CS100 Lecture 21

Contents

Basics

Operator overloading: Provide the behaviors of **operators** for class types.

- At least one operand should be a class type. Modifying the behavior of operators on built-in types is not allowed.

  ```
  int operator+(int, int); // Error.
  ```

- Inventing new operators is not allowed.

  ```
  double operator**(double x, double exp); // Error.
  ```

- Overloading does not modify the **associativity** and **precedence**.

  ```
  std::cout << a + b; // Equivalent to std::cout << (a + b)
  ```

Basics

Some operators cannot be overloaded:

`obj.mem`, `::`, `?:`, `obj.*memptr` (not covered in CS100)

Some operators can be overloaded, but are strongly not recommended:

`cond1 && cond2`, `cond1 || cond2`

- Reason: There is no way to overload `&&` and `||` while preserving the **short-circuited** evaluation property.

Basics

We have already seen some:

- The **copy assignment operator** and the **move assignment operator** are two special overloads for `operator=`.
- The IOStream library provides overloaded `operator<<` and `operator>>` to perform input and output.
- The string library provides `operator+` for concatenation of strings, and `<`, `<=`, `>`, `>=`, `==`, `!=` for comparison in lexicographical order.
- Standard library containers and `std::string` have `operator[]`.
- Smart pointers have `operator*` and `operator->`.

Basics

Overloaded operators can be defined in two forms:

- as a member function, in which the leftmost operand is bound to `this`:
  - `a[i]` $\Leftrightarrow$ `a.operator[](i)`
  - `a = b` $\Leftrightarrow$ `a.operator=(b)`
  - `*a` $\Leftrightarrow$ `a.operator*()`
  - `f(arg1, arg2, arg3, ...)` $\Leftrightarrow$ `f.operator()(arg1, arg2, arg3, ...)`
- as a non-member function:
  - `a == b` $\Leftrightarrow$ `operator==(a, b)`
  - `a + b` $\Leftrightarrow$ `operator+(a, b)`

Example: `Rational`

A class for rational numbers

```cpp
class Rational {
  int m_num; // numerator
  unsigned m_denom; // denominator
  void simplify() {
    int gcd = std::gcd(m_num, m_denom); // std::gcd in <numeric> (since C++17)
    m_num /= gcd; m_denom /= gcd;
  }
 public:
  Rational(int x = 0) : m_num{x}, m_denom{1} {}
  Rational(int num, unsigned denom) : m_num{num}, m_denom{denom} { simplify(); }
  double to_double() const {
    return static_cast<double>(m_num) / m_denom;
  }
};
```

We want to support arithmetic operators for `Rational`.

`Rational`: arithmetic operators

A good way: define `operator+=` and the **unary** `operator-`, and then define other operators in terms of them.

- Define the arithmetic operators to use the compound assignment operators.

```
class Rational {
  friend Rational operator-(const Rational &);
 public:
  Rational &operator+=(const Rational &rhs) { /* ... */ }
  Rational &operator-=(const Rational &rhs) {
    return *this += -rhs;
  }
};
Rational operator+(const Rational &lhs, const Rational &rhs) {
  return Rational(lhs) += rhs;
}
Rational operator-(const Rational &lhs, const Rational &rhs) {
  return Rational(lhs) -= rhs;
}
```

`Rational`: arithmetic operators

```cpp
class Rational {
  friend Rational operator-(const Rational &);
 public:
  Rational operator+=(const Rational &rhs) {
    m_num = m_num * static_cast<int>(rhs.m_denom) // Be careful with unsigned
            + static_cast<int>(m_denom) * rhs.m_num;
    m_denom *= rhs.m_denom;
    simplify();
    return *this;
  }
};
```

`Rational`: arithmetic operators

```
class Rational {
  friend Rational operator-(const Rational &);
};
Rational operator-(const Rational &x) { // unary operator-, for "-x"
  return Rational(-x.m_num, x.m_denom);
}
```

A modern way:

```
Rational operator-(const Rational &x) {
  return {-x.m_num, x.m_denom};
}
```

`Rational`: relational operators

Define `<` and `==`, and define others in terms of them. (Before C++20)

- Since C++20: Define `==` and `<=>`, and the compiler will generate others.

A possible way: Use `to_double` and compare the floating-point values.

```cpp
bool operator<(const Rational &lhs, const Rational &rhs) {
  return lhs.to_double() < rhs.to_double();
}
```

- This does not require `operator<` to be a `friend`.

`Rational`: ralational operators

Another way (possibly better):

```cpp
class Rational {
  friend bool operator<(const Rational &, const Rational &);
  friend bool operator==(const Rational &, const Rational &);
};
bool operator<(const Rational &lhs, const Rational &rhs) {
  return static_cast<int>(rhs.m_denom) * lhs.m_num
        < static_cast<int>(lhs.m_denom) * rhs.m_num;
}
bool operator==(const Rational &lhs, const Rational &rhs) {
  return lhs.m_num == rhs.m_num && lhs.m_denom == rhs.m_denom;
}
```

If there are member functions to obtain the numerator and the denominator, these functions don't need to be `friend`.

`Rational`: relational operators

Define others in terms of `<` and `==`:

```
bool operator>(const Rational &lhs, const Rational &rhs) {
  return rhs < lhs;
}
bool operator<=(const Rational &lhs, const Rational &rhs) {
  return !(lhs > rhs);
}
bool operator>=(const Rational &lhs, const Rational &rhs) {
  return !(lhs < rhs);
}
bool operator!=(const Rational &lhs, const Rational &rhs) {
  return !(lhs == rhs);
}
```

`Rational`: arithmetic and relational operators

What if we define them (say, `operator==`) as member functions?

```
class Rational {
 public:
  Rational(int x = 0) : m_num{x}, m_denom{1} {}
  Rational operator==(const Rational &rhs) const {
    return m_num == rhs.m_num && m_denom == rhs.m_denom;
  }
};
```

`Rational`: arithmetic and relational operators

What if we define them (say, `operator==`) as member functions?

```cpp
class Rational {
 public:
  Rational(int x = 0) : m_num{x}, m_denom{1} {}
  Rational operator==(const Rational &rhs) const {
    return m_num == rhs.m_num && m_denom == rhs.m_denom;
  }
};
```

```cpp
Rational r = some_value();
if (r == 0) { /* ... */ } // OK, r.operator==(0)
                          // effectively r.operator==(Rational(0))
if (0 == r) { /* ... */ } // Error!
                          // 0.operator==(r) ???
```

`Rational`: arithmetic and relational operators

To allow implicit conversions on both sides, the operator should be defined as **non-member functions**.

```
Rational r = some_value();
if (r == 0) { /* ... */ } // OK, operator==(r, 0)
                          // effectively operator==(r, Rational(0))
if (0 == r) { /* ... */ } // OK, operator==(0, r)
                          // effectively operator==(Rational(0), r)
```

Relational operators

Define relational operators in a consistent way:

- `a != b` should mean `!(a == b)`
- `!(a < b)` and `!(a > b)` should imply `a == b`

Avoid abuse of relational operators:

```cpp
struct Point2d {
  double x, y;
};
bool operator<(const Point2d &lhs, const Point2d &rhs) {
  return lhs.x < rhs.x; // Is this the unique, best behavior?
}
```

`++` and `--`

`++` and `--` are often defined as **members**, because they modify the object.

To differentiate the postfix version `x++` and the prefix version `++x`: **The postfix version has a parameter of type `int`.**

- The compiler will translate `++x` to `x.operator++()`, `x++` to `x.operator++(0)`.

```cpp
class Rational {
 public:
  Rational &operator++() { ++m_num; return *this; }
  Rational operator++(int) { // This `int` parameter is not used.
    // The postfix version is almost always defined like this.
    auto tmp = *this;
    ++*this; // Make use of the prefix version.
    return tmp;
  }
};
```

`++` and `--`

```cpp
class Rational {
 public:
  Rational &operator++() { ++m_num; return *this; }
  Rational operator++(int) { // This `int` parameter is not used.
    // The postfix version is almost always defined like this.
    auto tmp = *this;
    ++*this; // Make use of the prefix version.
    return tmp;
  }
};
```

The prefix version returns reference to `*this`, while the postfix version returns a copy of `*this` before incrementation.

- Same as the built-in behaviors.

IO operators

Implement `std::cin >> r` and `std::cout << r`.

Input operator:

```
std::istream &operator>>(std::istream &, Rational &);
```

Output operator:

```
std::ostream &operator<<(std::ostream &, const Rational &);
```

- `std::cin` is of type `std::istream`, and `std::cout` is of type `std::ostream`.
- The left-hand side operand should be returned, so that we can write

  ```
  std::cin >> a >> b >> c; std::cout << a << b << c;
  ```

`Rational`: output operator

```cpp
class Rational {
  friend std::ostream &operator<<(std::ostream &, const Rational &);
};
std::ostream &operator<<(std::ostream &os, const Rational &r) {
  return os << r.m_num << '/' << r.m_denom;
}
```

If there are member functions to obtain the numerator and the denominator, it don't have to be a `friend`.

```cpp
std::ostream &operator<<(std::ostream &os, const Rational &r) {
  return os << r.getNumerator() << '/' << r.getDenominator();
}
```

`Rational`: input operator

Suppose the input format is `a b` for the rational number $\frac{a}{b}$, where `a` and `b` are integers.

```
std::istream &operator>>(std::istream &is, Rational &r) {
  int x, y; is >> x >> y;
  if (!is) { // Pay attention to input failures!
    x = 0;
    y = 1;
  }
  if (y < 0) { y = -y; x = -x; }
  r = Rational(x, y);
  return is;
}
```

Example: `Dynarray`

`operator[]`

```cpp
class Dynarray {
 public:
  int &operator[](std::size_t n) {
    return m_storage[n];
  }
  const int &operator[](std::size_t n) const {
    return m_storage[n];
  }
};
```

The use of `a[i]` will be translated into `a.operator[](i)`.

(C++23 allows `a[i, j, k]`!)

Other operators

Homework: Define `operator[]` and relational operators for `Dynarray`.

Example: `SharedPtr`

`SharedPtr`: indirection (dereference) operator

Recall the `SharedPtr` class we defined in previous lectures.

```
struct CountedObject {
  Object theObject;
  int refCnt;
};
class SharedPtr {
  CountedObject *m_ptr;
 public:
  Object &operator*() const { // Why should it be const?
    return m_ptr->theObject;
  }
};
```

We want `*sp` to return reference to the managed object.

`SharedPtr`: indirection (derefernce) operator

Why should `operator*` be `const`?

```
class SharedPtr {
  CountedObject *m_ptr;
 public:
  Object &operator*() const {
    return m_ptr->theObject;
  }
};
```

On a `const SharedPtr`, obtaining a non-`const` reference to the managed object may still be allowed.

- The (smart) pointer is `const`, but the managed object is not. ("top-level" `const`)

`SharedPtr`: member access through pointer

To make `operator->` in consistent with `operator*` (make `a->mem` equivalent to `(*a).mem`), `operator->` is almost always defined like this:

```
class SharedPtr {
 public:
  Object *operator->() const {
    return std::addressof(this->operator*());
  }
};
```

Why do we use `std::addressof(x)` instead of `&x`? - In case there is an overload for `operator&`!