Aula 06:

Sobrecarga de Operadores
Parte II

ECOP03 - Programação Orientada a Objetos
Prof. André Bernardi
andrebernardi@unifei.edu.br

Universidade Federal de Itajubá

Rev. Agosto 22

Sobrecarga de Operadores

E com os operadores unários, como funciona?

Também podem ser sobrecarregados através de funções membro não-*static* sem argumentos ou uma função global (*friend*) cujo único argumento deve ser um objeto do tipo da classe.

> Podemos, por exemplo, sobrecarregar o operador unário de negação **!** para verificar se um ponto possui o valor (0,0) ou diferente disso. O operador pode retornar true ou false.

Podemos ainda, sobrecarregar os operadores unários de incremento (**++**) e decremento (**--**), para que somem uma unidade em x e y de cada Point.

Declaração


```
// operadores unários
bool operator!() const;
Point& operator++(); //pre-incremento

// operador unário friend
friend Point& operator--(Point&); //pre-decremento
```

Implementação


```cpp
// operadores unários
bool Point::operator!() const {
    if(x == 0 && y == 0) return true;
    return false;
}

Point& Point::operator++() {
    x++;
    y++;
    return *this;
}

// operador unário friend
Point& operator--(Point& p) {
    p.x--;
    p.y--;
    return p;
}
```

```cpp
int main()
{
    Point p1{20, 20};
    Point p2 = p1 + 10; // chama Point::operator+(int value)

    ++p1;
    --p2;

    cout << "p1: (" << p1.get_x() << ", " << p1.get_y() << ")\n";
    cout << "p2: (" << p2.get_x() << ", " << p2.get_y() << ")\n";

    Point p3{0,0};

    cout << "p1 esta com valor " << (!p1 ? "zero" : "nao-zero") << "\n";
    cout << "p3 esta com valor " << (!p3 ? "zero" : "nao-zero") << "\n";
}
```

Repare que **++** e **--** podem ser tanto operadores de pré-incremento (decremento), como o do exemplo, quanto pós-incremento(decremento), quando escrevemos, por exemplo **p1**++ ou **p2**--.

Neste caso, apenas implementamos o pré-(incremento/decremento).

Sobrecarga de Operadores

Os operadores de incremento e decremento entram em uma categoria especial porque há **duas** variantes de cada:

- Pré-incremento e Pós-incremento;
- Pré-Decremento e Pós-Decremento.

Quando escrevemos a função de sobrecarga do operador, pode ser útil implementar versões separadas para a forma prefixo e sufixo destes operadores. Para distinguir entre elas, é observada a regra seguinte:

- A forma prefixo do operador é declarada exatamente do mesmo modo como qualquer outro operador unário;
- A forma sufixo aceita um argumento adicional de tipo **int**.

```
Point operator++(int); // pos-incremento
friend Point operator--(Point&, int); //pos-decremento
```

```
Point Point::operator++(int value) {
    Point temp = *this;
    ++(*this);
    return temp;
}

// operador unário friend
Point operator--(Point& p, int value) {
    Point temp = p;
    --p;
    return temp;
}
```

Os operadores de sufixo realizam o incremento, mas retornam uma referência intacta do objeto chamador. Lembre-se de que o pós-(incremento/decremento) será realizado somente ao final da expressão.

Importante: Quando especificar um operador sobrecarregado para o sufixo do incremento ou decremento, o argumento adicional deve ser do tipo **int**; especificando qualquer outro tipo gera erro.

Operadores de conversão de tipo

A maioria dos programas processa informações de muitos tipos. Às vezes, todas as operações “permanecem dentro de um tipo”. Por exemplo, somar um **int** a um **int** produz um **int**. Porém, com certa frequência, é necessário converter dados de um tipo para outro.

Isso pode acontecer em atribuições, cálculos, passagem de valores para funções e retorno de valores de funções.

O compilador sabe como fazer algumas conversões entre tipos fundamentais, é possivel utilizar operadores de conversão para forçar conversões entre tipos fundamentais (*cast*).

Operadores de conversão de tipo

Mas e os tipos definidos pelo usuário?

Os operadores de conversão de tipo proporcionam um mecanismo para conversão de um objeto de determinada classe em um outro tipo. A sintaxe de um operador de conversão de tipo assemelha-se bastante à sintaxe de um operador sobrecarregado.

Por exemplo,

```
A::operator int() const;
A::operator OutraClasse() const;
```

convertem um objeto do tipo A para o tipo fundamental *int* e para um objeto do tipo *OutraClasse*.

Veja como fazer uma conversão do tipo **Point** para **int**:

```
public:
    Point(int xx = 0, int yy = 0) : x{xx}, y{yy} {}
    ~Point() {}
    operator int();
    // Repare que não é necessário especificar o tipo de retorno
    // O tipo de retorno está implícito.
```

```
// função converte Ponto para int
// retornando o valor do módulo da coordenada
Point::operator int()
{
    return sqrt(x*x + y*y);
}
```

```
// Para realizar a conversão, utilize o operador
// de cast, como já estamos acostumados
cout << "Modulo de " << p2 << ": " << (int)p2;
```

```
Modulo de (6, 8): 10
```

Este tipo de conversão aconteceria tanto através da utilização *explícita* do operador de conversão (int), quanto em uma atribuição ou passagem de parâmetros. A este tipo, chamamos de conversão *implícita*.

```
Point p2{6,8};
int x = p2; // converte implicitamente (compilador "pressupõe")
cout << "Valor de x = " << x << "\n";
```

Caso o criador do tipo de dados queira evitar este tipo de conversão implícita, restringindo a conversão apenas a manifestações explícitas de intenção, é necessário declarar o operador de conversão como ***explicit***. Veja:

```
explicit operator int(); // somente permite conversões explicitas
```

```
int x = (int)p2; // necessário tornar explícito
```

Operadores de comparação

Sobrecarga de operadores de comparação é muito simples uma vez que já tenhamos entendido tudo sobre operadores aritméticos. Veja as opções:

Operator name	Syntax	Over loadable	Prototype examples (for `class T`)	
			As member function	As free (namespace) function
equal to	`a == b`	Yes	`bool T::operator ==(const T2 &b) const;`	`bool operator ==(const T &a, const T2 &b);`
not equal to	`a != b`	Yes	`bool T::operator !=(const T2 &b) const;`	`bool operator !=(const T &a, const T2 &b);`
less than	`a < b`	Yes	`bool T::operator <(const T2 &b) const;`	`bool operator <(const T &a, const T2 &b);`
greater than	`a > b`	Yes	`bool T::operator >(const T2 &b) const;`	`bool operator >(const T &a, const T2 &b);`
less than or equal to	`a <= b`	Yes	`bool T::operator <=(const T2 &b) const;`	`bool operator <=(const T &a, const T2 &b);`
greater than or equal to	`a >= b`	Yes	`bool T::operator >=(const T2 &b) const;`	`bool operator >=(const T &a, const T2 &b);`

Notes

- All built-in operators return `bool`, and most user-defined overloads also return `bool` so that the user-defined operators can be used in the same manner as the built-ins. However, in a user-defined operator overload, any type can be used as return type (including `void`).
- T2 can be any type including T

Da tabela anterior, podemos ver que:

- É possível sobrecarregar qualquer um dos seguintes operadores de comparação: ==, !=, <, >, <= e >=.

- Para que os operadores funcionem de maneira semelhante aos utilizados nos tipos fundamentais, é recomendado que o tipo de retorno das funções seja do tipo *bool*.

- A sobrecarga do operador == não implica em uma sobrecarga implícita do operador !=.

- É possível fazer a comparação entre objetos de dois tipos diferentes T e T2 ou entre objetos de um mesmo tipo T (T2 == T)

Exemplo:

```
// faz a comparação utiizando uma função membro
bool operator==(Point&);
// faz a comparação com função friend
friend bool operator!=(Point&, Point&);
```

```
bool Point::operator==(Point& p)
{
    if(x == p.x && y == p.y) return true;
    return false;
    // um simples return (x == p.x && y == p.y); bastaria.
}

bool operator!=(Point& p1, Point& p2)
{
    if(!(p1 == p2)) return true;
    return false;
}
```

```
int main() {
    Point p1{10,10};
    Point p2{10,10};

    cout << p1 << " e " << p2;
    cout << " sao " << (p1 == p2 ? "iguais" : "diferentes");
    cout << "\n";

    Point p3{10,11};
    if(p1 != p3)
        cout << p1 << " e " << p3 << " sao diferentes.\n";
}
```

Veja como torna-se simples a utilização de operadores de comparação utilizando tipos definidos pelo usuário. O critério para “igualdade” e “diferença” está encapsulado na implementação do próprio tipo. **É uma comparação completamente segura**.

Agora vejamos um exemplo diferente para a aplicação de operadores de comparação < e >.

```
class Classroom {
    private:
        string *names;
        int number_students;

    public:
        Classroom(int s) : number_students{s} {
            names = new string[s];
        }

        ~Classroom() { delete[] names;  }

        bool operator<(Classroom& c) {
            return (number_students < c.number_students);
        }

        bool operator>(Classroom& c) {
            if(number_students > c.number_students) return true;
            return false;
        }
};
```

Esta classe define que “tamanho” de uma classe de aula está relacionado apenas ao **número de estudantes**, e é somente isso que ela compara. Existe outro membro de dados, que é um ponteiro para strings com os nomes dos alunos. Ele não contém informação relevante para a comparação.

```
#include "classroom.h"
#include <iostream>

using namespace std;

int main()
{
    Classroom c1{100};
    Classroom c2{80};

    if(c1 < c2) cout << "c1(100) < c2(80)";
    if(c1 > c2) cout << "c1(100) > c2(80)";
}
```

```
c1(100) > c2(80)
```

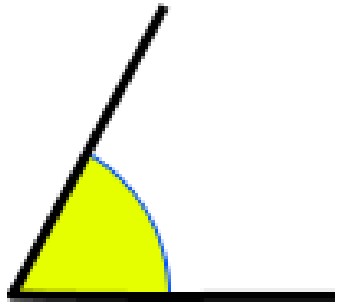
O mesmo tipo de lógica poderia ser aplicada nos operadores <= e >=, para tornar o tipo mais completo.

Hands-On

Vamos implementar agora um tipo completo, com seus membros de dados, funções membro e operadores sobrecarregados.

O tipo no qual estou interessado é “Ângulo”, para ser utilizados em problemas de trigonometria. Veja a definição:

Euclides definiu um ângulo plano como
a inclinação entre duas linhas que se encontram em um mesmo plano. Pode ser medido em **graus** ou **radianos**.




```h
#ifndef ANGULO_H
#define ANGULO_H

#include <iostream>
using namespace std;

class Angulo
{
    private:
        double grau;

    public:
        Angulo(double=0);
        ~Angulo() {}

        // trigonometricas
        double radiano();
        double seno();
        double cosseno();
        double tangente();
        double complementar();

        // aritmeticas
        Angulo operator+(Angulo&);
        Angulo operator-(Angulo&);

        // entrada e saida
        friend ostream& operator<<(ostream&, const Angulo&);
        friend istream& operator>>(istream&, Angulo&);

        // conversao de tipo
        explicit operator double() const;
};

#endif
```

```cpp
#include "angulo.h"
#include <cmath>

#define PI 3.141592

Angulo::Angulo(double g) {
    grau = fmod(g, 360);
}

double Angulo::radiano()
{
    return grau*PI/180;
}

double Angulo::seno()
{
    return sin(radiano());
}

double Angulo::cosseno()
{
    return cos(radiano());
}

double Angulo::tangente()
{
    return tan(radiano());
}

double Angulo::complementar()
{
    if(grau > 90) return -1;
    return (90-grau);
}
```

```cpp

Angulo Angulo::operator+(Angulo& a)
{
    return Angulo{fmod((grau+a.grau),360)};
}

Angulo Angulo::operator-(Angulo& a)
{
    return Angulo{fmod((grau-a.grau),360)};
}

ostream& operator<<(ostream& saida, const Angulo& a)
{
    saida << a.grau;
    return saida;
}

istream& operator>>(istream& entrada, Angulo& a)
{
    entrada >> a.grau;
    return entrada;
}

        // conversao de tipo
Angulo::operator double() const
{
    return grau;
}

```


```cpp
#include "angulo.h"

int main()
{
    Angulo a1{45};
    Angulo a2;
    Angulo a3{170};
    Angulo a4{180};
    Angulo a5{390};

    cout << "a5 = " << a5 << "\n";
    cout << "a2 = " << a2 << "\n";
    cout << "Entre com a2: ";
    cin >> a2;
    cout << "a2 = " << a2 << "\n";

    cout << a2 << " graus = " << a2.radiano() << " radianos";
    cout << "sen(" << a2 << ") = " << a2.seno() << "\n";
    cout << "cos(" << a2 << ") = " << a2.cosseno() << "\n";
    cout << "tan(" << a2 << ") = " << a2.tangente() << "\n";
    cout << "complemente de " << a2 << " = " << a2.complementar() << "\n";

    double difference = (double)(a4-a3);
    cout << a1 << " + " << a3 << " = " << a1+a3 << "\n";
    cout << a4 << " - " << a3 << " = " << difference << "\n";

}
```

```
a5 = 30
a2 = 0
Entre com a2: 60
a2 = 60
60 graus = 1.0472 radianossen(60) = 0.866025
cos(60) = 0.5
tan(60) = 1.73205
complemente de 60 = 30
45 + 170 = 215
180 - 170 = 10
```

Referências

- https://cplusplus.com/reference/

- Notas de aula da disciplina Programação Orientada a Objetos, Prof. André Bernardi, Prof. João Paulo Reus Rodrigues Leite.