# Sobrecarga de Operadores

Roland Teodorowitsch

Programação Orientada a Objetos - ECo - Curso de Engenharia de Computação - PUCRS

11 de setembro de 2023

# Sobrecarga de Operadores

# Sobrecarga de Operadores

- A sobrecarga de um operador deve ser feita criando-se uma função ou método cujo nome deve iniciar pela palavra `operator`
- Exemplos:
  - Para sobrecarregar o comportamento do operador de soma o nome da função/método deve ser `operator+`
  - Para sobrecarregar o operador "`>`" a função seria `operator>`
- A sobrecarga de um operador para objetos de determinada classe pode ser feita **dentro** (com métodos) ou **fora** (com funções) da respectiva classe

# Sobrecarga de Operadores com Funções

- Quando a sobrecarga for feita com uma função, ou seja, **fora de uma classe**, esta função deverá ter:
  - 2 parâmetros caso o operador seja binário
  - 1 parâmetro caso o operador seja unário

# Sobrecarga de Operadores Binários com Funções

- No caso de um operador **binário**, como o de multiplicação (*), o primeiro parâmetro desta função representará o operando que fica à esquerda do operador e o segundo representará o operando que fica à direita
- No exemplo a seguir, o operador + é sobrecarregado através de uma função, para operar sobre dois objetos da classe `Ponto`
- O tipo do retorno da função deve corresponder ao tipo de dados a ser gerado pela operação. O tipo deste retorno pode ser inclusive `void`, caso o operador nada retorne

```
Ponto operator+ (Ponto &p1, Ponto &p2){
  Ponto temp;
  temp.setX(p1.getX()+p2.getX());
  temp.setY(p1.getY()+p2.getY());
  return temp;
}
```

# Sobrecarga de Operadores Unários com Funções

- No caso de um operador **unário**, como o de `!`, o primeiro parâmetro desta função representará o operando que fica à direita do operador
- No exemplo a seguir, o operador `!` é sobrecarregado através de uma função, para zerar os atributos de objetos da classe `Ponto`
- Neste caso o retorno é `void`, mas poderia ser de outro tipo caso fosse necessário

```
void operator! (Ponto &p1){
  p1.setX(0);
  p1.setY(0);
}
```

# Sobrecarga de Operadores com Métodos

- Quando a sobrecarga for feita com um método de uma classe, ou seja, quando ela faz parte da definição da classe (é definida **internamente**), esta função deverá ter:
  - 1 parâmetros caso o operador seja binário
  - nenhum parâmetro caso o operador seja unário

# Sobrecarga de Operadores Binários com Métodos

- No caso de um operador **binário**, como o de multiplicação (*), o parâmetro desta função representará o operando que fica à direita do operador. O operando que fica à esquerda é representado pelos atributos da classe
- No exemplo a seguir, o operador + é sobrecarregado através de um método, para operar sobre dois objetos da classe `Ponto`
- O tipo do retorno da função deve corresponder ao tipo de dados a ser gerado pela operação

```
Ponto Ponto::operator+(const Ponto &p) const {
  return Ponto(x+p.obtemX(), y+p.obtemY());
}
```

# Sobrecarga de Operadores Unários com Métodos

- No caso de um operador **unário**, como o de `!`, o método não terá parâmetros, sendo que os atributos da classe representarão o operando que fica à direita do operador
- No exemplo a seguir, o operador `!` é sobrecarregado através de um método para zerar os atributos de objetos da classe `Ponto`
- Neste caso o retorno é `void`, mas poderia ser de outro tipo caso fosse necessário

```
void Ponto::operator! (){
  this->x = 0;
  this->y = 0;
}
```

# Sobrecarga com `cin` e `cout`

- A sobrecarga do operador << permite definir como um objeto será mostrado em `cout`, que é um objeto da classe `ostream`
- A sobrecarga do operador >> permite especificar como um objeto será lido de `cin`, que é um objeto da classe `istream`
- No exemplo da página a seguir, os dois operadores são sobrecarregados para operarem com objetos da classe `Ponto`

# Sobrecarga com cin e cout

```
ostream &operator<<(ostream &out,const Ponto &p) {
  out << p.str();
  return out;
}

istream &operator>>(istream &in,Ponto &p) {
  cout << "X: ";
  in >> p.x;
  cout << "Y: ";
  in >> p.y;
  return in;
}
```

# Operadores que podem ser sobrecarregados

- É possível sobrecarregar os seguintes operadores:
  - Unários:

    ```
    +   -   *   &   ~   !     ++   --   ->  ->*
    ```

  - Binários:

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

- A maioria dos operadores em C++ podem ser sobrecarregados, exceto os seguintes operadores: `::` `.` `.*` `?:` `sizeof()`
- Os seguintes operadores **não** podem ser sobrecarregados como funções amigas (só como funções membro): `=` `()` `[]` `->` `new` `delete`

# Exemplo: Classe `Ponto`

# Ponto.hpp

```
#ifndef _PONTO_HPP_
#define _PONTO_HPP_

#include <string>

using namespace std;

class Ponto {
private:
  double x,y;
public:
  Ponto(double x=0, double y=0);
  ~Ponto();
  double obtemX() const;
  double obtemY() const;
  void defineX(double x);
  void defineY(double y);
  double distancia(const Ponto &p) const;
  Ponto pontoMedio(const Ponto &p) const;
  string str() const;
  Ponto operator+(const Ponto &p) const;
  Ponto operator-(const Ponto &p) const;
  Ponto operator-() const;
  bool operator==(const Ponto &p) const;
  bool operator!=(const Ponto &p) const;
  void operator=(const Ponto &p);
  friend ostream &operator<<(ostream &out,const Ponto &p);
  friend istream &operator>>(istream &in,Ponto &p);
};

#endif
```

# Ponto.cpp

```
#include <iostream>
#include <sstream>
#include <cmath>
#include "Ponto.hpp"

Ponto::Ponto(double x, double y) {
  this->x = x;
  this->y = y;
#ifdef DEBUG
  cout << "+ Ponto " << str() << " criado..." << endl;
#endif
}

Ponto::~Ponto() {
#ifdef DEBUG
  cout << "- Ponto " << str() << " destruido..." << endl;
#endif
}

double Ponto::obtemX() const {  return x;  }
double Ponto::obtemY() const {  return y;  }
void Ponto::defineX(double x) {  this->x = x;  }
void Ponto::defineY(double y) {  this->y = y;  }

double Ponto::distancia (const Ponto &p) const {
  return sqrt(pow(x-p.obtemX(),2)+pow(y-p.obtemY(),2));
}

Ponto Ponto::pontoMedio (const Ponto &p) const {
  return Ponto( (x+p.obtemX())/2.0, (y+p.obtemY())/2.0 );
}
```

# Ponto.cpp (continuação)

```
string Ponto::str() const {
  stringstream ss;
  ss << "(" << x << "," << y << ")";
  return ss.str();
}

Ponto Ponto::operator+(const Ponto &p) const {  return Ponto(x+p.obtemX(), y+p.obtemY());  }
Ponto Ponto::operator-(const Ponto &p) const {  return Ponto(x-p.obtemX(), y-p.obtemY());  }
Ponto Ponto::operator-() const {  return Ponto(-x, -y);  }
bool Ponto::operator==(const Ponto &p) const {  return p.obtemX() == x && p.obtemY() == y;  }
bool Ponto::operator!=(const Ponto &p) const {  return p.obtemX() != x || p.obtemY() != y;  }

void Ponto::operator=(const Ponto &p) {
  x = p.obtemX();
  y = p.obtemY();
}

ostream &operator<<(ostream &out,const Ponto &p) {
  out << p.str();
  return out;
}

istream &operator>>(istream &in,Ponto &p) {
  in >> p.x;
  in >> p.y;
  return in;
}
```

# PontoMain.cpp

```
#include <iostream>
#include "Ponto.hpp"

int main() {
  Ponto a(4,0), b(0,3);

  cout << "a.str()=" << a.str() << endl;
  cout << "b.str()=" << b.str() << endl;
  cout << "a=" << a << endl << "b=" << b << endl;

  cout << "a.distancia(b)=" << a.distancia(b) << endl;
  cout << "a.pontoMedio(b)=" << a.pontoMedio(b) << endl;
  a.defineX(a.obtemX()+1);
  a.defineY(a.obtemY()+1);
  cout << "a=" << a << endl << "b=" << b << endl;
  cout << "a.obtemX()=" << a.obtemX();
  cout << " a.obtemY()=" << a.obtemY() << endl;
  cout << "a.distancia(b)=" << a.distancia(b) << endl;
  cout << "a.pontoMedio(b)=" << a.pontoMedio(b) << endl;
  cout << endl;

  Ponto c(5,1), d(4,0), e;

  e = c+d;
  cout << "c=" << c << endl << "d=" << d << endl;
  cout << "e=c+d=" << e << endl;
  cout << endl;

  e = c-d;
  cout << "c=" << c << endl << "d=" << d << endl;
  cout << "e=c-d=" << e << endl;
  cout << endl;

  e = -c;
  cout << "c=" << c << endl;
  cout << "e=-c=" << e << endl;
  cout << endl;

  cout << "a=" << a << endl << "c=" << c << endl;
  cout << "d=" << d << endl;
  if ( a == c ) cout << "a == c" << endl;
  else cout << "a != c" << endl;
  if ( a == d ) cout << "a == d" << endl;
  else cout << "a != d" << endl;
  if ( a != c ) cout << "a != c" << endl;
  else cout << "a == c" << endl;
  if ( a != d ) cout << "a != d" << endl;
  else cout << "a == d" << endl;
  cout << endl;

  cout << "a=" << a << endl << "b=" << b << endl;
  a=b;
  cout << "a=b;" << endl;
  cout << "a=" << a << endl << "b=" << b << endl;
  cout << endl;

  cout << "e=" << e << endl;
  cout << "e:" << endl;
  cin >> e;
  cout << "e=" << e << endl;

  return 0;
}
```

# Exercício

# Exercício 1

1. Definir uma classe que represente um círculo e que apresente as seguintes características:
   - Métodos Privados para calcular: área do círculo ($\pi r^2$), circunferência do círculo ($2\pi r$)
   - Construtores: `Circulo()` com centro em (0,0) e raio 1, `Circulo(double raio)` com centro em (0,0), `Circulo (double x, double y)` com raio 1, `Circulo (double x, double y, double raio)`
   - Métodos Públicos para: obter posição X do centro do círculo, obter posição Y do centro do círculo, definir posição X do centro do círculo, definir posição Y do centro do círculo, definir o raio R do círculo, multiplicar o raio R do círculo por determinado fator (de escala), obter o raio do círculo, obter uma cadeia de caracteres com as informações básicas do círculo, obter a área do círculo, obter a circunferência do círculo
   - Método para sobrecarregar o operador << para imprimir os dados de um círculo
   - Método para sobrecarregar o operador >> para ler os dados de um círculo do terminal
   - Métodos para sobrecarregar os operadores > e < para comparar a área de dois círculos

   Criar um programa principal para testar a classe.

# Créditos

# Créditos

- Estas lâminas contêm trechos de materiais disponibilizados pelos professores Rafael Garibotti e Márcio Pinho.

# Soluções

# Circulo.hpp

```
#ifndef _CIRCULO_HPP
#define _CIRCULO_HPP
#include <string>
using namespace std;

class Circulo {
private:
  double x, y, raio;
  double calculaArea() const;
  double calculaCircunferencia() const;
public:
  Circulo();
  Circulo(double r);
  Circulo(double px, double py);
  Circulo(double px, double py, double r);
  void     defineX(double px);
  void     defineY(double py);
  void     defineRaio(double raio);
  void     escala(double fator);
  double   obtemX() const;
  double   obtemY() const;
  double   obtemRaio() const;
  double   obtemArea() const;
  double   obtemCircunferencia() const;
  bool     operator< (const Circulo &c) const;
  bool     operator> (const Circulo &c) const;
  string   str() const;
  friend ostream &operator<< (ostream &out,const Circulo &c);
  friend istream &operator>> (istream &in,Circulo &c);
};
#endif
```

# Circulo.cpp

```
#include <iostream>
#include <sstream>
#include <cmath>
#include "Circulo.hpp"

double Circulo::calculaArea() const {  return M_PI * raio * raio;  }
double Circulo::calculaCircunferencia() const {  return 2.0 * M_PI * raio;  }
Circulo::Circulo() {  x = y = 0.0;  raio = 1.0;  }
Circulo::Circulo(double r) {  x = y = 0.0;  raio = r;  }
Circulo::Circulo(double px, double py) {  x = px;  y = py;  raio = 1.0;  }
Circulo::Circulo(double px, double py, double r) {  x = px;  y = py;  raio = r;  }
void Circulo::defineX(double px) {  x = px;  }
void Circulo::defineY(double py) {  y = py;  }
void Circulo::defineRaio(double r) {  raio = r;  }
void Circulo::escala(double fator) {  raio *= fator;  }
double Circulo::obtemX() const {  return x;  }
double Circulo::obtemY() const {  return y;  }
double Circulo::obtemRaio() const {  return raio;  }
double Circulo::obtemArea() const {  return calculaArea();  }
double Circulo::obtemCircunferencia() const {  return calculaCircunferencia();  }
bool Circulo::operator< (const Circulo &c) const {  return raio < c.obtemRaio();  }
bool Circulo::operator> (const Circulo &c) const {  return raio > c.obtemRaio();  }
ostream &operator<<(ostream &out,const Circulo &c) {  out << c.str();  return out;  }
istream &operator>>(istream &in,Circulo &c) {  in >> c.x >> c.y >> c.raio;  return in;  }

string Circulo::str() const {
  stringstream ss_point;
  ss_point << "x=" << x << "; y=" << y << "; raio=" << raio;
  return ss_point.str();
}
```

# CirculoMain.cpp

```
#include <iostream>
#include "Circulo.hpp"

int main() {
  Circulo c1, c2(2.0), c3(3.0,3.0), c4(4.0,4.0,2.0), c5;
  cout << "c1 = { " << c1 << " }" << endl;
  cout << "c2 = { " << c2 << " }" << endl;
  cout << "c3 = { " << c3 << " }" << endl;
  cout << "c4 = { " << c4 << " }" << endl;
  cout << "c5 = { " << c5 << " }" << endl;
  c1.defineX(1.0);
  c1.defineY(1.0);
  c2.defineX(2.0);
  c2.defineY(2.0);
  c3.defineRaio(3.0);
  c4.escala(2.0);
  cout << "Circulo(x,y,raio)? "; cin >> c5;
  cout << "c1 = { " << c1 << " }" << endl;
  cout << "c2 = { " << c2 << " }" << endl;
  cout << "c3 = { " << c3 << " }" << endl;
  cout << "c4 = { " << c4 << " }" << endl;
  cout << "c5 = { " << c5 << " }" << endl;
  cout << c1.obtemX() << endl;
  cout << c2.obtemY() << endl;
  cout << c3.obtemRaio() << endl;
  cout << c4.obtemArea() << endl;
  cout << c3.obtemCircunferencia() << endl;
  if (c5 > c2) cout << "OK" << endl;
  if (c3 < c4) cout << "OK" << endl;
  return 0;
}
```