This is a **method** version of overloading.
- Is a **method** because it belongs to a class and takes one argument as the right side operand.

Function Example

```
class Fraction
{
private:
    int numerator, denominator;
public:
    // Accessors don't break encapsulation
    // inline to avoid method call overhead
    inline int getNumerator() const {
        return numerator;
    }
    inline int getDenominator() const {
        return denominator;
    }
};

// Minimise amount of code that directly depends on internal implementation.
inline Fraction operator*(const Fraction& lf, const Fraction& rf) {
    int numerator, denominator;

    numerator = lf.numerator*rf.numerator;
    denominator = lf.denominator*rf.denominator;

    return Fraction(numerator, denominator);
}
```

- Passing by reference is more efficient, but provide guarantee object can **not** be modified by method by using **const**. Limits lines of code that can modify a variable, makes debugging/maintenance easier.

More on operators

- Can overload operators for different types, e.g.
Fraction `operator*(int lh, const Fraction& rh)`.
- Allows integer multiplication with a Fraction. This only allows `i*frac` and not `frac*i`.
- If implemented one way, best to have both to support `frac*i` too.
- Another way is to use conversions...

Conversions

- Constructors define conversions:
- Add constructor that takes a single int:

```
Fraction(int n) {  
    numerator = n;  
    denominator = 1;  
}
```

- Use the single operator definition of

```
Fraction operator*(const Fraction& lh, const Fraction&  
rh).
```

- If this is in place, the compiler implicitly casts the integer to a Fraction when either `i*frac` or `frac*i` is called.

Conversions - Extended

- Can convert to other types using:

```
operator double() const {  
    return numerator/static_cast<double>(denominator);  
}
```

- Can then easily convert to double and do things like:

```
// frac implicitly converted to double.  
double d = frac/3.0;
```

- ISSUE! Now two ways of implementing `i*frac`:
 - Convert `i` into a fraction and use overload as before...
 - Convert `frac` into a double and use build in `*`.
 - Ambiguity like this can stop it compiling.
- Can stop this issue by using the `explicit` keyword in front of the constructor. This stops implicit conversions from taking place.
- Can still be used as an explicit conversion using `static_cast<Fraction>(4);`

Relational Operators

- Often good to implement relational operators e.g:

```
inline bool operator == (const Fraction& f, const Fraction& g) {  
    return f.getNumerator()*g.getDenominator() == g.  
        getNumerator()*f.getDenominator();  
}
```

- With one defined, often easier to define other operators with regards to the original.

```
inline bool operator != (const Fraction& f, const Fraction& g) {  
    return !(f==g);  
}
```

Overloading OStreams

```
inline ostream& operator << (ostream& str, const Fraction& f) {  
    return str << f.getNumerator() << "/" << f.getDenominator()  
    ;  
}
```

- Returns a reference to an ostream
- Takes in an ostream but not as const as it changes.
- All output streams are subclasses of ostream.
- Allows output to any output stream.

Overloading IStreams

```
inline istream& operator >> (istream& str, Fraction& f) {  
    char c;  
    int numerator, denominator;  
    if(str >> numerator >> c >> denominator) {  
        if(c == '/')  
            f = Fraction(numerator, denominator);  
        else  
            str.clear(ios_base::failbit);  
    }  
    return str;  
}
```

- Overload istreams for stream based input.
- Takes in a reference to the object to alter, in this case Fraction f.
- Using ios_base::failbit makes the stream handle errors gracefully.

Stream Status Flags

- Streams have `condition states` to which evaluate to true if successful.
- Define symbolic constants for each bit:
 - `stream::badbit` - unrecoverable error.
 - `stream::failbit` - recoverable error.
 - `stream::eofbit` - stream has reach the end of file.
 - `stream::goodbit` - stream operated correctly.
- Can `enquire about` states e.g. `s.bad()` - returns true if fail or bad bits set.
- can `modify using` `s.clear(flags)` to reset all states apart from those specified.
- Can set `specific status` bits using `s.setstate(flags)`.

Output file

```
ofstream os("file.txt", ofstream::out);
if (os) {
    for(int i = 1; i <= 12; i++){
        for(int j = 1; j <= 12; j++) {
            os << setw(2) << i*j << " ";
        }
        os << endl;
    }
    os.close();
}
else {
    cerr << "Error" << endl;
}
```

Input file

```
ifstream is("file.txt", ifstream::in);
if (is) {
    for(int i = 1; i <= 12; i++){
        for(int j = 1; j <= 12; j++) {
            int n;
            is >> n;
            cout << setw(3) << n << " ";
        }
        cout << endl;
    }
    is.close();
}
else {
    cerr << "Error" << endl;
}
```

The End