Operator overloading

Programação (L.EIC009)

Eduardo R. B. Marques, DCC/FCUP

Introduction

Operator overloading let us associate programmer-defined behavior to standard operators like `+`, `-`, etc.

Operator overloading allows for more concise programs, and is used in many classes of the C++ runtime library. For instance, for `std::string` objects `a`, `b`, and `c` it is shorter to write

```
s = a;
s += b;
```

than

```
s.assign(a);
s.append(b);
```

We will now see how operator overloading works. Again we will use the `fraction` and `polynomial` classes an example, this time making use of operator overloading. The code is available at GitHub.

Example

To overload operators, special functions need to be written in correspondence to each overloaded operator and associated argument types.

```
return_type operatorOP( ... arguments ...)
```

This can be done for instance through class member functions. For example, `std::string` overloads operators `=`, `+=` e `[]` that are implemented through member functions:

```
string& operator=(const std::string& str);
string& operator=(const char* s);
...
string& operator+= (const string& str);
string& operator+= (char c);
...
const char& operator[](size_t pos) const;
char& operator[](size_t pos);
```

Example (cont.)

In `std::string` we have ...

```
string& operator=(const std::string& str);
string& operator=(const char* s);
...
string& operator+= (const string& str);
string& operator+= (char c);
...
const char& operator[](size_t pos) const;
char& operator[](size_t pos);
```

Example use:

```
std::string a("ABC"), b("DEF");
a += b;     // --> operator+= (const string& str);
a += b[0];  // --> string& operator+= (char c);
           // and const char& operator[](size_t pos) const;
b = a;      // --> string& operator= (const string& str);
a = "XYZ";  // --> string& operator=(const char* s);
```

Overloading binary operators

```
T1 a = ...;
T2 b = ...;
TR r = a OP b;
```

For a binary operator `OP` with operands of type `T1` and `T2`, and return type `R`, there are two options for implementations.

1 The first option is to implement `OP` as a member function of `T1`. The `this` self reference is implicitly the first argument:

```
class T1 {
  ...
  R operatorOP(T2 arg);
};
```

`a OP b` is equivalent to calling `a.operatorOP(b)`.

This is the best choice if we can define/change `T1` and `operatorOP` requires access to `private` declarations in `T1`.

Overloading binary operators (cont.)

1 In `fraction` we may overload `==` using a member function

```
class fraction {
  ...
  bool operator==(const fraction& f) const;
  ...
};
...
bool fraction::operator==(const fraction & f) const {
  return num == f.num && den == f.den;
}
```

Example use:

```
fraction a( ... ), b( ... );
if (a == b) // equivalent to a.operator==(b)
{ ... }
```

Overloading binary operators (cont.)

```
T1 a = ...;
T2 b = ...;
TR r = a OP b;
```

2 Alternatively, `OP` can be defined outside the definition of a class:

```
R operatorOP(T1 arg1, T2 arg2);
```

`a OP b` is equivalent to calling `operatorOP(a, b)`.

This is the only choice if we can not define/change `T1`. Moreover, `operatorOP` can only access `public` declarations in `T1`, `T2`, or `TR`, (unless `operatorOP` is declared to be a `friend`).

Overloading binary operators (cont.)

2 Alternatively, we can then overload `==` in `fraction` as:

```
class fraction { ... }; // no operator== here
// External function
bool operator==(const fraction& a, const fraction& b) const
  return a.numerator() == b.numerator()
      && a.denominator() == b.denonimator();
}
```

Example use:

```
fraction a( ... ), b( ... );
if (a == b) // equivalent to operator==(a,b)
{ ... }
```

Overloading unary operators

Similarly to binary operators, unary operators can be implemented as member functions or externally to a class. For

```
T a;
R b = OP a
```

1 `OP` can be a member function of `T`:

```
class T {
  ...
  R operatorOP();
};
```

2 or `OP` can be defined externally

```
TR operatorOP(T arg);
```

Overloading unary operators (cont.)

For instance, we can overload the - unary operator (- has two variants, unary and binary) in one of two ways:

1

```
class fraction {
   ...
   fraction operator-() const;
};
fraction fraction::operator-() const {
  return fraction(- num, den);
}
```

2

```
class fraction { ... }; // no operator-()
fraction operator-(const fraction& f) const {
   return fraction(- f.numerator(), f.denominator());
}
```

Common cases of operator overloading

```
class C {
  ...
  bool operator==(const C& other) const { ... }
  bool operator!=(const C& other) const { ... }
  bool operator<(const C& other) const { ... }
  bool operator<=(const C& other) const { ... }
  bool operator>(const C& other) const { ... }
  bool operator>=(const C& other) const { ... }
};
```

Equality operators `==` and `!=` test if two objects are equivalent. Comparison operators `<`, `<=`, `>`, `>=` are used to define element ordering. In line with this intention, usually the return type is `bool` and arguments are `const` references.

Common cases of operator overloading (cont.)

```
bool fraction::operator==(const fraction& f) const {
  return num == f.num && den == f.den;
}
bool fraction::operator!=(const fraction& f) const {
  return ! (*this == f); // calls operator==
}
bool fraction::operator<(const fraction& f) const {
  return num * f.den - f.num * den < 0;
}
bool fraction::operator<=(const fraction& f) const {
  return num * f.den - f.num * den <= 0;
}
bool fraction::operator>(const fraction& f) const {
  return ! (*this <= f); // calls operator<=
}
bool fraction::operator>=(const fraction& f) const {
   return ! (*this < f); // calls operator<
}
```

Common cases of operator overloading (cont.)

```
bool polynomial::operator==(const polynomial& p) const {
  return coeffs == p.coeffs; // use operator== in vector
}
bool polynomial::operator!=(const polynomial& p) const {
  return coeffs != p.coeffs; // use operator!= in vector
}
```

Note that the `==` and `!=` operator implementations for `vector` will use the `fraction` operators to test equality between individuals members of the two vector objects in the same position.

Common cases of operator overloading (cont.)

```
class C {
  ...
  C& operator=(const C& other) {
    ... // Copies state of other to this
    return *this; // returns reference to self
  }
  ...
};
```

The assignment operator `=` is used to assign state between objects. It is typically implemented as a member function that returns `*this` to allow for chained calls, e.g.

```
a = b = c;
```

Composed assignment operators like `+=` are implemented similarly.

Common cases of operator overloading (cont.)

```
fraction& fraction::operator=(const fraction& f) {
  num = f.num;
  den = f.den;
  return *this;
}
fraction& fraction::operator+=(const fraction& f) {
  num = num * f.den + f.num * den;
  den = den * f.den;
  reduce();
  return *this;
}
```

Common cases of operator overloading (cont.)

```
polynomial& polynomial::operator=(const polynomial& p) {
  coeffs = p.coeffs; // use operator= in vector
  return *this;
}
fraction polynomial::evaluate(const fraction& x) const {
  fraction r(0), pow(1);
  for (const fraction& c : coeffs) {
    r += c * pow; // uses fraction::operator+=
    pow *= x; // uses fraction::operator*=
  }
  return r;
}
```

Common cases of operator overloading (cont.)

```
const fraction& polynomial::operator[](size_t index) const {
  return coeffs[index];
}
fraction& polynomial::operator[](size_t index) {
  return coeffs[index];
}
```

Operator `[]` is used for indexing an object, e.g. using integer indexes as in `std::string` or `std::vector`.

Typically two variants are defined: a `const` variant that returns `const` references, and a non-`const` variant that returns mutable references.

Common cases of operator overloading (cont.)

Operators `>>` and `<<` are used for reading and writing to/from a stream.

They are implemented as external functions to classes, since we cannot change the declaration of classes in the C++ library like `std::istream` or `std::ostream`.

To allow chained calls, for instance as in `std::cout << a << b << ...`, the implementations typically return a reference to the stream object.

```cpp
std::ostream&
operator<<(std::ostream& out, const fraction& f) {
  out << f.numerator();
  if (f.denominator() != 1) out << '/' << f.denominator();
  return out; // to allow chained calls
}
```