ST.CHARLES
COMMUNITY COLLEGE

CPT-182 - Programming in C++

Module 7

Overloading Operators in User-Defined Classes

Dayu Wang

- **Operators**

Operator Type	Operands Taken	Example
Unary Operator	**1** Operand	**!** (Prefix) **++** (Prefix, Postfix) **−−** (Prefix, Postfix)
Binary Operator	**2** Operands	**+** (Arithmetic Addition) **−** (Arithmetic Subtraction) **<=** (Comparison)
Ternary Operator	**3** Operands	**? :** (Conditional)

→ **Operators behave differently for different data types of operands.**

[Example 1] A / B

1) **If both A and B are integers, what is the behavior of operator '/'?**

   Integer Division

2) **If A is an integer and B is a double, what is the behavior of '/'?**

   Floating-Point Division

[Example 2] A + B

1) **If both A and B are integers, what is the behavior of operator '+'?**

   Arithmetic Addition

2) **If both A and B are strings, what is the behavior of operator '+'?**

   String Concatenation

- **Let's take the `Fraction` class as an example.**

→ **Data fields**

In "Fraction.h"

```
private:
    // Data fields
    int numerator;
    unsigned int denominator;
```

Mathematically, to negate a fraction, we can either negate the denominator or the numerator.

[Example] Negation of $\frac{3}{5}$

$$\frac{-3}{5} = \frac{3}{-5} = -\frac{3}{5}$$

In the `Fraction` class, let's set `denominator` to be always positive. Then,

1) If `numerator` is positive, then the entire fraction is positive.

2) If `numerator` is negative, then the entire fraction is negative.

3) If `numerator` is 0, then the entire fraction is 0.

→ **Constructor**

In "Fraction.h"

```
public:
    // Constructor
    Fraction(int = 0, unsigned int = 1);
```

In "Fraction.cpp"

```
Fraction::Fraction(int numerator, unsigned int denominator) :
    numerator(numerator), denominator(denominator) {}
```

Since mathematically the denominator of a fraction cannot be 0, we set it to 1 by default.

This constructor is not completed yet.

Later, we will add code in it.

→ **The `.print()` function**

In "Fraction.h"

```
// Prints the fraction to an output stream.
void print(ostream&) const;
```

In "Fraction.cpp"

```
void Fraction::print(ostream& out) const {
    out << numerator << " / " << denominator << endl;
}
```

→ **Mathematically, two fractions can be added together.**

$$\frac{1}{2}+\frac{3}{10}=\frac{5}{10}+\frac{3}{10}=\frac{8}{10}=\frac{4}{5}$$

Can we use the '+' operator in C++ to add two `Fraction` objects?

In "Main.cpp"

```cpp
int main() {
    Fraction f_1(1, 2), f_2(3, 10);
    Fraction f_3 = f_1 + f_2;
    f_3.print(cout);
    system("pause");
    return 0;
}
```

Code does not compile.

Console

```
binary '+': 'Fraction' does not define this operator or a
conversion to a type acceptable to the predefined operator
```

Compiler knows how to use '+' to add two integers; the compiler knows how to use '+' to add two doubles.

However, the compiler does not know how to use '+' to add two `Fraction` objects, because `Fraction` is a user-defined class.

If you want '+' to be used to add two `Fraction` objects, you need to "tell" the compiler how to add two `Fraction` objects.

→ **How (and where) to tell the compiler the behavior of operator '+' for two `Fraction` objects?**

We need to define the operator function for operator '+' in class `Fraction`.

We call this overloading operator '+' in the `Fraction` class.

Syntax of overloading an operator

```
[return_type]·operator·[operator_symbol]([arguments]);
```

→ **Overloading operator '+' for "`Fraction + Fraction`" in class `Fraction`**

1) **What should be the return type?**

   Since the addition result of two fractions is another fraction, the return type should be `Fraction`.

2) **How many arguments? What is the data type of each argument?**

   There should be only one argument, which is a `Fraction` type.

→ **Here, the operator function is a class-member function.**

"`f1 + f2`" should be understood as "`f1.+(f2)`".

`f1` (calling object) is already a `Fraction`, so we only need to pass another `Fraction` (only one) to the function.

In "Fraction.h"

```
Fraction operator + (const Fraction&) const;
```

1) Here, the argument should be passed by const reference, since it will **not** be changed in the function.

2) The function should be a const function, since the function does **not** change the current object.

3) In other words, in calculation of "A + B", **neither** A **nor** B will change.

→ **Before we implement the operator function, we need to define the simplify() function to simply a fraction to lowest terms.**

How to simplify a fraction (mathematically)?

$$\frac{12}{28} = \frac{3}{7}$$

1) Find the **greatest common divisor (GCD)** for the numerator and denominator.

2) Divide the numerator and denominator by the GCD.

→ **The algorithm of finding the GCD of two positive integers is not required in this chapter.**

This algorithm will be discussed in future lectures

In "Fraction.h"

```
private:
    // Finds the GCD of two positive integers.
    static unsigned int GCD(unsigned int, unsigned int);
```

This function is in the private section since it is not expected to be used outside the Fraction class.

This function is static since it is not for any specific Fraction object.

In "Fraction.cpp"

```
/** Finds the GCD of two positive integers.
    @return: GCD of the two positive integers
*/
unsigned int Fraction::gcd(unsigned int x, unsigned int y) {
    if (x < y) { return gcd(y, x); }
    if (x % y == 0) { return y; }
    else { return gcd(y, x % y); }
}
```

→ The `simplify()` function

In "Fraction.h"

```cpp
private:
    void simplify();  // Reduces the fraction to lowest terms.
```

In "Fraction.cpp"

```cpp
/** Reduces the fraction to lowest terms. */
void Fraction::simplify() {
    if (!numerator) {
        denominator = 1;
        return;
    }
    unsigned int GCD = gcd((unsigned int)abs(numerator), denominator);
    numerator /= GCD;
    denominator /= GCD;
}
```

→ Improving the implementation of the constructor

In "Fraction.cpp"

```cpp
Fraction::Fraction(int numerator, unsigned int denominator) :
    numerator(numerator), denominator(denominator) { simplify(); }
```

A `Fraction` object will be automatically simplified when it is created.

→ Implementing the operator function for operator '+'

Mathematically, how to add two fractions together?

$$\frac{a}{b} + \frac{c}{d} = \frac{ad}{bd} + \frac{bc}{bd} = \frac{ad + bc}{bd}$$

Finally, simplify the result.

In "Fraction.h"

```cpp
Fraction operator + (const Fraction&) const;
```

In "Fraction.cpp"

```cpp
/** Overloads operator '+' for "Fraction + Fraction".
    @param other: the other fraction to add to this fraction
    @return: addition result
*/
Fraction Fraction::operator + (const Fraction& other) const {
    Fraction result;
    result.denominator = denominator * other.denominator;
    result.numerator = numerator * other.denominator
                     + denominator * other.numerator;
    result.simplify();
    return result;
}
```

→ **Test in the `main()` function**

In "Main.cpp"

```
int main() {
    Fraction f_1(1, 2), f_2(3, 10);
    Fraction f_3 = f_1 + f_2;
    f_3.print(cout);
    system("pause");
    return 0;
}
```

Console

```
4 / 5
```

→ **Mathematically, we can add a fraction and an integer together.**

$$\frac{1}{2} + 3 = \frac{1}{2} + \frac{6}{2} = \frac{7}{2}$$

In "Main.cpp"

```
int main() {
    Fraction f_1(1, 2)
    Fraction f_2 = f_1 + 3;
    f_2.print(cout);
    system("pause");
    return 0;
}
```

Compiler only knows how to do "Fraction + Fraction".

It does not know how to do "Fraction + int".

→ **Overloading "Fraction + int".**

In "Fraction.h"

```
Fraction operator + (const Fraction&) const;  // "Fraction + Fraction".
Fraction operator + (int) const;  // Overloads "Fraction + int".
```

In "Fraction.cpp"

```
/** Overloads operator '+' for "Fraction + int".
    @param other: the integer to add to this fraction
    @return: addition result
*/
Fraction Fraction::operator +(int other) const {
    return *this + Fraction(other, 1);
}
```

"`*this`" is a reference to the current object (`Fraction`).

"`Fraction(other, 1)`" converts an integer to a `Fraction`.

Now, the addition becomes "`Fraction + Fraction`" which has already been overloaded.

→ **How about "int + Fraction"?**

This is totally a different story.

"f1 + f2" should be understood as "f1.+(f2)".

Since the operator function is a class-member function in the Fraction class, only a Fraction object can call this function.

In other words, "f1" must be a Fraction object.

→ **Therefore, if we want to overload "int + Fraction", we cannot write the function in the Fraction class anymore.**

We have to write the function outside the Fraction class.

In "Fraction.h"

```cpp
class Fraction { ··· };

// Non-class-member functions

// Overloads "int + Fraction".
Fraction operator + (int, const Fraction&);
```

In "Fraction.cpp"

```cpp
/** Overloads operator '+' for "int + Fraction". */
Fraction operator + (int left, const Fraction& right) {
    return right + left;
}
```

In "Fraction.cpp"

```cpp
/** Overloads operator '+' for "int + Fraction".
    @param left: left operand (integer)
    @param right: right operand (fraction)
    @return: addition result
*/
Fraction operator + (int left, const Fraction& right) {
    return right + left;
}
```

Why this time, there are 2 arguments?

Because the function is no longer a ~~class-member function~~, which means that we do not have a Fraction object to call this function.

Therefore, we need 2 arguments.

→ **The '+' operator does not always mean "arithmetic addition".**

e.g., +3, +12

In "Fraction.h"

```cpp
Fraction operator + () const;  // Overloads unary '+' operator.
```

In "Fraction.cpp"

```cpp
/** Overloads the unary '+' operator.  @return: this fraction */
Fraction Fraction::operator + () const { return *this; }
```

- **Overloading Operator '−'**

→ Unary operator '−'

e.g., −3, −12

In "Fraction.h"

```cpp
Fraction operator - () const;  // Overloads the unary '-' operator.
```

In "Fraction.cpp"

```cpp
/** Overloads the unary '-' operator.
    @return: opposite fraction
*/
Fraction Fraction::operator - () const {
    return Fraction(-numerator, denominator);
}
```

→ Arithemetic subtraction operator '−'

3 − 5 = 3 + (−5)

10 − (−15) = 10 + 15

"Subtracting" is equivalent to "adding the opposite".

We can use operator '+' (already overloaded) to implement operator '−'.

→ Overloading "Fraction − Fraction".

In "Fraction.h"

```cpp
Fraction operator - (const Fraction&) const;  // "Fraction - Fraction"
```

In "Fraction.cpp"

```cpp
/** Overloads "Fraction - Fraction".
    @param other: the other fraction to subtract from this fraction
    @return: subtraction result
*/
Fraction Fraction::operator - (const Fraction& other) const {
    return *this + (-other);
}
```

→ Overloading "Fraction − int".

In "Fraction.h"

```cpp
Fraction operator - (int) const;  // Overloads "Fraction - int".
```

In "Fraction.cpp"

```cpp
/** Overloads "Fraction - int".
    @param other: the integer to subtract from this fraction
    @return: subtraction result
*/
Fraction Fraction::operator - (int other) const {
    return *this + (-other);
}
```

→ Overloading "int − Fraction"

In "Fraction.h"

```
class Fraction {
    ...
};

// Non-class-member functions

// Overloads "int - Fraction".
Fraction operator - (int, const Fraction&);
```

In "Fraction.cpp"

```
/** Overloads "int - Fraction".
    @param left: left operand (integer)
    @param right: right operand (fraction)
    @return: subtraction result
*/
Fraction operator - (int left, const Fraction& right) {
    return Fraction(left, 1) - right;
}
```

- **Overloading Arithmetic Multiplication Operator '*'**

In "Fraction.h"

```
class Fraction {
...
    Fraction operator * (const Fraction&) const;  // Fraction * Fraction
    Fraction operator * (int) const;  // "Fraction * int"
...
};
// Non-class-member functions
Fraction operator * (int, const Fraction&);  // "int * Fraction".
```

In "Fraction.cpp"

```
Fraction Fraction::operator * (const Fraction& other) const {
    Fraction result;
    result.numerator = numerator * other.numerator;
    result.denominator = denominator * other.denominator;
    result.simplify();
    return result;
}
Fraction Fraction::operator * (int other) const {
    return *this * Fraction(other, 1);
}
Fraction operator * (int left, const Fraction& right) {
    return right * left;
}
```

- **Overloading Arithmetic Division Operator '/'**

→ The reciprocal() function

In "Fraction.h"

```
private:
    // Returns the reciprocal of the fraction.
    Fraction reciprocal() const;
```

In "Fraction.cpp"

```
/** Returns the reciprocal of the fraction.
    @return: reciprocal of the fraction
*/
Fraction Fraction::reciprocal() const {
    int sign = numerator > 0 ? 1 : -1;
    unsigned int new_denominator = (unsigned int)std::abs(numerator);
    int new_numerator = (int)denominator * sign;
    return Fraction(new_numerator, new_denominator);
}
```

Here, for simplicity, we just ignore the ~~divide-by-zero issues~~. We just assume that this will not happen.

→ Overloading the '/' operator

"Divide by" is equivalent to "times the reciprocal".

In "Fraction.h"

```
class Fraction {
...
    Fraction operator / (const Fraction&) const;  // Fraction / Fraction
    Fraction operator / (int) const;  // "Fraction / int"
...
};
// Non-class-member functions
Fraction operator / (int, const Fraction&);  // "int / Fraction"
```

In "Fraction.cpp"

```
Fraction Fraction::operator / (const Fraction& other) const {
    return *this * other.reciprocal();
}

Fraction Fraction::operator / (int other) const {
    return *this / Fraction(other, 1);
}

Fraction operator / (int left, const Fraction& right) {
    return Fraction(left, 1) / right;
}
```

- **Compound Assignment Operators**

In "Fraction.h"

```
Fraction operator += (const Fraction&);  // "Fraction += Fraction"
Fraction operator += (int);  // "Fraction += int"
Fraction operator -= (const Fraction&);  // "Fraction -= Fraction"
Fraction operator -= (int);  // "Fraction -= int"
Fraction operator *= (const Fraction&);  // "Fraction *= Fraction"
Fraction operator *= (int);  // "Fraction *= int"
Fraction operator /= (const Fraction&);  // "Fraction /= Fraction"
Fraction operator /= (int);  // "Fraction /= int"
```

Why they are not const functions?

"a += b", b is not changed; a is changed.

Why there are no "int += Fraction" (and so on) versions?

"a += b", a is an integer. "a += b" is equivalent to "a = a + b", so after the assignment, a becomes a Fraction.

In C++, this is impossible (static typing).

In "Fraction.cpp"

```
Fraction Fraction::operator += (const Fraction& other) {
    return *this = *this + other;
}
Fraction Fraction::operator += (int other) {
    return *this = *this + other;
}
Fraction Fraction::operator -= (const Fraction& other) {
    return *this = *this - other;
}
Fraction Fraction::operator -= (int other) {
    return *this = *this - other;
}
Fraction Fraction::operator *= (const Fraction& other) {
    return *this = *this * other;
}
Fraction Fraction::operator *= (int other) {
    return *this = *this * other;
}
Fraction Fraction::operator /= (const Fraction& other) {
    return *this = *this / other;
}
Fraction Fraction::operator /= (int other) {
    return *this = *this / other;
}
```

- **Increment Operator "++" and Decrement Operator "−−"**

→ **Since prefix increment/decrement and postfix increment/decrement have different behaviors, we need to overload respectively for prefix increment/decrement and postfix increment/decrement.**

In "Fraction.h"

```
Fraction& operator ++ ();  // Prefix increment
Fraction operator ++ (int);  // Postfix increment
Fraction& operator -- ();  // Prefix decrement
Fraction operator -- (int);  // Postfix decrement
```

1) **Prefix increment/decrement operator returns a reference of the `Fraction` after increment/decrement.**

2) **Postfix increment/decrement operator returns a copy of the `Fraction` before increment/decrement.**

3) **For prefix increment/decrement operator, there is nothing in the "`()`".**

4) **For postfix increment/decrement operator, the "`(int)`" means nothing but tells the compiler that this is postfix operator.**

In "Fraction.cpp"

```
/** Overloads "++Fraction". */
Fraction& Fraction::operator ++ () {
    *this += 1;
    return *this;
}
/** Overloads "Fraction++". */
Fraction Fraction::operator ++ (int) {
    Fraction result = *this;
    ++*this;
    return result;
}
/** Overloads "--Fraction". */
Fraction& Fraction::operator -- () {
    *this -= 1;
    return *this;
}
/** Overloads "Fraction--". */
Fraction Fraction::operator -- (int) {
    Fraction result = *this;
    --*this;
    return result;
}
```

- **Comparison Operators**

1 operator < (2 other) 3;

1) What should be the return type?

2) "Fraction&" or "const Fraction&"?

3) const function or not?

In "Fraction.h"

```
bool operator < (const Fraction&) const;  // "Fraction < Fraction"
bool operator < (int) const;  // "Fraction < int"
bool operator <= (const Fraction&) const;  // "Fraction <= Fraction"
bool operator <= (int) const;  // "Fraction <= int"
bool operator > (const Fraction&) const;  // "Fraction > Fraction"
bool operator > (int) const;  // "Fraction > int"
bool operator >= (const Fraction&) const;  // "Fraction >= Fraction"
bool operator >= (int) const;  // "Fraction >= int"
bool operator == (const Fraction&) const;  // "Fraction == Fraction"
bool operator == (int) const;  // "Fraction == int"
bool operator != (const Fraction&) const;  // "Fraction != Fraction"
bool operator != (int) const;  // "Fraction != int"
```

- **Can we use the stream insertion operator ("<<") to write a Fraction object to the console?**

In "Main.cpp"

```
cout << Fraction(1, 2) << endl;  // Code does not compile.
```

Compiler does not know how to output a Fraction object, since it is a user-defined class.

You need to overload the "<<" operator in the Fraction class to tell the compiler how to output a Fraction object.

→ What is the left operand of the "<<" operator?

It is an output stream.

"cout" and "fout" are both output streams.

Therefore, "<<" is a non-class-member function.

→ What is the return type of the "<<" operator?

```
cout << b << c
```

Must return an output stream for the next evaluation.

In "Fraction.h"

```
friend ostream& operator << (ostream&, const Fraction&);
```

- **Overloading the "<<" operator**

→ **Suppose the output format of `Fraction(2, 3)` is "`2 / 3`".**

In "Fraction.cpp"

```
ostream& operator << (ostream& out, const Fraction& f) {
    out << f.numerator << " / " << f.denominator;
    return out;
}
```

1) Return type should be `ostream`, not ~~`ofstream`~~.

2) Do not forget that you need to return the `ostream` at the end.

- **Overloading the ">>" operator**

→ **Suppose the input format is "`[numerator] [denominator]`".**

In "Fraction.h"

```
friend istream& operator << (istream&, Fraction&);
```

In "Fraction.cpp"

```
istream& operator >> (istream& in, Fraction& f) {
    in >> f.numerator >> f.denominator;
    f.simplify();
    return in;
}
```

- **Overloading the subscript operator "`[]`"**

→ **You must overload the operator "`[]`" twice.**

lvalue: `arr[3] = 3`

rvalue: `my_var = arr[3]`

- **The lvalue returns a reference.**
- **The rvalue returns a `const` reference.**

In "Simple_String.h"

```
class Simple_String {
public:
    Simple_String();  // Default constructor
    void push_back(char);  // Appends a character to the end.
    char& operator [] (size_t);  // lvalue
    const char& operator [] (size_t) const;  // rvalue
private:
    char c[100];  // Stores the characters in the string.
    size_t size;  // Stores the number of characters in the string.
};
```

In "Simple_String.cpp"

```
// lvalue
char& Simple_String::operator [] (unsigned int index) {
    return c[index];
}
// rvalue
const char& Simple_String::operator [] (unsigned int index) const {
    return c[index];
}
```

→ How can compiler know which one (reference or `const` reference) to return?

Assignment statement: **A = B**

1) Left side of the assignment operator is a reference.

2) Right side of the assignment operator is a `const` reference.

- In this lecture, we overloaded 48 operators.

→ The following two operators will be discussed in future lectures:

▪ Assignment operator '='

▪ Dereferencing operators, '*' (unary) and '->'.

- Restrictions of Overloading Operators

→ You cannot ~~change the precedence~~ of an operator.

→ The associativity cannot be changed.

For example, the associativity of the arithmetic addition operator ('+') is from left to right, and it cannot be changed.

→ Default parameter values cannot be used in operator functions.

→ You cannot change the ~~number of parameters~~ an operator takes.

→ You cannot ~~create new operators~~.

Only existing operators can be overloaded.

→ Some operators cannot be overloaded.

For example: '.', '::', '? :'

→ The meaning of how an operator works with built-in types, such as `int`, remains the same.

You cannot redefine how operators work with built-in data types.