Sobrecarga de Operadores

Estrutura de Dados — QXD0010

Prof. Atílio Gomes Luiz

gomes.atilio@ufc.br

Universidade Federal do Ceará

2º semestre/2021

Leituras recomendadas para este tópico

- **Capítulo 11** (Sobrecarga de operadores) do livro *C++ Como Programar* dos autores Deitel, Quinta Edição.

- **Capítulo 13** (Operator overloading) do tutorial learn.cpp (https://www.learncpp.com/)

Introdução

Operadores

- Operador é um símbolo que indica uma operação.
  - Exemplos: $+$, $-$, $*$, $/$, $\%$, $\&$, $=$, $<$, $>$, $!=$, $==$, $<<$, $>>$, etc.
- Sobrecarga: atribuir diferentes significados a uma operação, dependendo do contexto.

Operadores

- Operador é um símbolo que indica uma operação.
  - Exemplos: $+$, $-$, $*$, $/$, $\%$, $\&$, $=$, $<$, $>$, $!=$, $==$, $<<$, $>>$, etc.
- Sobrecarga: atribuir diferentes significados a uma operação, dependendo do contexto.
- Por exemplo: operadores de entrada ($>>$) e saída ($<<$)
  - A definição padrão do operador $<<$ é para o deslocamento de bits.
  - Mas também é usado, juntamente com o objeto std::cout, para mostrar os valores de vários tipos de dados

Sobrecarga de operadores

- Técnica importante que tem melhorado o poder de extensibilidade do C++
- O C++ tenta fazer os tipos de dados definidos pelo usuário se comportarem de forma muito parecida como os tipos de dados nativos.
- Embora não permita que novos operadores sejam criados, o C++ permite que a maioria dos operadores existentes seja sobrecarregada de modo que, quando forem utilizados com objetos, eles tenham um significado apropriado para esses objetos. Essa é uma capacidade poderosa.
- Por exemplo, C++ nos permite adicionar dois objetos definidos pelo usuário usando a mesma sintaxe que é aplicada aos tipos nativos.

Sobrecarga de operadores

- O operador de adição (+) consegue trabalhar com operandos de tipos nativos como: char, int, float e double.

- Contudo, se `s1`, `s2`, `s3` são objetos da classe `string`, nós também conseguimos compilar a declaração:

```
s3 = s1 + s2;
```

- Isso significa que o C++ possui a habilidade de adaptar os operadores para que eles tenham um significado específico ao relacionarem dois objetos de um certo tipo de dados.

- O mecanismo de dar um significado especial a um operador, dependendo dos operandos, é conhecido como sobrecarga de operadores.

Operadores sobrecarregáveis

e operadores não-sobrecarregáveis

A maioria dos operadores em C++ pode ser sobrecarregada.

Operadores que podem ser sobrecarregados							
+	-	*	/	%	^	&	|
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&	||	++
--	->*	,	->	[]	()	new	delete
new[]	delete[]						

Operadores que não podem ser sobrecarregados			
.	.*	::	?:

A classe `std::string`

- Uma classe que sobrecarrega muitos operadores é a classe string.
- Analisar o arquivo `StringOperadores.cpp`

TAD Fraction (Fração)

TAD Fraction

- Um tipo de dados que representa uma fração pode ser implementado como uma classe chamada `Fraction`.

TAD Fraction

- Um tipo de dados que representa uma fração pode ser implementado como uma classe chamada `Fraction`.

- Uma `Fraction` deve ter como atributos um `numerator` e `denominator` inteiros, e deve ter pelo menos os seguintes métodos:
  - o construtor `Fraction(num, den)`: recebe dois inteiros `num` e `den` que são o numerador e o denominador da fração. O construtor deve simplificar a fração obtendo uma fração equivalente, mas com numerador e denominador o menor possível.
  - `double getNumerator()`: retorna o numerador.
  - `double getDenominator()`: retorna o denominador.

TAD Fraction

- Um tipo de dados que representa uma fração pode ser implementado como uma classe chamada `Fraction`.

- Uma `Fraction` deve ter como atributos um `numerator` e `denominator` inteiros, e deve ter pelo menos os seguintes métodos:
  - o construtor `Fraction(num, den)`: recebe dois inteiros `num` e `den` que são o numerador e o denominador da fração. O construtor deve simplificar a fração obtendo uma fração equivalente, mas com numerador e denominador o menor possível.
  - `double getNumerator()`: retorna o numerador.
  - `double getDenominator()`: retorna o denominador.

- Além disso, gostaríamos de realizar operações aritméticas entre dois objetos da classe `Fraction`. Também gostaríamos de imprimir no terminal um objeto da classe `Fraction` apenas passando-o para o objeto `cout` usando o operador de inserção (`<<`)

Operações aritméticas em frações

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- O tipo de dados `Fraction` representa o conjunto dos números racionais com a seguinte definição matemática:

$$\mathbb{Q} = \left\{ \frac{a}{b} \mid a, b \in \mathbb{Z}, b \neq 0 \right\}$$

Implementaremos as seguintes operações no nosso tipo TAD `Fraction`:

- $\frac{a}{b} + \frac{c}{d} = \frac{ad+cb}{bd}$
- $\frac{a}{b} - \frac{c}{d} = \frac{ad-cb}{bd}$
- $\frac{a}{b} \times \frac{c}{d} = \frac{ac}{bd}$
- $\frac{a}{b} \div \frac{c}{d} = \frac{ad}{bc}$

Simplificação de fração

- A fração $\frac{2}{4}$ é igual a $\frac{1}{2}$, mas $\frac{2}{4}$ não está reduzida aos termos mais baixos.

Simplificação de fração

- A fração $\frac{2}{4}$ é igual a $\frac{1}{2}$, mas $\frac{2}{4}$ não está reduzida aos termos mais baixos.

- Podemos reduzir qualquer fração dada aos termos mais baixos encontrando o máximo divisor comum (MDC) entre o numerador e o denominador e, em seguida, dividindo o numerador e o denominador pelo MDC.

Simplificação de fração

- A fração $\frac{2}{4}$ é igual a $\frac{1}{2}$, mas $\frac{2}{4}$ não está reduzida aos termos mais baixos.

- Podemos reduzir qualquer fração dada aos termos mais baixos encontrando o máximo divisor comum (MDC) entre o numerador e o denominador e, em seguida, dividindo o numerador e o denominador pelo MDC.

- A partir do C++17, existe a função `std::gcd` na biblioteca `<numeric>`, que calcula o MDC de dois inteiros.

Simplificação de fração

- A fração $\frac{2}{4}$ é igual a $\frac{1}{2}$, mas $\frac{2}{4}$ não está reduzida aos termos mais baixos.

- Podemos reduzir qualquer fração dada aos termos mais baixos encontrando o máximo divisor comum (MDC) entre o numerador e o denominador e, em seguida, dividindo o numerador e o denominador pelo MDC.

- A partir do C++17, existe a função `std::gcd` na biblioteca `<numeric>`, que calcula o MDC de dois inteiros.

- A mesma função pode ser implementada como a seguir:

```
1 #include <cmath>
2
3 int gcd(int a, int b) {
4     return (b == 0) ? std::abs(a) : gcd(b, a % b);
5 }
```

Programa principal

Primeiramente, vamos definir o código de utilização do nosso TAD:

```
1  #include <iostream> // mainFraction.cpp
2  #include "Fraction.h"
3  using namespace std;
4
5  int main() {
6      Fraction f1(2, 3);
7      Fraction f2(4, 5);
8
9      cout << f1 << endl;       // imprime 2/3
10     cout << f1 + f2 << endl; // imprime 22/15
11     cout << f1 - f2 << endl; // imprime -2/15
12     cout << f1 * f2 << endl; // imprime 8/15
13     cout << f1 / f2 << endl; // imprime 5/6
14 }
```

```
1  #ifndef FRACTION_H // Arquivo de cabeçalho Fraction.h
2  #define FRACTION_H
3  #include <iostream>
4  #include <numeric>
5
6  class Fraction {
7  private:
8      int numerator;
9      int denominator;
10
11 public:
12     // Construtor default
13     // Ele chama o construtor com dois argumentos
14     Fraction() : Fraction(1,1) {}
15
16     // Construtor com dois argumentos
17     Fraction(int num, int den)
18         : numerator(num), denominator(den)
19     {
20         reduce();
21     }
```

Continuação da declaração da classe `Fraction`

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
22      // função que simplifica a fração
23      // reduz numerador e denominador aos menores termos
24      void reduce() {
25          int gcd = std::gcd(numerator, denominator); // C++17
26          numerator = numerator/gcd;
27          denominator = denominator/gcd;
28      }
29
30      // getters e setters
31      int getNumerator() { return numerator; }
32      void setNumerator(int num) { numerator = num; }
33      int getDenominator() { return denominator; }
34      void setDenominator(int den) { denominator = den; }
```

Somando duas frações

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
Fraction f1 {2, 3};
Fraction f2 {1, 3};
```

- Gostaríamos de somar as frações `f1` e `f2`. Como fazer isso usando o operador aritmético de adição?
- Será que apenas fazendo `f1 + f2` obteremos o resultado desejado?

Somando duas frações

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
Fraction f1 {2, 3};
Fraction f2 {1, 3};
```

- Gostaríamos de somar as frações `f1` e `f2`. Como fazer isso usando o operador aritmético de adição?
- Será que apenas fazendo `f1 + f2` obteremos o resultado desejado?
  - Resposta: NÃO. O compilador não entende o que fazer neste caso.
- Precisamos sobrecarregar o operador binário de adição (+)

Sobrecarregando um operador

- Um operador é sobrecarregado escrevendo uma definição de função-membro (da classe ou do struct) ou uma definição de função global, exceto pelo fato de que o nome da função agora se torna a palavra-chave `operator` seguida pelo símbolo do operador sendo sobrecarregado.

  Por exemplo, o nome de função `operator+` seria utilizado para sobrecarregar o operador de adição (+)

Sobrecarregando um operador

- Um operador é sobrecarregado escrevendo uma definição de função-membro (da classe ou do struct) ou uma definição de função global, exceto pelo fato de que o nome da função agora se torna a palavra-chave `operator` seguida pelo símbolo do operador sendo sobrecarregado.

  Por exemplo, o nome de função `operator+` seria utilizado para sobrecarregar o operador de adição (+)

A forma geral de uma função operadora é:

```
tipo-de-retorno operator op (lista-de-argumentos)
{
    // corpo da função
}
```

Sobrecarga dos operadores aritméticos binários

- Vamos mostrar como sobrecarregar os operadores aritméticos binários usando definições de função-membro.

- Além disso, também será preciso sobrecarregar o operador de inserção (>>) e o operador de extração de conteúdo (<<)
  - Esses operadores serão sobrecarregados como funções globais, usando 'funções amigas' (`friend functions`)

Continuação da definição da classe `Fraction`

Declaração das funções operadoras

```
35      // operadores aritméticos sobrecarregados
36      // como funções-membro
37      Fraction operator+(const Fraction& f);
38      Fraction operator-(const Fraction& f);
39      Fraction operator*(const Fraction& f);
40      Fraction operator/(const Fraction& f);
41
42      // operador de extração de fluxo (<<) e operador
43      // de inserção (>>) sobrecarregados de forma global
44      // como funções 'friend'
45      friend std::ostream& operator<<(std::ostream& out,
46          const Fraction& f);
47      friend std::istream& operator>>(std::istream& in,
48          Fraction& f);
49  };
```

Continuação da definição da classe `Fraction`

Implementação das funções operadoras

```
50 Fraction Fraction::operator+(const Fraction& f) {
51     int num = this->numerator * f.denominator +
52               f.numerator * this->denominator;
53     int den = this->denominator * f.denominator;
54     return Fraction(num, den);
55 }
56
57 Fraction Fraction::operator-(const Fraction& f) {
58     int num = this->numerator * f.denominator -
59               f.numerator * this->denominator;
60     int den = this->denominator * f.denominator;
61     return Fraction(num, den);
62 }
```

Continuação da definição da classe `Fraction`

Implementação das funções operadoras

```
63 Fraction Fraction::operator*(const Fraction& f) {
64     int num = this->numerator * f.numerator;
65     int den = this->denominator * f.denominator;
66     return Fraction(num, den);
67 }
68
69 Fraction Fraction::operator/(const Fraction& f) {
70     int num = this->numerator * f.denominator;
71     int den = this->denominator * f.numerator;
72     return Fraction(num, den);
73 }
```

Continuação da definição da classe `Fraction`

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Implementação das funções operadoras

```
74 // Implementação do operador << como uma função global
75 std::ostream& operator<<(std::ostream& out, const Fraction& f)
76 {
77     out << f.numerator << "/" << f.denominator;
78     return out;
79 }
80
81 // Implementação do operador >> como uma função global
82 std::istream& operator>>(std::istream& in, Fraction& f) {
83     in >> f.numerator;
84     in.ignore(); // ignora o separador '/'
85     in >> f.denominator;
86     f.reduce();
87     return in;
88 }
89
90 #endif
```

Sobrecarregando operadores de Incremento e Decremento

Operador de incremento (++)

- Diferentemente dos operadores binários vistos até agora, os operadores $++$ e $--$ são operadores unários.

- Para sobrecarregar o operador de incremento para permitir tanto o uso de incremento prefixado como de pós-fixado, toda função operadora sobrecarregada deve ter uma assinatura distinta para que o compilador consiga determinar que versão de $++$ é pretendida.

- **Atenção:** toda a apresentação feita aqui para o operador de incremento se aplica ao operador de decremento. Por isso, vou omitir a explicação para este último.

Operador de incremento prefixado

- Suponha que quiséssemos adicionar 1 ao objeto `Fraction f`.
  - Ao ver a expressão `++f`, o compilador gera a chamada de função-membro `f.operator++()`

- O protótipo dessa função operadora seria:

```
Fraction& operator++();
```

Operador de incremento prefixado

- Suponha que quiséssemos adicionar 1 ao objeto `Fraction f`.
  - Ao ver a expressão `++f`, o compilador gera a chamada de função-membro `f.operator++()`

- O protótipo dessa função operadora seria:

```
Fraction& operator++();
```

- Se o operador de incremento prefixado é implementado como uma função global, então, ao ver a expressão `++d1`, o compilador gera a chamada de função `operator++( d1 )`

- O protótipo dessa função operadora seria declarado na classe `Fraction` como:

```
Fraction& operator++(Fraction &);
```

Implementação

Declaração dos protótipos das funções-membro na classe `Fraction`:

```
91      // Operadores de incremento e decremento prefixados
92      Fraction& operator++();
93      Fraction& operator--();
```

Implementação

Declaração dos protótipos das funções-membro na classe `Fraction`:

```
103     // Operadores de incremento e decremento prefixados
104     Fraction& operator++();
105     Fraction& operator--();
```

Implementação das funções fora da classe:

```
106 Fraction& Fraction::operator++() {
107     numerator = numerator + denominator;
108     return *this;
109 }
110
111 Fraction& Fraction::operator--() {
112     numerator = numerator - denominator;
113     return *this;
114 }
```

Operador de incremento pós-fixado

- Sobrecarregar o operador de incremento pós-fixado apresenta um desafio, porque o compilador deve ser capaz de distinguir entre as assinaturas das funções operadoras sobrecarregadas de incremento prefixado e pós-fixado.

- A convenção que foi adotada em C++ é que, ao ver a expressão de pós-incremento `f++`, o compilador gera a chamada de função-membro `d1.operator++( 0 )`

- O protótipo desta função é: `Fraction operator++( int );`

- O argumento 0 é um 'valor fictício' que permite ao compilador distinguir entre as funções operadoras de incremento pré-fixado e pós-fixado.

Operador de incremento pós-fixado

- Se o incremento pós-fixado é implementado como uma função global, então, quando o compilador vê a expressão `f++`, ele gera a chamada de função `operator++( f, 0 )`

- O protótipo dessa função seria:

```
Fraction operator++( Date &, int );
```

Atenção: o operador de incremento pós-fixado retorna objetos `Fraction` por valor, enquanto o operador de incremento prefixado retorna objetos `Fraction` por referência.

O operador de incremento pós-fixado, em geral, retorna um objeto temporário que contém o valor original do objeto antes de o incremento ocorrer. Já o operador de incremento prefixado retorna o objeto real incrementado com seu novo valor.

Implementação

Declaração dos protótipos das funções-membro na classe `Fraction`:

```
115     // Operadores de incremento e decremento pós-fixados
116     Fraction operator++(int);
117     Fraction operator--(int);
```

Implementação

Declaração dos protótipos das funções-membro na classe `Fraction`:

```
129     // Operadores de incremento e decremento pós-fixados
130     Fraction operator++(int);
131     Fraction operator--(int);
```

Implementação das funções fora da classe:

```
132 Fraction Fraction::operator++(int) {
133     Fraction temp(*this);
134     ++(*this); // usa operador prefixado para incrementar
135     return temp;
136 }
137
138 Fraction Fraction::operator--(int) {
139     Fraction temp(*this);
140     --(*this); // usa operador prefixado para decrementar
141     return temp;
142 }
```

Sobrecarregando operadores += e -=

Operador +=

- Suponha que tenhamos duas frações f1 e f2 e queiramos fazer a operação `f1 = f1 + f2`.
  - Ao ver esta expressão, o compilador gera a chamada de função-membro `f1.operator+=(f2)`

- O protótipo dessa função-membro operadora seria:

```
Fraction& operator+=(Fraction &);
```

Operador +=

- Suponha que tenhamos duas frações f1 e f2 e queiramos fazer a operação `f1 = f1 + f2`.
  - Ao ver esta expressão, o compilador gera a chamada de função-membro `f1.operator+=(f2)`

- O protótipo dessa função-membro operadora seria:

```
Fraction& operator+=(Fraction &);
```

- Se o operador += é implementado como uma função global, então, ao ver a expressão `f1 = f1 + f2`, o compilador gera a chamada de função `operator+=( f1, f2 )`

- O protótipo dessa função operadora seria declarado na classe `Fraction` como:

```
Fraction& operator+=(Fraction &, Fraction &);
```

Operadores += e -=

Implementação como função global

Fora da classe `Fraction`, digitamos o seguinte código:

```
143 Fraction& operator+=(Fraction& a, Fraction& b) {
144     a = a + b;
145     return a;
146 }
147
148 Fraction& operator-=(Fraction& a, Fraction& b) {
149     a = a - b;
150     return a;
151 }
```

Operadores += e -=

Implementação como função-membro da classe

Dentro da classe `Fraction`, digitamos o protótipo das funções:

```
152     // Operadores += e -=
153     Fraction& operator+=(Fraction& f);
154     Fraction& operator-=(Fraction& f);
```

Operadores += e -=

Implementação como função-membro da classe

Dentro da classe `Fraction`, digitamos o protótipo das funções:

```
164     // Operadores += e -=
165     Fraction& operator+=(Fraction& f);
166     Fraction& operator-=(Fraction& f);
```

Fora da classe, implementamos as funções

```
167 Fraction& Fraction::operator+=(Fraction& f) {
168     *this = *this + f;
169     return *this;
170 }
171
172 Fraction& Fraction::operator-=(Fraction& f) {
173     *this = *this - f;
174     return *this;
175 }
```

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

FIM