fit@hcmus

Object-Oriented Programming

Operator Overloading

Tran Duy Hoang

Topics Covered

- Operator Function
- Spectial Operator
- Friend Function
- The Big Three

Topics Covered

- **Operator Function**
- Spectial Operator
- Friend Function
- The Big Three

Operator Function

- A special function defining how an **operator** behave for **user-defined types**

```
// built-in types

int a, b, c;
float x;

a = b + c;
a = b + x;
if (a > b) …
if (a > x) …
```

```
// user-defined types

Fraction frac1, frac2, frac3;
int a;

frac1 = frac2 + frac3;
frac1 = frac2 + a;
if (frac1 > frac2) …
if (frac1 > a) …
```

Types of Operator Functions

- Member Functions (inside the class)
- Non-Member Functions (outside the class)

```
// member function

Fraction Fraction::operator+(Fraction frac) {}

int main()
{
    Fraction frac1, frac2;
    Fraction frac3 = frac1 + frac2;
    // ~ Fraction frac3 = frac1.add(frac2);
}
```

```
// non-member function

Fraction operator+(Fraction frac1, Fraction frac2) {}

int main()
{
    Fraction frac1, frac2;
    Fraction frac3 = frac1 + frac2;
    // ~ Fraction frac3 = add(frac1, frac2);
}
```

Types of Operators

N-nary	Group	Operator
Unary	Inc / Dec	++, --
	Math sign	+, -
	Bit	!, ~
	Pointer	*, &
	Type-cast	int, float, double, …
Binary	Arithmetic	+, -, *, /, %
	Comparison	>, <, ==, >=, <=, !=
	Logic	&&, ||, &, |
	Input / Output	<<, >>
	Assignment	=, +=, -=, *=, /=, %=
	Array indexing	[ ]

Syntax for Operator Overloading

return_type operator op (parameters);

- Return Type: The type of value the function returns
- operator op: The keyword “operator” followed by the “operator”
- Parameters: The operands
  - Member function: Fraction Fraction::operator+(Fraction frac) {}
  - Non-member function: Fraction operator+(Fraction frac1, Fraction frac2) {}

The number of arguments in an operator overloading function depends on whether the operator is **unary** or **binary**, and whether the function is a **member function** or a **non-member function**.

Example: Operators

```
class Fraction
{
private:
    int num;
    int denom;
public:
    Fraction operator+(Fraction frac);
    Fraction operator+(int a);
    bool operator==(Fraction frac);
    bool operator==(int a);
};
```

```
Fraction Fraction::operator+(Fraction frac) {}

Fraction Fraction::operator+(int a) {}

bool Fraction::operator==(Fraction frac) {}

bool Fraction::operator==(int a) {}
```

Rules and Limitations

- Cannot create new operators
- Cannot overload operators for built-in types
- Cannot change the number of operands
- Cannot change operator precedence and associativity
- Some operators cannot be overloaded
  - E.g., :: (scope resolution), . (member access),…

Topics Covered

- Operator Function
- **Spectial Operator**
- Friend Function
- The Big Three

Assignment Operator

- Used to copy values from one object to another
- Can be overloaded to define custom behavior
- C++ provides an implicit assignment operator
  - which performs a shallow copy
- Overloading the assignment operator
  - avoid shallow copying issues
- Copy Constructor vs. Assignment Operator

Assignment Operator Overloading

- Why return Array&?
- Self-assignment check
- Deletes old memory
- Performs deep copy

```
Array& Array::operator=(const Array& arr)
{
    if (this != &arr)                    // Avoid self-assignment
    {
        delete data;                     // Free old memory
        data = new int[arr.length];      // Deep copy
        // copy elements
    }

    return *this;
}
```

Assignment Operator Overloading

- Compound assignment operators (+=, -=, *=, /=, %=)

```
Fraction& Fraction::operator+=(const Fraction& frac);
Fraction& Fraction::operator+=(const int& a);
```

Increment & Decrement Operators

Operator	Type	Behavior
`++obj`	**Pre-increment**	Increases the value **before** returning.
`obj++`	**Post-increment**	Returns the value **before** increasing it.
`--obj`	**Pre-decrement**	Decreases the value **before** returning.
`obj--`	**Post-decrement**	Returns the value **before** decreasing it.

Increment & Decrement Operators

- Overloading Pre-Increment (++obj)
  - does not take any parameters

```
Fraction& Fraction::operator++()
{
    num += denom;
    return *this;
}
```

- Overloading Post-Increment (obj++)
  - takes an unused int parameter

```
Fraction Fraction::operator++(int)
{
    Fraction temp = *this;
    num += denom;
    return temp;
}
```

Topics Covered

- Operator Function
- Spectial Operator
- **Friend Function**
- The Big Three

Friend Function

- Does not belong to the class but can access its private data

- Syntax of a Friend Function
  - declared inside class using *friend* keyword
  - implemented outside class (not use *friend*)

Example: Friend Function

```
// declaration
class Fraction
{
    friend ostream& operator<<(ostream& os, const Fraction& frac);
};

// implementation
ostream& operator<<(ostream& os, const Fraction& frac)
{
    os << frac.num << "/" << frac.denom;
    return os;
}
```

Topics Covered

- Operator Function
- Spectial Operator
- Friend Function
- **The Big Three**

The Big Three Rule

- C++ provides default versions:
  - The default destructor does nothing
  - The default copy constructor performs shallow copy
  - The default assignment operator performs shallow copy

- The Big Three Rule state that
  - If a class allocates resources dynamically, then it should implement all three: Destructor, Copy Constructor, Assignment Operator.

The Big Three Rule

- Default Destructor Problem
  - Class has pointer attribute and memory allocation
  - Default destructor does not de-allocate memory

→ Implement destructor explicitly to de-allocate memory

Example: Destructor

```
class Array
{
private:
    int length;
    int* data;
public:
    Array(int length);
    ~Array();
};

Array::Array(int length)
{
    length = length;
    data = new int[length];
}
```

```
Array::~Array()
{
    delete[] this->data;
}
```

The Big Three Rule

- Default Copy Constructor Problem
  - Class has pointer attribute and memory allocation
  - Default copy constructor performs shallow copy

→ Implement copy constructor explicitly to performs deep copy

Example: Copy Constructor

```
class Array
{
private:
    int length;
    int* data;
public:
    Array(int length);
    Array(const Array &arr);
};

Array::Array(int length)
{
    length = length;
    data = new int[length];
}
```

```
Array::Array(const Array &arr)
{
    length = arr.length;
    data = new int[arr.length];
    for (int i = 0; i < length; i++)
        data[i] = arr.data[i];
}
```

The Big Three Rule

- Default Assignment Operator Problem
  - Class has pointer attribute and memory allocation
  - Default assignment operator
    - does not de-allocate memory
    - performs shallow copy

→ Implement assignment operator explicitly to
  - de-allocate memory
  - performs deep copy

Example: Assignment Operator

```
class Array
{
private:
    int length;
    int* data;
public:
    Array(int length);
    Array& operator=(const Array& arr);
};

Array::Array(int length)
{
    length = length;
    data = new int[length];
}
```

```
Array& Array::operator=(const Array& arr)
{
    if (this != &arr)
    {
        delete data;
        data = new int[arr.length];
        for (int i = 0; i < length; i++)
            data[i] = arr.data[i];
    }
    return *this;
}
```

Exercise 4.1

- Provide class Fraction the following operators:
  - Arithmetic: +, -, *, /
  - Comparation: >, <, ==, >=, <=, !=
  - Assignment: =, +=, *=
  - Increment & Decrement: ++, --
  - Type-cast: (float), (int)
  - Input & Output: >>, <<

Exercise 4.2

- Provide class Array (element of int) the following operators:
  - Concatenation: +
    - E.g., {1,2,3} + {4,5} = {1,2,3,4,5}
  - Assignment: =, +=
  - Array indexer: []
  - Type-case: (int*)
    - E.g., Array array; int* data = (int*) array;
  - Input & Output: >>, <<