

Programação Orientada a Objectos 2022/2023

Exercícios

Ficha Nº 4

Operadores

Introdução teórica aos conceitos usados nesta ficha de exercícios

Esta ficha lida com a redefinição de operadores. Redefinir um operador significa fazer com que o compilador aceite a utilização de operadores em situações onde esse operador normalmente não seria aceite (por exemplo, por se estarem a usar novos tipos de dados definidos pelo programador), ou então levar um operador a fazer algo diferente do comportamento *default*. Em c++ os operadores podem ser redefinidos segundo certas regras:

- Apenas podem ser redefinidos operadores que já existam. No entanto, existem alguns operadores não podem ser redefinidos.
- Só se podem redefinir operadores que envolvam objetos de classes. Não se pode redefinir um operador que trabalhe apenas com inteiros, por exemplo.
- A precedência, associatividade, e número de operandos não pode ser modificada.
- Pode-se modificar o que um operador faz, e pode-se modificar o tipo de resultado que é produzido, modificando bastante o comportamento de um operador. No entanto, normalmente, mantém-se a lógica habitual dos operadores, pois o objetivo é tornar o código mais claro, e não o oposto.

Esta lista não é exaustiva quanto a pormenores. É essencial ler o livro e assistir às aulas.

O objetivo principal da redefinição dos operadores é o de aplicar a tipos de dados novos (classes) os operadores a que o programador já está habituado e que já existem para os tipos de dados integrais (*int*, *char*, *etc.*). Pretende-se maioritariamente tornar o código mais claro e de entendimento universal.

Por exemplo, pretende-se substituir expressões como `a.multiplicaPor(b)` por isto: `a * b`, que é bem mais simples de ler e independente da língua do programador.

A redefinição de operadores deve ser entendida como algo que é útil no contexto de simplificação de código e não como um instrumento obscuro com o propósito de complicar. Existem algumas situações onde a redefinição de operadores assume um carácter mais importante e quase obrigatório, como por

exemplo, o caso do operador de atribuição em situações de composição envolvendo memória dinâmica, mas na maioria dos casos a motivação principal será o de aumentar a clareza de código.

Linhas gerais na sintaxe de redefinição de operadores:

Os operadores são entendidos pelo compilador como sendo funções. A forma geral é **operatorXX** em que **XX** é o operador em questão (**+** **-** **+=** **[]** etc.). Serão funções com nomes específicos e invulgares, mas ainda assim, funções. A redefinição de um operador passará então pela redefinição da função que lhe corresponde:

Exemplo:

a = b é entendido pelo compilador como sendo **a.operator=(b)**

Assim, para redefinir o operador de atribuição, bastará colocar a função com o nome **operator=** e com o parâmetro do tipo de **b** na classe a que pertence o objecto **a**.

Nota: **a = b** já é possível mesmo sem redefinir o operador = (“operador de atribuição”). Tem o comportamento *default* de atribuir os objetos envolvidos membro a membro: **a = b** ⇔ **a.abc = b.abc**, **a.xpto = b.xpto**, **a.etc = a.etc....**). Por vezes é necessário redefinir este operador levando-o a ter outro comportamento.

Funções membro x funções globais

Com algumas exceções, os operadores podem ser **funções membro de uma classe** ou então podem ser **funções globais**. Regra geral, de acordo com as boas práticas do encapsulamento, as funções devem ser membro da classe a que dizem respeito, e como tal, os operadores também devem ser membro. Mas, tal como discutido nas aulas teóricas e explicado no livro recomendado, é vantajoso que os operadores binários que não envolvam atribuição (por exemplo, **+** **-** ***** **==** etc. mas não os **+=** ***=** etc=.) sejam definidos como globais pois permite que sejam usados em mais situações.

Exemplo:

O operador **+** pode ser definido como membro ou como não membro.

Desta forma, a expressão **a+b** poderá ser vista como sendo:

- **a.operator+(b)** no caso em que a função é membro
ou
- **operator+(a,b)** no caso em que a função é global

Para que a expressão **a+b** seja aceite, o programador deverá definir uma destas funções, mas não ambas pois tal situação levantaria ambiguidade.

Sendo **a** e **b** objetos da classe **ABC** e **z** um valor/objecto de outra classe que pode ser usado para construir automaticamente um objeto de **ABC** através do seu construtor,

- A versão membro permite as expressões **a+b** e **a+z**
e
- A versão global permite as expressões **a+b** , **a+z** e **z+b**
(permite a conversão de **z** para objeto de **ABC** mesmo quando aparece à esquerda do operador, situação que não é suportado quando a função é membro). Pode assim ser usada em mais situações que a versão membro.

Número de parâmetros

No exemplo acima, é importante reparar nos parâmetros da função em ambas as formas membro/não-membro: o operador em questão no exemplo **é binário**, assim, existem sempre **dois objetos** com os quais se está a trabalhar.

- No caso da função membro, o operando à esquerda do operador é o objeto sobre o qual a função é chamada; o operando à direita é o parâmetro da função, obtendo-se dois objetos correspondentes aos dois operandos.
- No caso da função global, não existe nenhum “objeto sobre o qual a função é chamada”, uma vez que a função é global. Assim, ambos os operandos são passados como parâmetro da função. O operando À esquerda é o primeiro parâmetro, o operando à direita é o segundo parâmetro.

Em ambos os casos, há sempre duas coisas a serem manipuladas pela função, uma vez que o operador é binário e esse aspecto não pode ser modificado.

No caso dos operadores unários aplica-se a mesma lógica: na versão membro não é preciso passar nenhum parâmetro, e na versão global passa-se o operando único sobre a forma de parâmetro, resultando em ambos os casos na função ter acesso a um objecto sobre o qual trabalha e que corresponde ao operando único.

Exemplo:

++a será visto como:

- **a.operator++()** na versão membro da classe a que a pertence
ou
- **operator++()** na versão função global.

Independentemente do operador ser binário ou não, relembra-se que não se deve ter simultaneamente as versões membro e global, uma vez que a existência de ambas em simultâneo iria levantar ambiguidade, indo assim contra as regras do *overloading* de funções.

Em linhas gerais, pode-se seguir a seguinte tabela na escolha de membro/global

Operador	Membro / não membro
Todos os operadores unários	Membro
= () -> ->*	Têm sempre que ser membro
+= -= /= *= &= = %= >>= <<=	Membro
Todos os restantes operadores binários	Globais

Os seguintes operadores não podem ser redefinidos

. .* :: ?:

Os operadores podem envolver expressões com mais do que um tipo de dados. Por exemplo

`cout << a` onde **`a`** pertence a uma classe qualquer. Nestas situações, se se optar por implementar o operador como função membro, será membro da classe do objeto que está à esquerda (ficaria **`cout.operator<<(a)`**). Neste caso particular, tal opção não fará sentido pois presumiria a alteração da classe *standard* a que **`cout`** pertence, o que pode não ser possível e de certeza que não é desejável.

Os operadores podem ser, e normalmente são mesmo, usados em expressões compostas, como por exemplo: **`a * (b + c)`**. Nesse caso, o operador **`*`** seria usado entre o objeto **`a`** e o resultado da expressão **`b+c`**. O resultado de **`b+c`** é aquilo que o operador **`+`** retornar. Ou seja, equacionando o operador **`+`** como a função **`operator+`**, aquilo que a função retornar irá tomar o lugar de **`(b+c)`**, devendo o retorno dessa função ser compatível com o uso que lhe é dado. Por outras palavras, aquilo que **`operator+`** retorna deverá ser compatível com o facto de estar a ser usado em **`a * retorno-de-operator+`**. Isto aplica-se tanto às versões membro como às versões globais dos operadores.

O retorno do operador pode ser um valor (ou uma cópia de algo). Esta situação é compatível com muitos cenários. No entanto, se se pretender usar o resultado de um operador no lado esquerdo de uma atribuição, ou de uma forma mais genérica, usar o resultado do operador como alvo de uma modificação, então, o resultado do operador terá que ser uma referência para algo (variável, posição de memória) não constante e que possa ser modificado.

Exemplos:

- **`(a += b) += c;`** Com os parêntesis torna-se evidente que o objetivo desta expressão é somar **`b`** a **`a`** e depois **`c`**. Assim, a primeira parte da expressão, **`a += b`**, terá que retornar uma referência ao **`a`**, já depois de ter sido adicionado com **`b`**. Se operador retornasse uma cópia de **`a`**, o segundo **`+=`** iria afetar essa cópia e não a variável original **`a`**.
- **`++(++a);`** O **`++a`** dentro dos parêntesis é a primeira parte da expressão a ser avaliada. O resultado deverá ser uma referência para o próprio **`a`**, já depois de ter sofrido o incremento, de maneira a que o segundo **`++`** possa incrementar **`a`** uma segunda vez.

É importante ter presente que esta questão do tipo de retorno aplica-se tanto a operadores como a qualquer função.

Resumo

Essencialmente, os operadores são meras funções que são chamadas quando os operadores que lhe correspondem aparecem no código, e os operandos são os parâmetros e/ou objetos sobre os quais são chamadas. No entanto, a forma de uso habitual dos operadores é bastante variada e existem diversas situações particulares que devem ser analisadas caso a caso nas aulas, em casa, com o livro, e com exercícios. Exemplificam-se algumas dessas situações (a lista não é exaustiva):

- **`a++`** e **`++a`** são operadores diferentes (como se distinguem?)
- **`cout << a << b << c`**. Reparar no uso encadeado e no facto de se misturar **`ostream`** com outras classes. A opção de fazer este operador como membro, na prática, é inviável (porquê?)
- **`a=b`** a atribuição entre objetos da mesma classe já é possível por omissão, mas em algumas situações tem que ser redefinida, como por exemplo, em casos que envolvem memória dinâmica.

Existe uma técnica a que se pode recorrer sempre que surge alguma dúvida nas opções quando ao que um determinado operador deve fazer e como deve ser implementado. Consiste em estabelecer uma analogia com o mesmo operador quando aplicado a tipos de dados habituais (por exemplo, inteiros). Fazer com que o operador em questão tenha um comportamento análogo costuma ser uma boa opção.

Exercícios de aplicação - Observações

- Alguns destes exercícios têm muitas alíneas. Destinam-se a garantir que os vários aspetos da linguagem são devidamente explorados e nenhum objetivo do exercício fica por cumprir. Leia os objetivos de cada exercício e verifique que o seu código bate certo com as alíneas e que no fim do exercício usou as características listadas nos objetivos. Peça a ajuda do professor sempre que haja divergências.
 - Pretende-se cobrir o máximo de cenários e situações com os exercícios, mas é impossível abranger todos os pormenores numa mera ficha de exercícios. Faça as suas próprias modificações e experiências tendo por base estes exercícios. Se o fizer durante a aula peça ajuda ao professor para explicar as variações que imaginou sobre os exercícios. Por vezes as variações propostas são bastante interessantes e levantam aspetos relevantes da matéria.
 - Os exercícios têm inicialmente muitas chamadas de atenção quanto ao que se deve fazer ou não fazer. Progressivamente essas chamadas de atenção vão desaparecendo à medida que se vai assumindo um maior conhecimento por parte dos alunos.
-

1. Os números racionais podem ser representados de forma exata (sem perda de precisão) através do quociente de dois valores inteiros, ou seja, uma fração. Pretende-se uma classe, *Fracao*, que permita representar desta forma os números racionais. Pretende-se também que seja possível usar os operadores aritméticos e de comparação habituais e que estes executem as operações aritméticas normais, adaptadas à natureza das frações. A classe tem as seguintes características:

- Tanto o numerador como o denominador serão números inteiros. O denominador será representado por um valor positivo e não nulo. Assim, o sinal da fração será implicitamente o sinal do numerador.
- Deve ser possível construir objetos da classe *Fracao* apenas das seguintes maneiras:
 - Sem especificar nenhum inicializador; neste caso o valor inicial representará a fração 0/1.
 - Especificando um valor inteiro que será o numerador, considerando-se o denominador com o valor 1.
 - Especificando dois valores inteiros: o numerador e o denominador.
- Devem existir funções para obter e para modificar o numerador e o denominador. As funções para obter o numerador e o denominador devem poder ser chamadas mesmo sobre objetos que são constantes.

- a)** Construa a classe com as características pretendidas. Defina uma função *main* para testar a classe. Declare as frações: *a* com o valor $\frac{1}{2}$, *b* com o valor 3, e *c*, constante, com o valor $\frac{3}{4}$. Teste a classe obtendo, modificando e mostrando os valores. Verifique que não consegue chamar as funções para modificar o numerador e denominador sobre o objeto constante *c*, e que consegue chamar as funções para obter.
- b)** Pretende-se que seja possível obter a multiplicação de duas frações, atribuindo o resultado a outra fração através da expressão: $a = b * c$;
- o Existem duas formas de fazer com que esta expressão seja possível: um operador membro e um operador global. Analise as vantagens e desvantagens de cada.
 - o Faça de ambas as formas. Verifique que não consegue ter ambas em simultâneo. Explique porquê.
 - o Coloque cada uma das versões em comentários à vez e teste a funcionalidade da multiplicação com a expressão $a = b * c$;
 - o Explique a razão do número de parâmetros ser diferente nas formas membro e não membro.
- c)** Teste agora a expressão $a * b * c$. Identifique e explique que alterações são necessárias para suportar esta nova versão. Se não for necessário nada, explique porquê. Pela ajuda ao professor se achar a solução inesperada. Nesta alínea considere ambas as versões do operador membro e não membro (sempre uma de cada vez mantendo a outra em comentários).
- d)** Experimente a operação $a = b * 4$; confirme que apesar de não ter nenhum operador $*$ que receba um inteiro como segundo operando, a expressão é possível e o resultado é correto (obtenha e mostre os valores do resultado em *b* no ecrã). Confirme que é possível tanto na versão membro como não membro. Identifique e explique o que se passa. Se tiver dificuldade, peça ajuda ao professor do seu laboratório neste ponto pois é importante.
- e)** Acrescente a palavra chave *explicit* no início do protótipo do construtor da classe *Fração* que recebe um inteiro (ou que pode ser chamado apenas com um inteiro). Volte a tentar a expressão $a = b * 4$ (ou apenas $b * 4$). Confirme já não é possível. Explique a situação. Depois de ter explicado e confirmado com o professor que a sua explicação é correta, remova a palavra *explicit*.
- f)** Experimente agora a expressão $a = 4 * b$ (ou só $4 * b$). Confirme que é possível e correta quando usa a versão não membro do operador, mas com a versão membro não compila. Explique porquê à luz das conclusões das alíneas anteriores. Confirme com o professor e anote as conclusões das alíneas até agora no seu caderno.
- g)** Acrescente agora o código ao seu programa que suporte a seguinte expressão: `cout << a;`
- o Se fizer este operador como membro, será membro de que classe?
 - o Tente fazer o operador como membro. Se não conseguir, explique porque é que não consegue (peça a ajuda ao professor)
 - o Se não fez o operador como membro de uma classe, faça-o como global. Teste a sua funcionalidade.

h) Faça agora com que seja possível fazer `cout << a << b`; Qual é a alteração necessária? Explique qual. Se não for preciso nada, explique porquê.

- o Importante: teste também a expressão `cout << a << c`;

i) No decorrer das duas alíneas anteriores deve ter feito funções que recebem e passam objetos *ostream* por referência. Experimente passá-los por cópia e verifique que não é possível. Explique a forma que os programadores da classe *ostream* usaram pra impedir a passagem e retorno de objetos *ostream* por cópia. Peça ajuda e exemplos ao professor se for necessário.

j) Acrescente um operador que suporte a expressão `a *= b`. Uma vez que já passou pelas alíneas anteriores, em princípio já entende a diferença entre operadores membro e não membro. Assim faça logo o operador desta classe como membro. Teste o seu operador imprimindo o seu conteúdo no ecrã (através da expressão `cout << a`) e confirmando que os valores são os esperados.

k) Qual o tipo de retorno que o seu operador `*=` tem? Verifique se consegue fazer `a *= b *= c`. Analise com cuidado a expressão tendo em atenção que a associatividade do operador `*=` é da direita para a esquerda, ou seja, é como se fosse `a *= (b *= c)`;

l) Como resultado da alínea anterior, deverá ter um operador `*=` que retorna um objeto *Fraccao* por cópia. Experimente agora o seguinte código:

```
Fraccao a(1,2), b(2,3), c(3,4);
(a *= b) *= c;
cout << a
// é suposto aparecer 6/24
```

Aparece o resultado esperado? Se aparecer 2/6 é porque o seu operador não está totalmente correto em relação à forma pretendida. Veja o protótipo da função correspondente ao operador e confirme com o professor se não conseguir obter o resultado correto. Este especto é importante.

m) Acrescente à sua classe o suporte para as expressões `a++`; e `++a`; Depois de ter este especto a funcionar, experimente agora `c++`; e `++c`; e confirme que o compilador não deixa compilar essas duas expressões.

n) Considere o código abaixo. Pretende-se que funcione. Deve modificar algo na sua classe de forma a que o código abaixo funcione. Não pode alterar nada no código apresentado.

```
void func(int n) {
    cout << n; // aparece 2
}
int main() {
    const Fraccao f(7,3);
    Func(f); // é passado automaticamente o valor 7/3
             // arredondado para baixo
    return 0;
}
```

- o)** Pretende-se que a expressão *if (a == b)* seja aceite e funcione como esperado. Faça com que isso aconteça.
- p)** Antes que se esqueça, escreva no seu caderno todas as conclusões acerca de operadores obtidas ao longo deste exercício.
- q)** Para consolidar os seus conhecimentos acerca de operadores até agora obtidos e para concluir o exercício, faça com que o código apresentado abaixo compile e tenha o resultado intuitivamente esperado. Esta alínea pode ficar para trabalho de casa.

```
int main() {
    Fraccao x(2,1), y(1,3), z;
    cout << " z= " << z << endl;
    z=x*y;
    cout << x << " * " << y << " = " << z << endl;
    z = x/y;
    cout << x << " / " << y << " = " << z << endl;

    Fraccao a(2,-4), b(2);
    cout << " a= " << a << " b= " << b << endl;
    a *= b;
    cout << "a *= b " << endl;
    cout << "a= " << a << " b= " << b << endl;

    cin >> a;
}
```

Objetivos do exercício

- Introdução ao conceito de redefinição de operadores.
- Experimentar os operadores nas suas variadas formas: binários e unários, membro e não-membro.
- Entender as diferenças de aplicação entre operadores binários membro e operadores binários não membro, e as vantagens e desvantagens de cada uma das duas opções.
- Experimentar a redefinição de operadores no contexto do seu uso em expressões compostas.
- Experimentar a redefinição de operadores em situações em que são usados objectos de duas classes diferentes.
- Perceber a influência de passar e retornar objectos por cópia e por referência no contexto de redefinição de operadores.
- Entender o conceito de construção implícita e a forma como pode poupar código na redefinição e operadores (e funções em geral).
- Conhecer e experimentar os operadores de conversão.
- Treinar a redefinição de operadores



2. Considere os vectores da geometria analítica (não se trata dos vectores para guardar coisas de C++). Um vector é um segmento de recta com origem nas coordenadas 0,0 (sempre, e por isso não é preciso guardar esses valores) e por um ponto término nas coordenadas x,y . Assim, um vector é definido pelas coordenadas x,y . Pretende-se uma classe em C++ cujos objectos representem pontos. A classe deve cumprir o seguinte:

- Apenas deve ser possível construir objectos desta classe mediante:
 - A indicação de ambas as suas coordenadas (nota: "indicação" \neq "perguntar ao utilizador"). Qualquer valor inteiro é válido, tanto para x como para y .
 - A indicação de apenas um valor. Neste caso ambas as coordenadas ficam com esse valor.
- Deve ser possível obter e modificar cada uma das coordenadas, mas sem desrespeitar o conceito de encapsulamento. As funções que permitem obter os dados devem poder ser chamadas sobre objectos da classe constantes, e as que modificam as coordenadas não.
- Obter um objecto *string* com a descrição textual do seu conteúdo (formato: "(x,y)").

a) Escreva a classe *Vector* com as características enunciadas. Não inclua o *header* `<vector>` para evitar confusões entre o vector da STL e o vector deste exercício. Teste a classe através de uma função *main* que tenha dois vectores *a* e *b* com coordenadas (1,2) e (3,4). Confirme que não é possível ter vectores sem especificar as suas coordenadas.

b) Defina operadores (aritméticos, comparação para a igualdade/desigualdade, ...) que permitam a utilização da classe *Vector* expressa na seguinte função *main*:

```
int main() {
    Vector v1(2,1), v2(1,3), z;
    z = v1 + v2;
    cout << v1 << " + " << v2 << " = " << z << endl;
    z = v1 + Vector(10);
    z = 10 + v1;
    cout << v1 << " + " << " 10 = " << z << endl;
    z = v1 - v2;
    cout << v1 << " - " << v2 << " = " << z << endl;
    Vector a(1,1), b(2,4);
    cout << " a= " << a << " b= " << b << endl;
    a += b += v1;
    a += b;
    a += 10;
    cout << "a += b " << " a= " << a << endl;
    cout << "(a == b)? " << (a == b) << endl;
    cout << "(a != b)? " << (a != b) << endl;
}
```

c) Indique **duas** maneiras que permitem tornar possível a instrução: **$z=p1+10$** ;

d) Defina operadores (aritméticos, comparação para a igualdade/desigualdade, ...) que permitam a utilização da classe *Vector* expressa na seguinte função *main*:

```

int main() {
    Vector a(1,1);
    int n = int(a);
    int k = a;
    Vector b = 2;
    b = a + 4; // se fizer um operador para este caso dá erro
    Vector c(1,1);
    cout << "\n Operadores unários \n";
    cout << "\nc:" << c;
    cout << "\n++c:" << ++c;
    cout << "\nc:" << c;
    Vector d(1,1);
    cout << "\nd:" << d ;
    cout << "\nd++:" << d++;
    cout << "\nd:" << d << endl;
    return 0;
}

```

- e)** Explique qual a razão pela qual a instrução: `b = a + 4;` dá erro se fizer um operador explicitamente para o caso *Vector + int*. Existem duas formas distintas de remover o erro. Concretize ambas.

Objetivos do exercício

- Consolidar a matéria de redefinição de operadores



- 3.** Considere o conceito de *Automovel*. Um automóvel tem vários atributos (defina alguns), entre os quais se inclui a matrícula (*string*).
- a)** Construa a classe especificada, incluindo um construtor que faça sentido em relação ao significado de “Automóvel” e em relação aos dados que definiu.
- b)** Pretende-se que se traga para o programa a noção de “fazer um automóvel igual a outro”, tal como na vida real. Isto significa que se pretende ter a cor, os extras, etc., iguais ao “outro”, mas há uma coisa que nunca muda, que é a matrícula. Faça com que seja possível efectuar a atribuição entre dois automóveis com a expressão habitual `a = b`, mas em que a matrícula nunca é modificada no automóvel do lado esquerdo da atribuição.
- c)** Suponha que existe um contador de números de carros construídos. Pretende-se que esse contador leve em atenção as situações em que um automóvel é construído no contexto de um parâmetro de função passado por cópia. Acrescente/modifique o necessário à sua classe para que esta característica seja cumprida.

Objetivos do exercício

- Consolidar a matéria de redefinição de operadores.
- Contacto com o operador de atribuição.
- Construtor por cópia



4. Considere o seguinte programa:

```
class Solidos{
    double volume;
    static int n;
public:
    Solidos(double v);
    ~Solidos();
    static int getN();
};

int Solidos::n=0;

Solidos::Solidos(double v) {
    volume = v;
    ++n;
    cout<<"\nConstruindo";
}

Solidos::~~Solidos() {
    --n;
    cout<<"\nDestruindo";
}

int Solidos::getN(){
    return n;
}

void f( ) {
    Solidos x(4.4), y(5.5);
    cout<<"nC: " << x.getN();
    cout<<"\nD: " << Solidos::getN();
}

int main() {
    cout<<"\nA: "<<Solidos::getN();
    Solidos a(6.6);
    cout<<"\nB: "<<Solidos::getN();
    f( );
    cout<<"\nE: "<<Solidos::getN();
}
```

a) Qual será a saída resultante da execução deste programa? Explique o seu output.

b) Faça as alterações necessária à classe para que esteja correcta a instrução seguinte:

```
Solidos *p=new Solidos[4];
```

c) De acordo com a definição da classe Solidos, indique, justificando, quais das seguintes instruções estariam correctas ou incorrectas. Relativamente a cada instrução que considerar incorrecta, indique como poderia a classe solidos "adaptar-se" de modo a eliminar o erro sem alterar a referida instrução.

- `double x=12.34;`
- `Solidos ob;`
- `ob=x;`
- `ob += x;`

Objetivos do exercício

- Consolidar a matéria de redefinição de operadores.
- Consolidar conversões implícitas via operador de conversão.
- Consolidar construções implícitas via construtor.

