

Capítulo 4

Sobrecarga de operadores

O uso de funções sobrecarregadas não só uniformiza chamadas de funções para diferentes objetos como também permite que os nomes sejam mais intuitivos. Se um dos objetivos da sobrecarga é permitir que as funções sejam chamadas pelo nome mais natural possível, não importa se o nome já foi usado, porque não deixar o programador sobrecarregar também os operadores?

Na realidade, um operador executa algum código com alguns parâmetros, assim como qualquer função. A aplicação de um operador é equivalente à chamada de uma função. Em C++ é permitido sobrecarregar um operador, com o objetivo de simplificar a notação e uniformizar a expressão.

Existem duas maneiras de implementar operadores para classes de objetos: como funções membro e como funções globais. Por exemplo, dado um objeto `w` e um operador unário `!`, a expressão

```
!w
```

é equivalente às chamadas de funções

```
w.operator!();    // usando uma função membro
operator!(w);     // usando uma função global
```

Vejamos agora como seria com um operador binário, por exemplo, `&`. A expressão

```
x & y
```

é equivalente às chamadas

```
x.operator&(y); // usando uma função membro
operator&(x,y); // usando uma função global
```

Um detalhe importante é que uma função `y.operator&(x)` nunca será considerada pelo compilador para resolver a expressão `x&y`, já que isto implicaria que o operador é comutativo.

Antes do primeiro exemplo, precisamos ter em mente que C++ não permite a criação de novos operadores; só podem ser redefinidos os operadores que já existem na linguagem. Isto implica que, por exemplo, o operador `/` será sempre binário. Outra característica é que a prioridade também não pode ser alterada, é preservada a original do C.

Um número complexo pode ser modelado com uma classe que permita que as operações matemáticas sobre ele sejam feitas da mesma maneira que os tipos primitivos, ou seja, com os operadores `+`, `-` etc. Uma possibilidade seria:

```
class Complex {
public:
    Complex operator+ (const Complex&);
    Complex operator- (const Complex&);
};
```

Com estes operadores, é possível fazer:

```
void main()
{
    Complex c1, c2, c3;
    c1 = c2 + c3;
}
```

Exercício 5 - Classe Complex

Implementar uma classe que represente um número complexo.

Operadores como funções globais

Suponhamos que a classe que modela complexos possui um construtor da forma:

```
Complex(float re = 0.0, float im = 0.0);
```

Este construtor permite criar um complexo especificando suas partes real e imaginária, só especificando a parte real ou ainda sem dizer nada. Em particular, este construtor define como converter

um float em um complexo, o que é equivalente a chamar o construtor com apenas um parâmetro. Esta conversão permite expressões do tipo:

```
c1 = c2 + 3.0; // equivalente a c1 = c2 + Complex(3.0, 0.0)
```

Considerando que o operador é uma função, esta expressão poderia ser vista como:

```
c1 = c2.operator+(Complex(3.0,0.0));
```

A conversão foi feita porque a função `operator+` espera um *Complex*, e o valor era um float. Nesse caso o compilador converte automaticamente o valor. Mas e se a expressão for a seguinte:

```
c1 = 3.0 + c2;
```

Nesse caso 3.0 não é parâmetro de função nenhuma, então a conversão não é feita. Para possibilitar esta expressão, seria preciso converter o valor explicitamente:

```
c1 = Complex(3.0) + c2;
```

No entanto, se 3.0 fosse o parâmetro para alguma função, o compilador saberia fazer a conversão. Lembrando que os operadores podem ser definidos como métodos ou como funções globais, é possível tornar 3.0 um parâmetro. É o caso de definir o operador como uma função global:

```
Complex operator+ (const Complex&, const Complex&);
```

Com esta função definida, os dois operandos passam a ser parâmetros, e ambos podem ser convertidos automaticamente.

Operadores que podem ser redefinidos

A maior parte dos operadores podem ser sobrecarregados. São eles:

```
new delete  
+ - * / % ^& | ~  
! = < > += -= *= /= %=  
^= &= |= << >> >>= <=<= == !=  
<= >= && || ++ -- , ->* ->  
( ) [ ]
```

Tanto as formas unárias como as binárias de

```
+ - * &
```

podem ser sobrecarregadas, assim como as formas pré-fixadas ou pós fixadas de

```
++ --
```

Os seguintes operadores não podem ser sobrecarregados:

```
. .* :: sizeof ?:
```

já que estes operadores já tem um significado predefinido (exceto ?:) para objetos de qualquer classe.

A função de atribuição `operator=()` é definida por default para todos os objetos como a atribuição byte a byte dos campos do objeto.

Exemplo de redefinição do operador [] - classe Vector

As estratégias usadas para redefinições variam muito de acordo com o operador. Esta seção apresenta um exemplo que traz vários detalhes que devem ser levados em consideração dependendo do operador. O exemplo encapsula um vetor como uma classe com o objetivo de checar as indexações, evitando que posições aleatórias da memória sejam acessadas. Um vetor normal de C++ permite a indexação com qualquer inteiro; mesmo se o índice estiver além da área alocada o acesso é permitido, com consequências imprevisíveis.

O operador `[]` será redefinido para permitir o uso dos objetos desta classe da mesma maneira que um vetor C++. Além do operador, a classe terá um construtor que aloca os elementos do vetor. O funcionamento será o seguinte: o construtor, além de alocar os elementos, guarda o tamanho do vetor. O operador, que retorna o elemento correspondente do vetor alocado, checa se o índice é válido (o construtor guarda o tamanho) antes de fazer o acesso. Alguma coisa da forma:

```
float Vector::operator[](int i)
{
    if (i>=0 && i<size) return elems[i];
    else
    {
        printf("índice %d inválido\n", i);
        return -1.0;
    }
}
```

No entanto isto não é suficiente. É preciso lembrar que este operador pode ser usado de duas maneiras:

```
a = vetor[10];
vetor[20] = b;
```

Na primeira linha não há problema. O operador é uma função que retorna um valor, que será atribuído à variável *a*. Já na segunda linha a atribuição não é permitida, pois a função retorna o valor da posição 20 do vetor; para a atribuição é necessário saber o endereço da posição 20 do vetor. A solução é retornar uma referência para um float. Assim o valor de retorno é o endereço (necessário na segunda linha), mas este é usado como um valor (primeira linha). Se simplesmente mudarmos o tipo de retorno da função para float&, um erro será sinalizado durante a sua compilação, pois o valor -1.0, retornado caso o índice seja inválido, não tem endereço. Ou seja, é necessário retornar uma variável, mesmo nesse caso. Outra particularidade deve ser observada aqui: como que será retornado será o endereço da variável retornada, esta precisa continuar existindo mesmo depois que a função termina a sua execução. O que significa que não se pode retornar uma variável local, pois variáveis locais deixam de existir assim que a função termina. A implementação abaixo utiliza uma variável static dentro da função só para este fim:

```
class Vector {
    float *elems;
    int    size;
public:
    Vector(int s);
    float& operator[](int i);
};

Vector::Vector(int s)
{
    size=s;
    elems = new float[size];
}

float& Vector::operator[](int i)
{
    if (i>=0 && i<size) return elems[i];
    else
    {
        static float lixo;
        printf("índice %d inválido\n", i);
        return lixo;
    }
}
```

Operadores de conversão

É possível definir operadores especiais para conversão de tipos. Além das conversões padrão, o programador pode definir como um objeto pode ser convertido para algum outro tipo. Consideremos uma classe *Arquivo* que modela um arquivo. Internamente esta classe pode ter um ponteiro para um arquivo (tipo FILE*) privado. Se fosse necessário fazer alguma operação já definida na biblioteca que não tenha sido mapeada na classe, seria preciso ter acesso ao ponteiro para arquivo. Ao invés de tornar público este campo, seria mais elegante definir como um objeto do tipo *Arquivo* se converte em um FILE*, que é o objetivo. Esta definição se dá da seguinte forma:

```

class Arquivo {
    FILE *file;
public:
    Arquivo( char* nome ) { file=fopen(nome, "r"); }
    ~Arquivo() { fclose(file); }
    char read() { return file?fgetc(file):EOF; }
    int aberto() { return file!=NULL; }
    operator FILE*() { return file; }
}

void main()
{
    int i;
    Arquivo arq("teste.c");
    fscanf( (FILE*)arq, "%d", &i );
}

```

Exercício 6 - Classe Complex com operadores globais

Alterar a classe Complex para permitir expressões do tipo $1+c$, onde c é complexo.

Exercício 7 - Calculadora RPN para complexos

Modificar a calculadora para operar com números complexos.

Exercício 8 - Classe String

Implementar uma classe String. A classe deve permitir comparações, concatenações, atribuições e acesso a cada caracter, além de poder ser utilizada em funções como printf. O usuário desta classe não precisa se preocupar com tamanho de memória alocado. Ou seja, se for necessário realocar memória para uma atribuição ou uma concatenação, esta realocação deve ser feita automaticamente dentro da classe. O acesso aos caracteres deve ser seguro. Por exemplo, no caso de a string ter 10 caracteres e o usuário tentar escrever no vigésimo, isto não pode alterar uma área de memória aleatória.

