INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
PARAÍBA
Campus Cajazeiras

Engenharia de Controle e Automação
Programação Orientada a Objetos – 2021.2

Professor: Leandro Luttiane da Silva Linhares
E-mail: leandro.luttiane@ifpb.edu.br

Documento Texto

Sobrecarga de Operadores – Parte I

Neste momento da disciplina de **Programação Orientada a Objetos (POO)** do **Curso de Engenharia de Controle e Automação**, iremos estudar um conteúdo bastante interessante. Chegou a hora de falarmos um pouco sobre a *sobrecarga de operadores*!! Antes de iniciarmos, é importante comentar que o conteúdo apresentado neste documento tem como base o ***Capítulo 11 do livro C++: Como Programar*, de Deitel e Deitel (2006)**. Juntamente a este texto, estão sendo disponibilizadas as classes `CellNumber` e `Array`, assim como arquivos de exemplos de sua utilização. Sugere-se que estes arquivos sejam abertos quando seus códigos forem ser analisados.

Até então, para acessarmos os serviços oferecidos por um objeto de uma determinada classe, realizávamos a chamadas aos seus métodos públicos. Em alguns tipos de classes, como as matemáticas, esta notação pode ser incômoda. Seria muito mais prático e intuitivo, por exemplo, fazer com que o operador aritmético de adição "`+`" fosse utilizado para realizar a adição de dois objetos que representam números complexos (`complexo1 + complexo2`) do que efetuar a chamada de uma função-membro (método) de uma classe chamada `Complex` para realizar a mesma operação (`complexo1.soma(complexo2)`). Felizmente, a linguagem C++ permite que a maioria de seus operadores possam ser customizados, fazendo com que eles possam ser utilizados com objetos, em um procedimento denominado **sobrecarga de operadores**.

A sobrecarga de operadores é algo bastante comum. Nós a utilizamos regularmente de maneira implícita. Um exemplo de operador constantemente sobrecarregado no C++ é o "`<<`", que é utilizado tanto como operador de inserção de fluxo quanto como operador de bits de deslocamento para a esquerda. Outro exemplo é o próprio operador de adição já mencionado. Este operador é executado de forma diferente, dependendo se o seu contexto está relacionado com a aritmética de números inteiros, de ponto flutuante ou na aritmética de ponteiros.

Pelo exemplo dado de adição de dois objetos de uma suposta classe `Complex`, podemos induzir que as ações realizadas pelos operadores sobrecarregados também podem ser efetuadas por chamadas explícitas a métodos/funções (`complexo1.soma(complexo2)`). Porém, em muitos casos, o uso de sobrecarga torna a notação mais clara e intuitiva para os programadores. Vamos então aprender como utilizar esta incrível funcionalidade da linguagem C++? Ela nos será bastante útil!!

1. Fundamentos de Sobrecarga de Operadores

A rica variedade de operadores da linguagem C++ pode ser utilizada com os tipos de dados fundamentais da linguagem. Entretanto, os programadores também podem definir novos tipos, como as classes, que irão dar origem às suas instâncias, os objetos. Apesar do C++ não permitir que novos operadores sejam criados, ele permite que a maioria dos operadores existentes seja sobrecarregada, de tal forma que, quando aplicados a objetos, eles tenham uma ação/significado específico para esses objetos.

Para sobrecarregar um operador, deve-se escrever a definição de um método não estático ou a de uma função global, utilizando como nome a palavra reservada `operator` seguida pelo símbolo do operador que se deseja sobrecarregar. Por exemplo, o nome de função `operator+` seria adotado para sobrecarregar o operador de adição "`+`".

Para que um operador da linguagem C++ possa ser utilizado em objetos de uma determinadada classe, esse operador precisa ser sobrecarregado. As exceções são os operadores de atribuição "`=`", de endereço "`&`" e vírgula "`,`". Como exemplo, o operador de atribuição pode ser utilizado em toda classe para realizar atribuição membro a membro de dados da classe — o valor de cada atributo de um objeto é atribuído ao atributo correspondente de um segundo objeto. Deve-se ter cuidado com este tipo de atribuição quando a classe possui atributos que são ponteiros. Nesses casos, o mais prudente é realizar a sobrecarga explícita do operador de atribuição "`=`".

Para que a sobrecarga seja realizada, é necessário que sejam implementadas funções/métodos de sobrecarga de operadores com o objetivo de realizar as operações desejadas. Estas funções podem ser métodos de classe, funções `friend` e, em alguns casos, funções `não-friend` globais.

2. Observações em Relação à Sobrecarga de Operadores

Anteriormente, foi mencionado que a maioria dos operadores da linguagem C++ podem ser sobrecarregados. Quais seriam então as exceções? Quais os operadores que se inadvertidamente tentarmos sobrecarregá-los isso irá ocasionar um erro de sintaxe? São realmente poucos, apenas os seguintes operadores não podem ser sobrecarregados: "`.`", "`.*`', "`::`" e "`?:`".

A precedência de um operador não pode ser modificada por uma operação de sobrecarga. Em alguns casos, isso pode gerar algumas situações indigestas. Entretanto, o uso dos parênteses pode forçar a ordem de avaliação de operadores sobrecarregados em uma expressão, conforme a precedência desejada.

Não é possível alterar a associatividade de um operador por meio de sobrecarga. Se um operador é aplicado da direita para a esquerda, ele não poderá ter esta característica modificada a partir de sua sobrecarga.

A aridade de um operador, número de operandos que um operador manipula, também não pode ser modificada. Operadores unários sobrecarregados permanecem operadores unários, assim como os binários continuam sendo binários. Os operadores `&`, `*`, `+` e `-` possuem tanto a

versão unária como a binária, podendo ser sobrecarregados. O único operador ternário da linguagem C++ "`?:`" não pode ser sobrecarregado, como já dito anteriormente.

A criação de novos operadores por meio de sobrecarga não é permitida. Não é possível, por exemplo, criar o operador "`**`" para representar a operação de exponenciação. Entretanto, o operador "`^`" pode ser sobrecarregado para a exponenciação, da mesma forma que em outras linguagens.

Não é possível alterar a maneira com que os operadores atuam sobre os tipos fundamentais da linguagem. O programador não pode, por exemplo, modificar o significado de como o operador "`*`" multiplica dois números inteiros. A sobrecarga de operadores funciona apenas com objetos de tipos definidos pelo usuário ou em uma mistura de um objeto definido pelo usuário e um item de tipo fundamental da linguagem.

3. Funções-Membro (métodos) ou Funções Globais

As funções operadoras (as que efetuam a sobrecarga dos operadores) podem ser funções-membro (métodos) de uma classe ou funções globais. Normalmente, as funções globais são do tipo `friends`, por questões de desempenho. No caso dos métodos que efetuam sobrecarga de operadores, o ponteiro `this` é implicitamente utilizado para obter um dos argumentos de objeto de classe (operando esquerdo para operadores binários). Em uma chamada de função global, ambos os operandos de um operador binário devem ser explicitamente listados pelos seus parâmetros.

Ao sobrecarregar "`()`", "`[]`", "`->`" ou qualquer um dos operadores de atribuição, a função operadora deve ser declarada como um método de classe. Para os demais operadores, não há esta restrição. Eles podem ser sobrecarregados por métodos ou por meio de funções globais. Qual será então a melhor implementação?

Quando se opta por realizar a sobrecarga de um operador por meio de um método de classe, o operando à esquerda (ou único) deve ser um objeto ou referência a um objeto da classe do operador. Se o operando do lado esquerdo precisar ser um objeto de uma classe diferente ou um tipo fundamental, então a função operadora deve ser implementada como uma função global. A função operadora global deve ser declarada como `friend` de uma classe se essa função precisar acessar membros `private` ou `protected` dessa classe de forma direta.

As funções operadoras (métodos operadores) de uma classe são chamadas implicitamente pelo compilador quando o operando esquerdo de um operador binário for um objeto dessa classe ou quando o único operando de um operador unário for um objeto dessa classe.

4. Sobrecarga dos Operadores de Inserção e Extração de Fluxo

Em C++, a entrada e saída de dados de tipos fundamentais da linguagem é efetuada, respectivamente, por meio do operador de *extração de fluxo* "`>>`" e o operador de *inserção de fluxo* "`<<`". As classes fornecidas em bibliotecas juntamente com os compiladores C++ sobrecarregam estes operadores e iremos fazer o mesmo no código exemplo da classe `CellNumber`. Desta

maneira, faremos com que os operadores de inserção e extração de fluxo sejam sobrecarregados para efetuar a entrada e saída de um objeto desta classe (um tipo definido pelo programador).

Neste momento, sugere-se que os arquivos `CellNumber.hpp`, `CellNumber.cpp` e `mainCellNumber.cpp` sejam abertos para que vocês possam acompanhar as explicações que serão feitas sobre suas implementações e funcionamento. É importante comentar que o programa desenvolvido considera que os números de celular são informados corretamente.

Vamos iniciar a análise deste exemplo com uma pergunta: como é possível sobrecarregar os operadores de inserção e extração de fluxo? A sobrecarga do operador de inserção de fluxo "<<" deve ocorrer em uma expressão cujo operando esquerdo é do tipo `ostream &`, como em `cout << objetoDeClasse`. Observem que neste caso o objeto definido pelo usuário é o *operando direito do operador* que se deseja sobrecarregar. Nestes casos, o operador "<<" deve ser sobrecarregado por meio de uma *função operadora global*.

É de se imaginar que o mesmo ocorra com o operador de extração de fluxo. Este operador deve ser sobrecarregado em uma expressão cujo operando esquerdo é do tipo `istream &`, como em `cin >> objetoDeClasse`. Mais uma vez, o objeto definido pelo usuário é o *operando direito do operador* que se deseja sobrecarregar. Assim, o operador ">>" deve ser sobrecarregado por meio de uma *função operadora global*.

Para que essas funções operadoras tenham acesso aos membros `private` do objeto, essas funções podem ser transformadas em funções `friend` da classe (funções amigas), o que também irá contribuir para um melhor desempenho. Ao declarar uma função externa à classe (global) como `friend` no interior de uma classe, isso faz com que essa função amiga tenha acesso direto aos atributos e métodos da classe, incluindo os que foram definidos como `private` e `protected`. Deve-se comentar que o uso de funções amigas deve ser evitado sempre que possível, pois o seu uso diminui a identidade da orientação a objetos. O uso de funções amigas de classes representa uma quebra do conceito de encapsulamento.

Vamos agora analisar a implementação da classe `CellNumber`. No arquivo de implementação, `CellNumber.cpp`, a função operadora `operator>>` (linhas 25-35), que sobrecarrega o operador ">>", aceita a referência `istream input` e a referência `CellNumber num` como argumentos e retorna uma referência `istream`. A função `operator>>` efetua a entrada de número telefônicos de celular na forma `(99) 99123-4567` em objetos da classe `CellNumber`. Quando o compilador encontra a expressão `cin >> phone` (linha 22 do arquivo `mainCellNumber.cpp`), ele gera a chamada de função global `operator>>(cin,phone)`.

```
25 istream &operator>>( istream &input, CellNumber &number )
26 {
27    input.ignore(); // pula (
28    input >> setw( 2 ) >> number.areaCode; // entrada do código de área
29    input.ignore( 2 ); // pula ) e espaço
30    input >> setw( 5 ) >> number.exchange; // entrada do prefixo (exchange)
31    input.ignore(); // pula traço (-)
32    input >> setw( 4 ) >> number.line; // entrada de linha
33
34    return input; // permite cin >> a >> b >> c;
35 } // fim da função operator>>
```

```
16      CellNumber phone; // cria objeto phone
17
18      cout << "Digite numero do celular na forma (99) 99999-9999:" << endl;
19
20      // cin >> phone invoca operator>> emitindo implicitamente
21      // a chamada da função global operator>>( cin, phone )
22      cin >> phone;
```

Quando essa chamada é realizada, o parâmetro de referência `input` (linha 25, arquivo `CellNumber.cpp`) se torna um alias (apelido) para `cin` e o parâmetro de referência `number` se torna um alias para `phone`. No bloco de comandos da função `operator>>`, as três partes do número de telefone celular são lidas como `strings` nos atributos `areaCode`, `exchange` e `line` do objeto referenciado (linhas de código 28, 30 e 32, respectivamente).

O manipulador de fluxo `setw`, utilizado no bloco de comandos da função `operator>>`, limita o número de caracteres lidos em cada array de caractere. Assim, `setw(2)`, em conjunto com `cin` e `strings`, permite que apenas dois caracteres sejam lidos do fluxo de entrada. Os caracteres de parênteses, espaço e traço digitados pelo usuário são descartados pela chamada do método `ignore` da classe `istream`, que ignora o número especificado de caracteres no fluxo de entrada (um caractere por padrão).

A função `operator>>` retorna `istream` à referência `input` (isto é, `cin`). Isso tona possível que as operações de entrada em objetos `CellNumber` sejam realizadas em cascata com operações de entrada sobre outros objetos `CellNumber`, ou sobre objetos de outros tipos de dados. Por exemplo, um determinado programa pode utilizar dois objetos `CellNumber` em uma instrução como esta: `cin >> phone1 >> phone2`.

No último exemplo de instrução, inicialmente, a expressão `cin >> phone1` é executada por meio da chamada da função global `operator>>(cin, phone1)`. Essa chamada irá retornar uma referência a `cin` que irá substituir `cin >> phone1` na expressão, de modo que a parte restante da expressão é interpretada somente como `cin >> phone2`. Isso será executado por meio da chamada da função global `operator>>(cin, phone2)`.

A função `operator<<`, implementada nas linhas 15-20 do arquivo `CellNumber.cpp`, aceita uma referência `ostream` (`output`) e uma referência `const CellNumber` (`number`) como argumentos e retorna uma referência `ostream`. A função global `operator<<` sobrecarrega o operador "<<", exibindo objetos da classe `CellNumber`.

```
15  ostream &operator<<( ostream &output, const CellNumber &number )
16  {
17      output << "(" << number.areaCode << ") "
18             << number.exchange << "-" << number.line;
19      return output; // permite cout << a << b << c;
20  } // fim da função operator<<
```

Quando o compilador encontra a expressão `cout << phone` na linha 28 do arquivo `mainCellNumber.cpp`, a chamada de função global `operator<<(cout,phone)` é realizada.

Esta função operadora exibe as partes do número do celular como `strings` (linhas 17 e 18), pois eles são armazenados em atributos `string`.

```
26      // cout << phone invoca operator<< emitindo implicitamente
27      // chamada da função global operator<<( cout, telefone )
28      cout << phone << endl;
```

No arquivo de definição `CellNumber.hpp`, pode-se observar que as funções `operator>>` e `operator<<` foram declaradas na classe como funções `friend` globais (linhas 18 e 19). Já discutimos que elas precisam ser globais porque os objetos da classe `CellNumber` aparecem, nestes casos, como o operando direito dos operadores “>>” e “<<”. Estas funções operadoras foram declaradas como `friends`, pois precisavam acessar membros da classe não-`public` de forma direta. Isto normalmente é realizado por questões de desempenho ou porque a classe não oferece funções `get` e `set` apropriadas.

```
16  class CellNumber
17  {
18          friend ostream &operator<<( ostream &, const CellNumber & );
19          friend istream &operator>>( istream &, CellNumber & );
20      private:
21          string areaCode;  // código de área (de cidade) de 2 algarismos
22          string exchange;  // prefixo
23          string line;      // sufixo
24  }; // fim da classe PhoneNumber
```

Outra observação interessante é que a referência `CellNumber` na lista de parâmetros da função `operator<<` (linha 18) é `const`, porque um objeto `CellNumber` será simplesmente enviado para a saída, o que não ocasiona modificação no estado de um determinado objeto da classe. Em contrapartida, a referência a um objeto `CellNumber` na lista de parâmetros de operator>> (linha 19) é não-`const`, pois o objeto passado como argumento será modificado para armazenar o número telefônico informado. É bom lembrarmos que o termo `const` impede que modificações sejam realizadas em uma variável, assim como em objetos.

REFERÊNCIAS

DEITEL, H. M.; DEITEL, P. J. **C++: como programar**. 5 ed. São Paulo: Pearson Prentice Hall, 2006.