

Sobrecarga de operadores

Definição

A sobrecarga de operadores permite a definição de novas operações para os operadores padrões da linguagem C++. A maior vantagem da sobrecarga é aumentar a abstração e facilitar a legibilidade do código. Imagine uma classe `Matriz` com as operações de soma, subtração e multiplicação implementadas. Considere a seguinte expressão a ser implementada:

$$resultado = m_1 + m_2 + (m_3 - m_4) * m_5$$

A implementação ficaria algo da seguinte maneira:

```
Matriz parte1 = m1.soma(m2);  
Matriz parte2 = m3.subtracao(m4);  
Matriz parte3 = parte2.multiplicacao(m5);  
Matriz resultado = parte1.soma(parte3);
```

Com a sobrecarga de operadores é possível associar aos operadores as operações a serem realizadas, aumentando assim o poder de expressividade da linguagem. A expressão anterior poderia ser reescrita da seguinte maneira:

```
Matriz resultado = m1 + m2 + (m3 - m4) * m5;
```

Operadores

Em C++, os operadores podem ser sobrescritos são descritos a seguir [\[1\]](#).

Operador	Descrição
!	não lógico
&	endereço
*	desreferenciação de ponteiros
+	positivo
-	negativo
++	pré/pós-incremento
--	pré/pós-decremento
+	adição
-	subtração
*	multiplicação
/	divisão
%	resto
==	igualdade
!=	diferença
>	maior
>=	maior ou igual
<	menor

<code><=</code>	menor ou igual
<code>&&</code>	e lógico
<code> </code>	ou lógico
<code>=</code>	atribuição
<code>+=</code>	adição e atribuição
<code>-=</code>	subtração e atribuição
<code>*=</code>	multiplicação e atribuição
<code>/=</code>	divisão e atribuição
<code>%=</code>	resto e atribuição

Outros operadores que podem ser sobrescritos são:

- Chamada de função: `()`
- Subscrito de matriz: `[]`
- Bit a bit: `&`, `|`, `^`, `~`, `>>`, `<<`
- Acesso: `->`, `->*`
- Instanciação: `new`, `new []`, `delete`, `delete []`

Exemplo de implementação

```
class Ponto {
public:
    Ponto(x, y) : _x(x), _y(y) {}

    // -----
    // Operadores unários
    // -----
    Ponto operator-() {
        return Ponto(-_x, -_y);
    }
}
```

```

}

// -----
// Operadores ++ e --
// -----
// pré-fixado: ++ponto
Ponto& operator++() {
    _x++;
    _y++;
    return *this;
}

// pós-fixado: ponto++
Ponto operator++(int) {
    Ponto antigo = *this;
    _x++;
    _y++;
    return antigo;
}

// -----
// Operadores binários
// -----
Ponto operator+(const Ponto &outro) {
    return Ponto(_x + outro._x, _y + outro._y);
}

Ponto operator-(const Ponto &outro) {
    return Ponto(_x - outro._x, _y - outro._y);
}

// -----
// Operadores relacionais
// -----
bool operator==(const Ponto &outro) {
    return (_x == outro._x) and (_y == outro._y)
}

bool operator!=(const Ponto &outro) {
    return !operator==(outro)
}

```

```

// -----
// Operadores de atribuição
// -----
void operator=(const Ponto &outro) {
    _x = outro._x;
    _y = outro._y;
}

void operator+=(const Ponto &outro) {
    _x += outro._x;
    _y += outro._y;
}

void operator*=(int escalar) {
    _x *= escalar;
    _y *= escalar;
}

// -----
// Operadores () e []
// -----
void operator()(int a, int b) {
    _x = a;
    _y = b;
}

int operator[](int i) {
    if (i == 0) return _x;
    if (i == 1) return _y;
    return -999;
}

private:
    int _x;
    int _y;
}

```



Atividade prática

1. Crie uma classe Pilha com as seguintes operações:
 - `+=` : adiciona um elemento ao topo da pilha
 - `--` : remove o topo da pilha e retorna ele
 - `()` : acessa o topo da pilha
 - `==` , `!=` : compara se as pilhas são iguais ou diferentes
 - `>` , `<` , `>=` , `<=` : compara os tamanhos de duas pilhas



Leitura recomendada

- Capítulo 10: MIZRAHI, Vctorine Viviane. **Treinamento em Linguagem C++ - Módulo II**. Makron Books, 1994.



Referência bibliográficas

- [1] MIZRAHI, Vctorine Viviane. **Treinamento em Linguagem C++ - Módulo II**. Makron Books, 1994.