

>>> Programação Orientada a Objetos (POO)

... Segredos de Classe

Prof: André de Freitas Smaira

>>> Conversão de Tipos

>>> Conversão de tipos

- * Tipos básicos => padrão
- * Como fazer o mesmo pras **nossas classes**?
- * Duas formas:
 - * tipo existente => novo tipo
 - * novo tipo => tipo existente
- * C++ permite definir os dois tipos de conversão
- * Explicitamente ou implicitamente

```
>>> tipo A => novo tipo
```

* **Construtor** com parâmetro do tipo A

```
Rational(int x) : _numerator(x), _denominator(1) {}
```

```
Rational a = Rational(5);
```

```
// ou
```

```
a = (Rational)5;
```

```
// ou
```

```
a = static_cast<Rational>(5);
```

```
// ou
```

```
a = 5; // Implícito
```

* Se não desejar a última, basta declarar o construtor como **explícito**:

```
class Rational {
```

```
    int _numerator, _denominator;
```

```
public:
```

```
    explicit Rational(int num) : _numerator(num), _denominator(1) {}
```

```
    ...
```

```
};
```

```
>>> novo tipo => tipo A
```

- * Operadores de conversão

- * Palavra `operator`

- * Podemos trocar o `to_double()` que tínhamos criado

```
>>> Conversores
```

```
class Rational {  
    int _numerator, _denominator;  
public:  
    Rational(int num, int den);  
    Rational();  
    Rational(const Rational &r);  
    int numerator();  
    int denominator();  
    Rational plus(const Rational &b);  
    Rational minus(const Rational &b);  
    Rational times(const Rational &b);  
    Rational over(const Rational &b);  
    operator double();  
};
```

>>> Objetos Constantes

>>> Objetos Constantes

- * "Variáveis" podem ser **constantes**
 - * Variáveis podem ser **objetos de uma classe**
 - * Em **constantes**, o valor não pode ser alterado
 - * **Métodos** a princípio poderiam **alterar objetos**
- ```
bool compare(const Rational &a, const Rational &b)
{
 return a.numerator() == b.numerator() &&
 a.denominator() == b.denominator();
}
```
- \* A princípio OK, mas como o **compilador sabe?**
  - \* Não sabe, **esse código não funciona**



## >>> Objetos Constantes

- \* Devemos **indicar para cada método** se ele pode
- \* Se indicados, não podem alterar
- \* **const** após a lista de parâmetros

```
>>> Constantes
```

```
class Rational {
 int _numerator, _denominator;
public:
 Rational(int num, int den);
 Rational();
 Rational(const Rational &r);
 int numerator() const;
 int denominator() const;
 Rational plus(const Rational &b) const;
 Rational minus(const Rational &b) const;
 Rational times(const Rational &b) const;
 Rational over(const Rational &b) const;
 operator double() const;
};
```

```
>>> Composição de Classes
```

## >>> Composição

- \* Programa P00:

  - \* Classes

  - \* Objetos que interagem

- \* Relação entre objetos de diversas formas

- \* Composição é quando um objeto é composto por objetos de outras classes

- \* Composição é caracterizada pela relação de "tem-um" (has-a)

  - \* Um número racional tem-um numerador (`int`) e tem-um denominador (`int`)

  - \* Um círculo tem-um centro (`ponto`) e tem um raio (`float`)

  - \* Uma pessoa tem-um nome (`string`), tem-um endereço (`string`), tem-uma data de nascimento (`data`), etc.

## >>> Inicialização

- \* Ao criar um **objeto composto**, criamos os **objetos componentes**
- \* Todos os **construtores** são chamados
- \* Precisa-se dos parâmetros dos construtores
- \* Usamos **lista de inicialização de membros**

```
>>> Amigos
```

## >>> Funções Amigas

- \* **Membros privados** apenas pelos métodos da classe
- \* Será que é isso mesmo?
- \* É **possível permitir** acesso a funções externas
- \* Quando queremos uma **operação** da classe com **sintaxe de função**
- \* Declarar a função como amiga (**friend**)

## >>> Funções Amigas

```
class Rational {
 int _numerator, _denominator;
public:
 // ...
 friend Rational times(const Rational &a, const Rational &b);
};

Rational times(const Rational &a,
const Rational &b)
{
 return Rational(a._numerator * b._numerator,
a._denominator * b._denominator);
}
// ...

Rational r1, r2, r3;
r1 = times(r2,r3); // ao inves de r1 = r2.times(r3)
```



```
>>> Classes Amigas
```

- \* Se a classe A é amiga da classe B, então **todos os métodos de A podem acessar membros privados de B** (são amigos de B)
- \* **Raramente** é usado

```
>>> this
```

```
>>> this
```

- \* Os métodos se referem aos membros do objeto sobre o qual foram chamados simplesmente citando o nome do membro
- \* Às vezes, é necessária uma **referência explícita** ao objeto.
- \* Ponteiro **this** (aponta para o objeto)

```
class Rational {
 int _numerator, _denominator;
public:
 // etc...
 Rational times(const Rational &r);
 Rational squared();
};
```

```
Rational Rational::squared() {
 return times(*this);
}
```

>>> Membros Estáticos

## >>> Membros Estáticos

- \* Normalmente, os **membros** se relacionam ao **objeto**
- \* Cada objeto tem **os membros** indicados pela classe
- \* O **valor do membro** em cada objeto pode ser **diferente**
- \* Membro **existe** quando o **objeto é criado**
- \* Se um membro é **static**, passa a ser um **membro da classe**
- \* **Todos os objetos** de um programa tem o **mesmo valor** desse membro
- \* **Inicialização** separadamente

## >>> Métodos Estáticos

- \* Métodos também podem ser estáticos
- \* Podem ser chamados sem usar um objeto
- \* Só podem acessar membros estáticos
- \* Podem ser chamados sobre objetos

>>> Exemplo

```
#include<iostream>
```

```
class Conta {
 static int _n;
public:
 Conta() { _n++; }
 static int quantos() { return _n; }
 ~Conta() { _n--; }
};
```

```
int Conta::_n = 0;
```

```
int main() {
 Conta a, *b;
 std::cout << a.quantos() << " ";
 b = new Conta;
 std::cout << b->quantos() << " ";
 std::cout << Conta::quantos() << " ";
 delete b;
 std::cout << Conta::quantos() << " ";
 return 0;
}
```

[~]\$ \_

## >>> Sobrecarga de Operadores

- \* Podemos fazer contas com tipos básicos ( $a = b + c$ )
- \* **Tipos definidos** podem ser **como tipos básicos**
- \* => **Sobrecarga de Operadores**
- \* Declarada usando **operator**



```
>>> operator
```

```
class Rational {
 int _numerator, _denominator;
public:
 Rational(int num, int den);
 Rational();
 Rational(const Rational &r);
 int numerator() const;
 int denominator() const;
 Rational operator+(const Rational &b) const;
 Rational operator-(const Rational &b) const;
 Rational operator*(const Rational &b) const;
 Rational operator/(const Rational &b) const;
 operator double() const;
};
```

```
>>> Problema
```

```
Rational r1(2,3), r2(3,2), r3;
r3 = r1 * 3; // FUNCIONA
```

```
Rational r1(2,3), r2(3,2), r3;
r3 = 2 * r2; // NÃO FUNCIONA
```

```
>>> Operadores Amigos
```

- \* Solução anterior **assimétrica**

- \* Para **simetria**, definimos como **funções amigas**

```
>>> operator
```

```
class Rational {
 int _numerator, _denominator;
public:
 // etc...
 friend Rational operator*(const Rational &a,
 const Rational &b);
};
```

```
Rational operator*(const Rational &a, const Rational &b) {
 return Rational(a._numerator * b._numerator,
 a._denominator * b._denominator);
}
```

```
>>> Problema
```

```
Rational r1(2,3), r2(3,2), r3;
r3 = r1 * 3; // FUNCIONA
```

```
Rational r1(2,3), r2(3,2), r3;
r3 = 2 * r2; // FUNCIONA
```

## >>> Regras

- \* Quase todos operadores funcionam
- \* Exceções: . .\* :: ? : sizeof
- \* Não se pode alterar precedência ou associatividade
- \* Não se pode criar operadores
- \* Não se pode usar argumentos assumidos
- \* Operadores () [] -> e = devem ser métodos
- \* Os operadores ++ e -- são diferentes, pois existem duas variantes (pré e pós)
- \* Para distinguir os dois, um parâmetro tipo int inútil é inserido no pós-incremento (GAMBIARRA)

>>> Exemplo

```
class A {
 int x;
public:
 A(int i = 0) : x(i) {}
 A& operator++() {
 ++x; return *this;
 }
 A operator++(int i) {
 return A(x++);
 }
};
```

```
>>> operator
```

Como uso `cin` e `cout`?

```
class Rational {
 int _numerator, _denominator;
public:
 // etc...
 friend std::ostream& operator<<(std::ostream &out, const Rational &r);
 friend std::istream& operator>>(std::istream &in, Rational &r);
};
```

```
std::ostream& operator<<(std::ostream &out, const Rational &r) {
 out << r._numerator << " / " << r._denominator;
 return out;
}
```

```
std::istream& operator>>(std::istream &in, Rational &r) {
 std::cout << "Numerador: ";
 in >> r._numerator;
 std::cout << "Denominador: ";
 in >> r._denominator;
 return in;
}
```



>>> Atribuição

## >>> Atribuição

- \* **Atribuições** são também **operadores**
- \* Se **não for sobrecarregado**, usa **construtor de cópia**
- \* Cópia de membros não funciona quando objeto lida com recursos (ex: memória).

```
Rational b(1,2);
```

```
Rational a = b; // CONSTRUTOR DE CÓPIA
```

```
a = b; // OPERATOR =
```

>>> Regra Geral

Sempre que uma classe lida com ponteiros para elementos alocados dinamicamente, deve definir:

Operador de atribuição  
Construtor de cópia

## >>> Referências

\* Apostila e Aulas do **Gonzalo Travieso** (IFSC/USP)