

>>> Programação Orientada a Objetos (POO)

... Construtores e Destruidores

Prof: André de Freitas Smaira

```
>>> Já vimos...
```

- * Representação é encapsulada
- * Apenas **métodos** podem acessar **representação**
- * Métodos dependem da **representação correta**:
 - * Dado objeto com **representação correta**
 - * Executam-se operações
 - * Objeto deve continuar com **representação correta**
- * Como ter **representação correta no início?**

```
>>> Aula anterior...
```

```
class Rational {  
    int _numerator, _denominator;  
public:  
    void set(int num, int den);  
    void set(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);  
    double to_double();  
};
```

```
int main() {  
    Rational a, b, c;  
    b.set(1,2);  
    c.set(2,3);  
    a.set(b.times(c));  
    return 0;  
}
```

>>> Construtores

- * E se o cliente **esquecesse de inicializar** (``nunca duvide do usuário'')
- * Como **garantir consistência**?
- * Métodos devem **terminar num estado consistente**
- * A classe deve ter um **construtor** (**inicializa** objetos num estado consistente)
- * Construtor pode ter **parâmetros** (para determinar valor inicial)
- * Pode haver mais do que um construtor (**sobrecarga** de nome)
- * **Construtor sem parâmetros** é denominado **construtor assumido** (padrão)
- * **Construtor assumido** é essencial para, por exemplo, criar **vetores de objetos**

>>> Construtores

- * O **construtor** é um **método**:
 - * Mesmo nome da classe;
 - * **Sem retorno**
 - * Chamado automaticamente quando um objeto é criado:
 - * **Declaração de uma variável** da classe
 - * **Alocação dinâmica** de um objeto

```
>>> Construtores
```

```
class Rational {  
    int _numerator, _denominator;  
public:  
    Rational(int num, int den);  
    void set(int num, int den);  
    void set(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);  
    double to_double();  
};
```

```
Rational::Rational(int num, int den) {  
    _numerator = num;  
    _denominator = den;
```

```
51$ -
```

>>> Construtores

Antes:

```
int main() {  
    Rational a, b, c;  
    b.set(1,2);  
    c.set(2,3);  
    a.set(b.times(c));  
    return 0;  
}
```

Agora:

```
int main() {  
    Rational a, b(1,2), c(2,3);  
    a.set(b.times(c));  
    return 0;  
}
```

Mas...

rational.cpp: In function 'int main()':

rational.cpp:20:14: error: no matching function for call to 'Rational::Rational()'

```
20 |     Rational a, b(1,2), c(2,3);  
   |                   ^
```

```
>>> Construtores
```

Para corrigir isso:

```
class Rational {  
    int _numerator, _denominator;  
public:  
    Rational(int num, int den);  
    Rational() = default;  
    void set(int num, int den);  
    void set(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);  
    double to_double();  
};
```


>>> Construtores

Mas...

```
int main() {  
    Rational a;  
    std::cout << a.numerator() << std::endl;  
    std::cout << a.denominator() << std::endl;  
    std::cout << a.to_double() << std::endl;  
    return 0;  
}
```

Vão aparecer números aleatórios (lixo da memória)

```
>>> Construtores
```

Para corrigir isso:

```
class Rational {  
    int _numerator, _denominator;  
public:  
    Rational(int num=0, int den=1);  
    void set(int num, int den);  
    void set(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);  
    double to_double();  
};
```

```
Rational::Rational(int num, int den) {  
    _numerator = num;  
    _denominator = den;  
}
```

```
>>> Construtores
```

Ou ainda:

```
class Rational {  
    int _numerator, _denominator;  
public:  
    Rational(int num, int den);  
    Rational();  
    void set(int num, int den);  
    void set(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);  
    double to_double();  
};
```

```
Rational::Rational() {  
    _numerator = 0;  
    _denominator = 1;
```

```
} $ _
```

>>> Construtores

Aí sim:

```
int main() {  
    Rational a;  
    std::cout << a.numerator() << std::endl;  
    std::cout << a.denominator() << std::endl;  
    std::cout << a.to_double() << std::endl;  
    return 0;  
}
```

Vai aparecer

0

1

0

>>> Inicialização

- * Construtores são executados para a inicialização
- * Os membros podem ser inicializados pelo construtor
- * Sintaxe especial

```
Rational::Rational(int num, int den) : _numerator(num), _denominator(den) {}  
Rational::Rational() : _numerator(0), _denominator(1) {}
```

>>> Destruidores

- * Alguns objetos não podem ser **simplesmente descartados** (ex: ponteiros, dados sigilosos)
- * Código de limpeza: **Destruidor da classe**
- * **Apenas um destruidor** por classe
- * Método com nome da classe precedido pelo caracter ~
Rational::~~Rational() {}
- * **Sem retorno**
- * **Sem parâmetro**
- * **Chamado automaticamente** quando um objeto é destruído

```
>>> Quando?
```

- * **Construtores**

- * Declaração de variável
- * Objeto temporário
- * **new** para criar objeto

- * **Destruidores**

- * Sai de **escopo**
- * Objeto temporário
- * **delete**

>>> Construtor de cópia

- * Muitas vezes queremos **construir** um objeto com base no **valor de outro** objeto da classe (ex: atribuição, parâmetro por valor)
- * => **Construtor de cópia**
- * Compilador **fornece um assumido**: cópia membro a membro (como em struct)
- * Assumido **não funciona com ponteiros**
- * Definido por **receber uma referência** para objeto da **mesma classe**

```
Rational::Rational(const Rational &r)
: _numerator(r._numerator),
  _denominator(r._denominator) {}
```


>>> Métodos inline

- * Encapsulação => vários métodos pequenos
- * Custo de chamada alto
- * => métodos **inline**
- * Implementação na **declaração da classe**

>>> Construtores

```
class Rational {
    int _numerator, _denominator;
public:
    Rational(int num, int den)
        : _numerator(num), _denominator(den) {}
    Rational() : _numerator(0), _denominator(1) {}
    Rational(const Rational &r)
    { _numerator = r._numerator; _denominator = r._denominator; }
    int numerator() { return _numerator; }
    int denominator() { return _denominator; }
    Rational plus(const Rational &b)
    { return Rational(_numerator*b._denominator + _denominator*b._numerator,
        _denominator*b._denominator); }
    Rational minus(const Rational &b)
    { return Rational(_numerator*b._denominator - _denominator*b._numerator,
        _denominator*b._denominator); }
    Rational times(const Rational &b)
    { return Rational(_numerator*b._numerator,
        _denominator*b._denominator); }
    Rational over(const Rational &b)
    { return Rational(_numerator*b._denominator,
        _denominator*b._numerator); }
    double to_double() { return (double)_numerator/_denominator; }
};
```

>>> Referências

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