CS100 Lecture 19

Operator Overloading

Contents

Basics

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

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)`

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: Since `x && y` would become `operator&&(x, y)` , there is no way to overload `&&` (or `||`) that preserves the **short-circuit evaluation** property.

Basics

- 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.
MyInt operator-(int, int); // Still error.
```

- Inventing new operators is not allowed.

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

- Overloading does not modify the **associativity**, **precedence** and the **operands' evaluation order**.

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

Example: `Rational`

A class for rational numbers

```
class Rational {
  int m_num;        // numerator
  unsigned m_denom; // denominator
  void simplify() { // Private, because this is our implementation detail.
    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} {} // Also a default constructor.
  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 have arithmetic operators supported for `Rational`.

`Rational`: arithmetic operators

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

```
class Rational {
  friend Rational operator-(const Rational &); // Unary `operator-` as in `-x`.
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; // `x += y` should return a reference to `x`.
  }
};
Rational operator-(const Rational &x) {
  return {-x.m_num, x.m_denom};
  // The above is equivalent to `return Rational(-x.m_num, x.m_denom);`.
}
```

`Rational`: arithmetic operators

Define the arithmetic operators in terms of the compound assignment operators.

```
class Rational {
public:
  Rational &operator-=(const Rational &rhs) {
    // Makes use of `operator+=` and the unary `operator-`.
    return *this += -rhs;
  }
};
Rational operator+(const Rational &lhs, const Rational &rhs) {
  return Rational(lhs) += rhs; // Makes use of `operator+=`.
}
Rational operator-(const Rational &lhs, const Rational &rhs) {
  return Rational(lhs) -= rhs; // Makes use of `operator-=`.
}
```

[Best practice] Avoid repetition.

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

The arithmetic operators for `Rational` are simple yet requires carefulness.

- Integers with different signed-ness need careful treatment.
- Remember to `simplify()`.

Fortunately, we only need to pay attention to these things in `operator+=`. Everything will be right if `operator+=` is right.

[Best practice] Avoid repetition.

The code would be very error-prone if you implement every function from scratch!

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

```cpp
Rational operator+(const Rational &lhs,
                   const Rational &rhs) {
  return {
    lhs.m_num * static_cast<int>(rhs.m_denom)
      + static_cast<int>(lhs.m_denom) * rhs.lhs,
    lhs.m_denom * rhs.m_denom
  };
}
Rational operator-(const Rational &lhs,
                   const Rational &rhs) {
  return {
    lhs.m_num * static_cast<int>(rhs.m_denom)
      - static_cast<int>(lhs.m_denom) * rhs.lhs,
    lhs.m_denom * rhs.m_denom
  };
}
```

`Rational`: arithmetic operators

Exercise: Define `operator*` (multiplication) and `operator/` (division) as well as `operator*=` and `operator/=` for `Rational`.

`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 * static_cast<int>(rhs.m_denom)
          + static_cast<int>(m_denom) * rhs.m_num,
      m_denom * rhs.m_denom
    };
  }
};
```

`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 {
    // ...
  }
};
```

```
Rational r = some_value();
auto s = r + 0; // OK, `r.operator+(0)`, effectively `r.operator+(Rational(0))`
auto t = 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();
auto s = r + 0; // OK, `operator+(r, 0)`, effectively `operator+(r, Rational(0))`
auto t = 0 + r; // OK, `operator+(0, r)`, effectively `operator+(Rational(0), r)`
```

[Best practice] <u>The "symmetric" operators, whose operands are often exchangeable, often should be defined as non-member functions.</u>

`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.

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

- This does not require `operator<` to be a `friend`.
- However, this is subject to floating-point errors.

`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

[Best practice] Avoid repetition.

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

```cpp
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);
}
```

Relational operators

Define relational operators in a consistent way:

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

C++20 has devoted some efforts to the design of **consistent comparison**: P0515r3.

Relational operators

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?
}
// Much better design: Use a named function.
bool less_in_x(const Point2d &lhs, const Point2d &rhs) {
  return lhs.x < rhs.x;
}
```

[Best practice] Operators should be used for operations that are likely to be unambiguous to users.

- If an operator has plausibly more than one interpretation, use named functions instead. Function names can convey more information.

`std::string` has `operator+` for concatenation. Why doesn't `std::vector` have one?

`++` 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; simplify(); return *this; }
  Rational operator++(int) { // This `int` parameter is not used.
    // The postfix version is almost always defined like this.
    auto tmp = *this;
    ++*this; // Makes use of the prefix version.
    return tmp;
  }
};
```

`++` and `--`

```cpp
class Rational {
public:
  Rational &operator++() { ++m_num; simplify(); 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:

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

Output operator:

```cpp
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

  ```cpp
  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.get_numerator() << '/' << r.get_denominator();
}
```

`Rational`: input operator

Suppose the input format is `a b` for the rational number $\dfrac ab$, where `a` and `b` are integers.

```cpp
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]` is interpreted as `a.operator[](i)`.

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

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

Example: `WindowPtr`

`WindowPtr`: indirection (dereference) operator

Recall the `WindowPtr` class we defined in the previous lecture.

```cpp
struct WindowWithCounter {
  Window theWindow;
  int refCount = 1;
};
class WindowPtr {
  WindowWithCounter *m_ptr;
public:
  Window &operator*() const { // Why should it be const?
    return m_ptr->theWindow;
  }
};
```

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

`WindowPtr`: indirection (derefernce) operator

Why should `operator*` be `const`?

```cpp
class WindowPtr {
  WindowWithCounter *m_ptr;
public:
  Window &operator*() const {
    return m_ptr->theWindow;
  }
};
```

On a `const WindowPtr` ("top-level" `const`), obtaining a non-`const` reference to the managed object may still be allowed.

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

- `this` is `const WindowPtr *`, so `m_ptr` is `WindowWithCounter *const`.

`WindowPtr`: member access through pointer

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

```cpp
class WindowPtr {
public:
  Window *operator->() const {
    return std::addressof(operator*());
  }
};
```

`std::addressof(x)` is almost always equivalent to `&x`, but the latter may not return the address of `x` if `operator&` for `x` has been overloaded!

User-defined type conversions

Type conversions

A **type conversion** is a function $f:T\mapsto U$ for two different types T and U.

Type conversions can happen either **implicitly** or **explicitly**. A conversion is **explicit** if and only if the target type `U` is written explicitly in the conversion expression.

Explicit conversions can happen in one of the following forms:

expression	explanation	example
`what_cast<U>(expr)`	through named casts	`static_cast<int>(3.14)`
`U(expr)`	looks like a constructor call	`std::string("xx")`, `int(3.14)`
`(U)expr`	old C-style conversion	Not recommended. Don't use it.

Type conversions

A **type conversion** is a function $f:T\mapsto U$ for two different types T and U.

Type conversions can happen either **implicitly** or **explicitly**. A conversion is **explicit** if and only if the target type `U` is written explicitly in the conversion expression.

- Arithmetic conversions are often allowed to happen implicitly:

  ```cpp
  int sum = /* ... */, n = /* ... */;
  auto average = 1.0 * sum / n; // `sum` and `n` are converted to `double`,
                                // so `average` has type `double`.
  ```

- The dangerous conversions for built-in types must be explicit:

  ```cpp
  const int *cip = something();
  auto ip = const_cast<int *>(cip);       // int *
  auto cp = reinterpret_cast<char *>(ip); // char *
  ```

Type conversions

A **type conversion** is a function $f:T\mapsto U$ for two different types T and U.

Type conversions can happen either **implicitly** or **explicitly**. A conversion is **explicit** if and only if the target type `U` is written explicitly in the conversion expression.

- This is also a type conversion, isn't it?

  ```cpp
  std::string s = "hello"; // from `const char [6]` to `std::string`
  ```

- This is also a type conversion, isn't it?

```cpp
std::size_t n = 1000;
std::vector<int> v(n); // from `std::size_t` to `std::vector<int>`
```

How do these type conversions happen? Are they implicit or explicit?

Type conversions

We can define a type conversion for our class `X` in one of the following ways:

1. A constructor with exactly one parameter of type `T` is a conversion from `T` to `X`.
   - Example: `std::string` has a constructor accepting a `const char *`. `std::vector` has a constructor accepting a `std::size_t`.
2. A **type conversion operator**: a conversion from `X` to some other type.

```cpp
class Rational {
public:
  // conversion from `Rational` to `double`.
  operator double() const { return 1.0 * m_num / m_denom; }
};
Rational r(3, 4);
double dval = r;  // 0.75
```

Type conversion operator

A type conversion operator is a member function of class `X`, which defines the type conversion from `X` to some other type `T`.

```cpp
class Rational {
public:
  // conversion from `Rational` to `double`.
  operator double() const { return 1.0 * m_num / m_denom; }
};
Rational r(3, 4);
double dval = r;  // 0.75
```

- The name of the function is `operator T`.
- The return type is `T`, which is not written before the name.
- A type conversion is usually a **read-only** operation, so it is usually `const`.

Explicit type conversion

Some conversions should be allowed to happen implicitly:

```cpp
void foo(const std::string &str) { /* ... */ }
foo("hello"); // implicit conversion from `const char [6]` to `const char *`,
              // and then to `std::string`.
```

Some should never happen implicitly!

```cpp
void bar(const std::vector<int> &vec) { /* ...*/ }
bar(1000);                   // ??? Too weird!
bar(std::vector<int>(1000)) // OK.
std::vector<int> v1(1000);  // OK.
std::vector<int> v2 = 1000; // No! This should never happen. Too weird!
```

Explicit type conversion

To disallow the implicit use of a constructor as a type conversion, write `explicit` before the return type:

```cpp
class string { // Suppose this is the `std::string` class.
public:
  string(const char *cstr); // Not marked `explicit`. Implicit use is allowed.
};

template <typename T> class vector { // Suppose this is the `std::vector` class.
public:
  explicit vector(std::size_t n); // Implicit use is not allowed.
};
```

```
class Dynarray {
public:
  explicit Dynarray(std::size_t n) : m_length{n}, m_storage{new int[n]{}} {}
};
```

Explicit type conversion

To disallow the implicit use of a type conversion operator, also write `explicit`:

```
class Rational {
public:
  explicit operator double() const { return 1.0 * m_num / m_denom; }
};
Rational r(3, 4);
double d = r;                     // Error.
void foo(double x) { /* ... */ }
foo(r);                           // Error.
foo(double(r));                   // OK.
foo(static_cast<double>(r));      // OK.
```

[Best practice] Avoid the abuse of type conversion operators.

Type conversion operators can lead to unexpected results!

```
class Rational {
public:
  operator double() const { return 1.0 * m_num / m_denom; }
  operator std::string() const {
    return std::to_string(m_num) + " / " + std::to_string(m_denom);
  }
};
int main() {
  Rational r(3, 4);
  std::cout << r << '\n'; // Ooops! Is it `0.75` or `3 / 4`?
}
```

In the code above, either **mark the type conversions as** `explicit`, or remove them and **define named functions** like `to_double()` and `to_string()` instead.

Contextual conversion to `bool`

A special rule for conversion to `bool`.

Suppose `expr` is an expression of a class type `X`, and suppose `X` has an `explicit` type conversion operator to `bool`. In the following contexts, that conversion is applicable even if it is not written as `bool(expr)` or `static_cast<bool>(expr)`:

- `if (expr)`, `while (expr)`, `for (...; expr; ...)`, `do ... while (expr)`
- as the operand of `!`, `&&`, `||`
- as the first operand of `?:`: `expr ? something : something_else`

Contextual conversion to `bool`

Exercise: We often test whether a pointer is non-null like this:

```
if (ptr) {
  // ...
}
auto val = ptr ? ptr->some_value : 0;
```

Define a conversion from `WindowPtr` to `bool`, so that we can test whether a `WindowPtr` is non-null in the same way.

- Should this conversion be allowed to happen implicitly? If not, mark it `explicit`.

Summary

Operator overloading

- As a non-member function: `@a` $\Leftrightarrow$ `operator@(a)`, `a @ b` $\Leftrightarrow$ `operator@(a, b)`

- As a member function: `@a` $\Leftrightarrow$ `a.operator@()`, `a @ b` $\Leftrightarrow$ `a.operator@(b)`
  - The postfix `++` and `--` are special: They have a special `int` parameter to make them different from the prefix ones.
  - The arrow operator `->` is special: Although it looks like a binary operator in `ptr->mem`, it is unary and involves special rules.
    - You don't need to understand the exact rules for `->`.
- Avoid repetition.
- Avoid abuse of operator overloading.

Summary

Type conversions

- Implicit vs explicit
- User-defined type conversions: either through a constructor or through a type conversion operator.
- To disable the implicit use of the user-defined type conversion: `explicit`
- Avoid abuse of type conversion operators.
- Conversion to `bool` has some special rules (*contextual conversion*).