Sobrecarga de Operadores

Gonzalo Travieso

2019

1 Uma inconveniência com a última versão de `Rational`

Na última versão para a classe `Rational`, vimos como o uso de *funções amigas* para as operações aritméticas sobre racionais permite um código mais natural:

```
Rational a(1,2), b(3,4), c;
c = add(a, b);
```

em contraposição com o uso de métodos, como na versão anterior:

```
Rational a(1,2), b(3,4), c;
c = a.add(b);
```

Entretanto, isso ainda é bem mais inconveniente do que o que podemos fazer com outros tipos de dados, como por exemplo inteiros:

```
int x{1}, y{2}, z;
z = x + y;
```

Isto fica mais claro em operações mais complexas. Contraste:

```
double x{1.2}, y;
y = 2 * x * x - 5 * x + 6;
```

com o correspondente para `Rational`:

```
Rational r{12, 10}, s;
Rational const cinco{5};
Rational const dois{2};
Rational const seis{6};
s = add(subtract(multiply(dois, multiply(r, r)),
                 multiply(cinco, r)),
        seis);
```

Nestes casos, C++ permite ainda maior conveniência do que com o uso de funções amigas: trata-se da possibilidade de definir **sobrecarga de operadores**.

2 Sobrecarregando operadores

Lembre-se que o termo *sobrecarga de nome de função* indica que funções distintas podem ter o mesmo nome, desde que operem sobre parâmetros de tipos distintos. O mesmo ocorre com os operadores em C++. Veja no exemplo:

```
int a{1}, b{2}, c;
c = b * b - a;
double x{1}, y{2}, z;
z = y * y - x;
```

Os mesmos operadores `*` e `-` indicam operações *diferentes*, de acordo com o tipo de dados dos operandos: no primeiro exemplo, operações aritméticas sobre números em complemento de 2, no segundo exemplo, operações aritméticas sobre números de ponto flutuante de precisão dupla IEEE-754.

Da mesma forma que os operadores têm sobrecarga para diversos tipos pré-definidos da linguagem, eles podem ser sobrecarregados para tipos definidos pelo usuário. A forma mais simples de demonstrar como fazer isso é através do exemplo da classe `Rational`, com as operações aritméticas implementadas com sobrecarga de operadores:

```
// VERSÃO 5!
class Rational {
  int _numerator;
  int _denominator;

  // Garante que:
  // - r.denominator > 0
  // - std::gcd(r.numerator, r.denominator) == 1
  void _normalize();

public:
  // Cria um racional com numerador e denominador especificados.
  Rational(int numerator = 0, int denominator = 1)
      : _numerator(numerator), _denominator(denominator) {
    _normalize();
  }

  // Calcula a * b.
  friend Rational operator*(Rational a, Rational b);
  // Calcula a / b.
  friend Rational operator/(Rational a, Rational b);
  // Calcula a + b.
  friend Rational operator+(Rational a, Rational b);
  // Calcula a - b.
  friend Rational operator-(Rational a, Rational b);
  // Calcula -a.
  friend Rational operator-(Rational a);

  // Verifica se a == b.
  friend bool operator==(Rational a, Rational b) {
```

```
    return a._numerator == b._numerator && a._denominator == b._denominator;
  }

  // Acesso (apenas leitura) a numerador e denominador.
  int numerator() const { return _numerator; }
  int denominator() const { return _denominator; }
};
```

A única mudança necessária foi alterar os nomes das funções aritméticas para algo do tipo `operator$`, onde o `$` é substituído pelo operador que estamos implementando. Ao implementar as funções, devemos fazer uma alteração correspondente do nome. Por exemplo, para o operador de multiplicação:

```
// Calcula a * b.
Rational operator*(Rational a, Rational b) {
  Rational result;
  result._numerator = a._numerator * b._numerator;
  result._denominator = a._denominator * b._denominator;
  result._normalize();
  return result;
}
```

Isto permite que usemos a seguinte sintaxe:

```
Rational x{1, 2}, y{3, 4}, z;
z = x * y;
```

Para operações como divisão e subtração, é importante lembrar que, quando escrevemos o código `x * y`, o operando à esquerda (`x` neste caso) será passado para o primeiro parâmetro (`a` no código acima), enquanto que o operando à direita (`y`) será passado para o segundo parâmetro (`b`).

Outro ponto a notar é que definimos dois `operator-` o que é mais um exemplo de sobrecarga: estamos sobrecarregando o operador `-` para duas coisas: subtração e troca de sinal. O compilador distingue os dois casos pelo número de operandos (número de parâmetros): Quando escrevemos `x - y` estamos usando dois operandos, e portanto ele chama a função definida com dois parâmetros, quando escrevemos `-x` estamos usando apenas um operando, e portanto o compilador chama a função com apenas um parâmetro.

Com essas definições, o nosso exemplo mais complexo anterior poderia ser simplificado para:

```
Rational r{12, 10}, s;
Rational const cinco{5};
Rational const dois{2};
Rational const seis{6};
s = dois * r * r - cinco * r + seis;
```

3 Conversão automática

Este último código ainda está mais complexo do que o necessário, visto que o compilador C++ realiza *conversão automática* quando adequado.

Lembre-se do que ocorre em códigos como o seguinte:

```
double x{3}, y;
y = 2 * x;
```

Neste caso, temos duas conversões automáticas ocorrendo:

1. Ao inicializar `x`, que é `double`, usamos o valor 3, que é `int`. O compilador automaticamente converte o `int` para `double` e usa o valor convertido na inicialização.

2. Ao realizar o produto `2 * x`, `2` é `int`, enquanto `x` é `double`; o `2` precisa então ser convertido para `double` para se poder realizar a operação de produto de ponto flutuante.

Da mesma forma que existem conversões automáticas entre tipos pré-definidos como `int` e `double`, existem conversões automáticas para tipos definidos pelo usuário. Quando misturamos um tipo definido pelo usuário, como `Rational`, com outros tipos, podem ocorrer conversões automáticas, desde que o compilador seja instruído de como fazer isso. Um tipo definido pelo usuário pode ter dois tipos de conversão:

1. Conversão de outro tipo para esse tipo.

2. Conversão desse tipo para um outro tipo.

No nosso exemplo de números racionais, podemos considerar um inteiro n como equivalente a um racional $n/1$, e portanto temos como converter de um `int` para um `Rational` (o primeiro tipo de conversão acima). Também podemos considerar que um racional é um subconjunto dos reais, e portanto podemos querer converter de um `Rational` para um `double` (apesar de que neste caso pode ocorrer perda de precisão, pois o `double` pode não ter casas significativas suficientes para representar o mesmo valor do `Rational` fornecido).

Essas conversões são especificadas da seguinte forma:

1. Uma conversão de outro tipo para o seu tipo é especificada por meio de um *construtor* do seu tipo que aceita um parâmetro do outro tipo.

2. Uma conversão do seu tipo para outro tipo é especificada por um método com um nome especial de `operator T` para converter para o tipo `T`.

No nosso exemplo:

1. Para converter de um `int` para um `Rational`, precisamos definir um construtor de `Rational` que aceite um inteiro e o use como numerador, usando 1 como denominador. *Mas o construtor previamente definido já faz isso*, pois o segundo parâmetro tem *default* de 1. Portanto, sem saber, já havíamos definido uma conversão de `int` para `Rational`.

2. Para converter de `Rational` para `double` precisamos definir um `operator double` como método da classe `Rational`. Essa função de conversão não deve especificar tipo de retorno:

   ```
   operator double() const {
     return static_cast<double>(_numerator) / _denominator;
   }
   ```

Infelizmente, as regras de C++ não permitem que definamos as duas conversões apresentadas acima, pois como o C++ tenta converter entre `double` e `int` automaticamente, se definidos as duas conversões acima, quanto fazermos a operação `2*r` onde `r` é um `Rational`, o compilador não saberia se queremos converter o 2 para `Rational` e usar o operador definido para essa classe ou se queremos usar a conversão de `r` para `double` e usar o produto de `double` (depois de converter o 2 também para `double`)! Portanto, precisamos definir apenas uma delas. A mais útil é a de `int` para `Rational` (que não envolve perda de precisão), e deixamos a conversão para `double` não-automática, através de um método especial, por exemplo `to_double`.

Com a conversão automática, nosso exemplo anterior pode agora ser feito de forma muito mais simples:

```
Rational r{12, 10}, s;
s = 2 * r * r - 5 * r + 6;
```

4 Sobrecarga dos operadores de inserção e extração

Vimos como os operadores « (inserção) e » (extração) podem ser usados para realizar operações de entrada e saída, respectivamente, sobre *streams* de arquivos ou entrada/saída padrão:

```
int a{3};
std::cout << a;
std::cin >> a;
std::ofstream sai("saida.txt");
sai << a;
std::ifstream entra("entrada.txt");
entra >> a;
```

Esses operadores são definidos como operadores cujo primeiro parâmetro é uma referência para um *stream* de dados, um `ostream` no caso do operador « e um `istream` no caso do operador ».

Isso significa que podemos definir como escrever ou ler um tipo de dados que estamos criando simplesmente sobrecarregando esses operadores para o nosso tipo. No exemplo de `Rational` podemos fazer:

```
std::ostream& operator<<(std::ostream &os, Rational r) {
    os << r.numerator() << "/" << r.denominator();
    return os;
}

std::istream &operator>>(std::istream &is, Rational &r) {
  int n, d;
  char separador;
  // Lê o numerador.
  is >> n;
  if (!is.good()) // Interrompe se erro.
    return is;
```

```
  // Lê o separador /
  is >> separador;
  if (!is.good()) // Interrompe se não conseguiu ler
    return is;
  if (separador != '/') {
    // Se não era um /, é um erro de formato! Interrompe.
    is.setstate(std::ios::failbit);
    return is;
  }
  // Lê o denominador.
  is >> d;
  if (!is.good()) // Interrompe se erro.
    return is;
  // Ajusta o valor de r para o lido.
  r = Rational{n, d};
  return is;
}
```

Vejamos os detalhes:

- O operador `«` tem à sua esquerda uma referência para um `std::ostream`, e à sua direita um objeto do tipo a ser escrito.
- O operador `»` tem à sua esquerda uma referência para um `std::istream` e à sua direita uma *referência* para um objeto do tipo a ser lido. Passar o objeto a ser lido por referência é necessário, visto que a leitura altera o seu valor.
- Ambos *retornam uma referência* para o mesmo objeto do tipo *stream* à sua esquerda (verifique o tipo de retorno e as expressões `return`). Isso é feito para permitir usos concatenados, da forma:

  ```
  std::cout << r1 << ", " << r2 << std::endl;
  ```

- Nenhum desses operadores faz acesso direto aos membros privados de `Rational`, portanto eles não precisam ser declarados `friend` da classe.
- Adicionalmente, o código do operador `»` foi escrito para apenas alterar o valor do racional a ser lido se um racional for lido com sucesso. Caso contrário, o objeto passado permanece inalterado, e o *flag* correspondente de erro fica ligado na *stream* `is`.

5 Quais operadores podem ser sobrecarregados

A grande maioria dos operadores de C++ pode ser sobrecarregada para novos tipos de dados. A exceção são os operadores `::` (operador de escopo), `.` (operador de acesso a membro), `.*` (operador de ponteiro para membro) e `?:` (operador ternário de condicional), que não podem ser sobrecarregados.

Entre os operadores que podem ser sobrecarregados, a maioria pode ser sobrecarregada tanto como membro (método) ou como função amiga. Entre tanto, os seguintes operadores só podem ser sobrecarregados como membros: `->` (operador de acesso a membro por ponteiro), `[]` (operador de indexação),

`()` (operador de chamada de função) e `=` (operador de atribuição). Recomenda-se dar preferência a sobrecarga por funções amigas, quando possível, devido à maior versatilidade em combinação com conversões automáticas; por outro lado, os operadores de atribuição `+=`, `-=`, etc e de auto-incremento e auto-decremento. são mais naturalmente expressos como operadores membro.

O C++ não permite a definição de novos operadores, inexistentes na linguagem original; ele também não permite mudar precedência ou associatividade de operadores. Portanto, tome cuidado para que a precedência e associatividade dos operadores que você define permitam um uso natural para o intuito do seu tipo. Por exemplo, não tente sobrecarregar o operador `^` como um operador de exponenciação no seu tipo, pois a interpretação natural de `a ^ 2 - b` como uma expressão aritmética seria `(a ^ 2) - b`, mas a interpretação dada pelo C++ será `a ^ (2 - b)`, pois o `^` tem precedência mais baixa que o `-`.

6 A palavra-chave `this`

Quando um método é executado, ele é executado sobre um objeto específico. No código do método, a menção de um membro da classe sem especificar qual objeto usar será um acesso ao membro do objeto sobre o qual o método foi chamado. Isso resolve a maioria dos casos onde queremos trabalhar com o objeto sobre o qual o método foi chamado. Entretanto, existem algumas situações, que veremos logo a seguir, onde queremos nos referir explicitamente a esse objeto. Isso pode ser feito com o uso da palavra-chave `this`: no código de um método, a palavra-chave `this` é um **ponteiro** para o objeto sobre o qual o método foi chamado.

Por causa disso, um código como o seguinte:

```
int numerator() const { return _numerator; }
```

é exatamente equivalente a:

```
int numerator() const { return this->_numerator; }
```

Como dito, na chamada `a.numerator()`, durante a execução do método, `this` será um ponteiro para `a`, e portanto `this->_numerator` significa: acesse o campo `_numerator` do objeto apontado por `this`, que é o `a`.

7 Alguns operadores especiais

7.1 Operador de indexação

O operador de indexação permite usar um objeto como um vetor de elementos. Esse operador deve ter exatamente um parâmetro, que será usado como índice; esse parâmetro pode em princípio ser de qualquer tipo. O valor de retorno deve ser *uma referência* para o objeto indexado. O fato de ser uma referência permite que usemos a indexação à esquerda de uma atribuição, ou passemos o resultado da indexação para um parâmetro que espera uma referência.

Um exemplo não muito útil que simula a indexação com índices negativos do Python:

```cpp
#include <vector>
#include <iostream>

class FunkyVector {
    std::vector<int> _v;
public:
    FunkyVector(size_t n) : _v(n) {}
    size_t size() const { return _v.size(); }
    int &operator[] (int i) {
        if (i >= 0) return _v[i];
        else return _v[_v.size() + i];
    }
};

int main(int, char *[])
{
    FunkyVector v(10);
    for (size_t i = 0; i < v.size(); ++i) {
        v[i] = i; // v[i] é uma referência, então isto funciona!
    }

    for (int i = -5; i < 5; ++i) {
        std::cout << "v[" << i << "] = " << v[i] << std::endl;
    }
}
```

7.2 Operador de chamada de função

É possível definir objetos que se comportam como se fossem funções. São os denominados *objetos funcionais*. Para definir um objeto funcional, devemos sobrecarregar o operador de chamada de função para a classe do objeto.

Como exemplo, abaixo está uma classe que funciona como uma função para o cálculo do i-ésimo elemento da sequência de Fibonacci, usando um método conhecido como *memoização* para evitar recalcular valores já conhecidos (aumentando a eficiência no caso de múltiplas chamadas):

```cpp
#include <iostream>
#include <vector>

class Fibonacci {
    std::vector<unsigned int> _known_values{1, 1};
public:
    unsigned int operator() (unsigned int i) {
        while (_known_values.size() <= i) {
            auto n = _known_values.size();
            auto new_value = _known_values[n-1] + _known_values[n-2];
            _known_values.push_back(new_value);
        }
        return _known_values[i];
    }
```

```
};

int main(int, char *[])
{
    Fibonacci fib;
    for (unsigned int i = 0; i < 30u; ++i) {
        std::cout << fib(i) << " ";
    }
    std::cout << std::endl;
    // Nenhum novo valor será calculado no próximo loop.
    for (unsigned int i = 0; i < 20u; ++i) {
        std::cout << fib(i) << " ";
    }
    std::cout << std::endl;
}
```

Note como o objeto `fib`, definido como um objeto da classe `Fibonacci`, é usado nos *loops* como se fosse uma função. Isso ocorre pois a classe define uma sobrecarga para o operador `()`, que é o operador de chamada de função. O método correspondente é chamado quando o objeto é usado como uma função. Neste caso, o objeto guarda um vetor com valores já calculados da sequência de Fibonacci. Quando uma chamada é realizada, se o índice desejado já foi calculado, a função apenas retorna o valor conhecido. Quando um novo índice é pedido pela primeira vez, os valores até aquele índice são calculados.

Neste caso, a função definida tem apenas um parâmetro, mas não existem restrições quanto a isso: o operador `()` pode ter quantos parâmetros se desejar, dos tipos que se desejar. Podemos inclusive usar sobrecarga de nome de função para definir chamadas de função com parâmetros diferentes.

7.3 Operadores de auto-incremento e auto-decremento

Existem quatro operadores de auto-incremento/decremento:

- pré-incremento `++a`
- pós-incremento `a++`
- pré-decremento `--a`
- pós-decremento `a--`

A diferença entre as versões pré e pós se manifesta apenas quando usamos esses operadores em uma expressão, ao invés de isoladamente. Já comentei que não recomendo o uso desses operadores em expressões, mas apenas isoladamente, e para isso recomendo o uso das versões pré.

No entanto, se estamos sobrecarregando os operadores, devemos fazê-lo de forma correta e consistente. Para isso, precisamos saber a diferença entre as versões pré e pós. Para entender isso, vejamos o código seguinte:

```
int a{10}, b{10};
int c, d;
c = ++a; // Primeiro incrementa a, depois coloca o resultado em c.
```

```
d = b++; // Primeiro usa o valor original de b (10) para colocar
         // em d, depois incrementa b.
// Agora a == 11, b == 11, c == 11, d == 10
```

De acordo com a definição de C++, os operadores pré devem retornar **uma referência** para o objeto incrementado/decrementado. Já os operadores pós devem retornar **o valor** anterior do objeto antes do incremento/decremento.

Mas existe um problema a resolver: tanto `++a` quanto `a++` são um operador com nome `++` e que tem apenas um parâmetro. Portanto, com as regras de sobrecarga de C++ *não conseguimos distinguir um do outro.* Para resolver isto, a linguagem diz que os operadores pós devem ter um parâmetro adicional do tipo `int`, que deve ser ignorado. Esse parâmetro serve apenas para o compilador conseguir distinguir a sobrecarga do operador. Não é uma solução elegante, mas funciona.

Normalmente, implementados diretamente o operador pré e então implementamos o operador pós correspondente fazendo uma chamada ao operador pré.

Vejamos o exemplo para a classe `Rational`, com os operadores implementados como membros:

```
// Implementa ++a.
Rational &Rational::operator++() {
  _numerator += _denominator;
  return *this;
}

// Implementa a++ usando ++a.
Rational Rational::operator++(int) {
  Rational tmp{*this};
  ++(*this);
  return tmp;
}
```

Note como usamos o `this` em dois casos: nos operadores pré para poder retornar uma referência para o objeto incrementado (`this` aponta para esse objeto, então o objeto é `*this`) e nos operadores pós para executar o operador pré sobre o objeto.

7.4 Operadores de atribuição

Os diversos operadores de atribuição podem ser também sobrecarregados.

7.4.1 Operador de atribuição simples =

Esse operador é tão importante que o compilador C++ fornece uma implementação automática, que consiste em copiar membro a membro de um objeto para outro. Por exemplo, na classe `Rational`, quanto temos:

```
Rational a(5, 1), b;
b = a;
```

como não definimos explicitamente uma sobrecarga do operador de atribuição para essa classe, esse código é transformado em:

```
Rational a(5, 1), b;
b._numerator = a._numerator;
b._denminator = a._denominator;
```

(como se esse código fosse executado por um método da classe, portanto acessando livremente os membros privados).

Essa é a chamada *implementação default* do operador de atribuição. Quando, como no caso de `Rational`, essa implementação é suficiente não precisamos definir sobrecarga explicitamente. Entretanto, em alguns casos precisamos realizar isso, pois a implementação *default* levaria a erros. Vamos voltar a discutir isso quando falarmos de gerenciamento de recursos.

7.4.2 Operadores de atualização (+=, -=, etc.)

Existem também as operações de atribuição que atualizam os valores através de uma operação, como soma `+=`, subtração `-=` e similares. Esses operadores podem também ser sobrecarregados.

Para realizar a sobrecarga de um desse operadores, precisamos apenas considerar a forma correta:

- A execução do operador deve atualizar o valor do objeto corretamente (não gerar um novo valor, como no caso dos operadores simples).
- O método deve retornar uma referência para o objeto que recebeu a atribuição.

Como exemplo, veja uma forma de implementar o operador `+=` para a classe `Rational` (como método, e não função amiga):

```
// Implementa a += b.
Rational &Rational::operator+=(Rational other) {
  *this = *this + other;
  return *this;
}
```

O objeto que recebe a chamada é o objeto à esquerda do operador, isto é, aquele que vai ter seu valor atualizado. O método realiza o cálculo do novo valor desse objeto, atualiza o objeto e então retorna uma referência para esse objeto.

8 Mantenha a sanidade

O uso de sobrecarga de operadores é importante em C++. Além da conveniência no uso, os operadores permitem escrever mais naturalmente diversos algoritmos que servem para diferentes tipos de dados. Por exemplo, o algoritmo `std::sort`, que ordena uma lista de valores, pode ser usado para listas contendo qualquer tipo de elementos, desde que exista um operador `<` implementado para esse tipo de dados.[1]

No entanto, para funcionar bem, algumas regras devem ser seguidas:

[1] Existe uma variante do algoritmo que pode trabalhar com tipos que não definem o operador `<`.

- Só implemente operadores que façam sentido para o seu tipo. Por exemplo, se você implementa o operador `+` para o seu tipo, isso signfica que o seu tipo possui algum conceito de "soma" de valores. Se você tem um tipo que representa um ponto no espaço bidimensional (como em um exemplo anterior), não faz sentido implementar o operador `+` para ele: o que signfica a soma de dois pontos? Por outro lado, se o seu tipo representa um vetor bidimensional, então o `+` faz sentido, mas o `/` não.

- O C++ não verifica a compatibilidade entre operadores distinto; isso fica por conta do programador, e deve ser considerada com cuidado. Por exemplo, em geral queremos que `2 * x` seja igual a `x + x`, e que `a += b` dê o mesmo resultado que `a = a + b`, e que `a != b` seja o inverso lógico de `a == b`; o programador deve realizar as implementações para garantir isso. Para ajudar, você pode usar um operador para implementar outro, quando conveniente; por exemplo, usar `==` na implementação de `!=`:

  ```
  bool operator!=(MeuTipo a, MeuTipo b) { return !(a == b); }
  ```