# Introdução à Programação com Classes em C++

João Pascoal Faria (versão original)

Ana Paula Rocha (versão 2004/2005)

Luís Paulo Reis (versão 2005/2006)



# Conceito de classe em C++

- Classe em sentido lato: tipo de dados definido pelo utilizador (programador) [Stroustrup]
  - inclui enumerações (**enum**), uniões (**union**), estruturas (**struct**) e classes em sentido estrito (**class**)
  - tipos de dados definidos em bibliotecas *standard* são classes
  - tipos de dados *built-in* ou construídos com apontadores, arrays ou referências (mesmo que nomeados com typedef) não constituem classes
- Classe em sentido estrito: tipo de dados definido com **`class`**
  - é uma generalização do conceito de estrutura em C
  - Para além de dados (*membros-dados*), uma classe pode também conter funções de manipulação desses dados (*membros-funções*), restrições de acesso a ambos (dados e funções) e redefinições de quase todos os operadores de C++ para objectos da classe

## Conceito de classe em C++

- Classe
  - novo tipo de dados que pode ser usado de forma semelhante aos tipos de dados built-in
- Um tipo de dados é uma representação concreta de um conceito
  - Exemplo: o tipo float (built-in) de C++ com as suas operações +, -, *, etc., proporciona um aproximação concreta ao conceito matemático de número real
    - Os detalhes da representação interna de um float (1 byte para a expoente, 3 bytes para a mantissa, etc.) são escondidos



## Conceito de classe em C+

- Novos tipos de dados
  - projectados para representar conceitos da aplicação que não têm representação directa nos tipos *built-in*
- Exemplo: tipo **`Data`**
  - Interessa poder usar os operadores -, ==, += , <<, >>, etc. para subtrair, comparar, incrementar, escrever e ler datas $\Rightarrow$ ***sobrecarga (overloading) de operadores***
  - Interessa poder esconder os detalhes da representação interna de uma data (três inteiros para o dia, mês e ano, ou um único inteiro com o número de dias decorridos desde uma data de referência) $\Rightarrow$ ***encapsulamento***
- Outro Exemplo: tipo IntCell
  - objecto que representa um inteiro

# Classe e objectos

- Conceito de objecto em sentido lato: região de armazenamento capaz de albergar um valor de um dado tipo
  - inclui variáveis, objectos alocados dinamicamente, objectos temporários que são produzidos durante a avaliação de expressões, etc.
- Conceito de objecto em sentido estrito: instância de uma classe
  - em vez de variáveis (do tipo T) fala-se em objectos (da classe ou tipo T)
- Um objecto tem identidade, estado e comportamento
  - A **identidade** é representada pelo **endereço** do objecto (ver apontador `this`)
  - O **estado** é representado pelos valores dos **membros-dados** (também chamados ***atributos*** noutras linguagens)
  - O **comportamento** é descrito pelos **membros-função** (também chamados ***métodos*** noutras linguagens), incluindo funções que definem operadores



# Membros

```
#include <iostream.h>

class Data {
 public:
  int dia;
  int mes;
  int ano;
  void escreve()
  { cout << dia << '/' << mes << '/' << ano; }
  void le()
  { char barra1, barra2;
    cin >> dia >> barra1 >> mes >> barra2 >> ano; }
};

main()
{   Data d = {1, 12, 2000};
    d.escreve();
    d.le();
    d.escreve();
    return 0;
}
```

***membros-dados***

***membros-função***

***membro do objecto a que se refere a chamada da função***

***um membro-função é chamada para um objecto da classe, com operadores de acesso a membros***

# Membros-função

- As funções definidas dentro da classe (como no slide anterior), são implicitamente **inline**
- Funções maiores devem ser apenas *declaradas* dentro da classe, e *definidas* fora da mesma, precedendo o nome da função do nome da classe seguido do operador de resolução de âmbito **::**
  - A função vê os membros da classe da mesma forma, quer seja definida dentro ou fora da classe
  - Permite separar o **interface** (o que interessa aos clientes da classe, normalmente colocado em "*header files*") da **implementação** (normalmente colocada em "*source-code files*")

```
// Interface  (data.h)
class Data {
  public:
    int dia, mes, ano;
    void escreve();
    void le();
};
```

```
// Implementação  (data.cpp)
void Data::escreve()
{ cout << dia << '/' << mes
       << '/' << ano; }

void Data::le()
{ char barra1, barra2;
  cin >> dia >> barra1 >> mes
      >> barra2 >> ano;
}
```



# Controlo de acesso a membros

```
class Data {
  public:
    // funções de escrita (set)
    void setDia(int d)
     { if (d >= 1 && d <= 31) dia = d; }
    void setMes(int m)
     { if (m >= 1 && m <= 12) mes = m; }
    void setAno(int a)
     { if (a >= 1 && a <= 9999) ano = a; }

    // funções de leitura (get)
    int getDia() { return dia; }
    int getMes() { return mes; }
    int getAno() { return ano; }

  private:
    int dia;  // 1 - 31
    int mes;  // 1 - 12
    int ano;  // 1 - 9999
};
...
Data d;    d.dia = 21; /* PROIBIDO! */;    d.setDia(21); /* OK */
```

***Os membros seguintes são visíveis por qualquer função***

***Os membros seguintes só são visíveis pelos membros-função e amigos da classe***

***Facilita manutenção da integridade dos dados!***

***Permite esconder detalhes de implementação que não interessam aos clientes da classe!***

## Diferença entre estruturas e classes

- Numa estrutura (definida com palavra chave `struct`) todos os membros são públicos por omissão
- Numa classe (definida com palavra chave `class`) todos os membros são privados por omissão
- De resto são equivalentes



## Exemplo : `IntCell`

```
#ifndef _IntCell_H_
#define _IntCell_H_

// a class for simulating an integer memory cell
class IntCell
{
 public:
   explicit IntCell (int initialValue = 0);
   int read() const;
   void write(int x);
 private:
  int storedValue;
};

#endif
```

***Interface***

ficheiro "IntCell.h"

# Exemplo : IntCell

```
#include "IntCell.h"
// Construct the IntCell with initialValue
IntCell::IntCell(int initialValue) :
  storedValue(initialValue) {}

// Return the stored value
int IntCell:: read() const
{
    return storedValue;
}

// Store x in IntCell
void IntCell::write(int x)
{
    storedValue = x;
}
```

***Implementação***

ficheiro "IntCell.cpp"



# Exemplo : IntCell

```
#include "IntCell.h"

// Write a Value in IntCell and Read it
int main()
{
    IntCell m;
    m.write(5);
    cout << "Cell contents: " << m.read() << endl;
    return 0;
}
```

***Teste***

# Exemplo : MemoryCell

```
#ifndef MEMORY_CELL_H
#define MEMORY_CELL_H

// A class for simulating a memory cell using Templates
template <class Object>
class MemoryCell
{
   public:
        explicit MemoryCell(const Object &initialValue=Object());
        const Object &read() const;
        void write( const Object &x );
   private:
        Object storedValue;
};
#endif
```



# Exemplo : MemoryCell

```
#include "MemoryCell.h"
// Construct the MemoryCell with initialValue
template <class Object>
MemoryCell<Object>::MemoryCell( const Object &initialValue )
   : storedValue( initialValue ) { }
// Return the stored value.
template <class Object>
const Object &MemoryCell<Object>::read( ) const
{
   return storedValue;
}
// Store x in MemoryCell.
template <class Object>
void MemoryCell<Object>::write( const Object &x )
{
   storedValue = x;
}
```

## Exemplo : `MemoryCell`

```
#include <iostream.h>
#include "MemoryCell.h"
#include "mystring.h"

//Use String and Int memory cells
int main( )
{
   MemoryCell<int> m1;
   MemoryCell<string> m2( "hello" );
   m1.write( 37 );
   m1.write( m1.read()*2 );
   m2.write( m2.read() + " world" );
   cout << m1.read( ) << endl << m2.read( ) << endl;
   return 0;
}
```



## Construtores

- Um membro-função com o mesmo nome da classe é um construtor
  - construtores não podem especificar tipos ou valores de retorno
- O construtor serve normalmente para inicializar os membros-dados
  - podem-se definir construtores com argumentos para receber valores a usar na inicialização
- Construtores podem ser *overloaded*
  - desde que difiram em número ou tipos de argumentos para se poder saber a que versão corresponde cada chamada implícita ou explícita

# Construtores

- O construtor é invocado automaticamente sempre que é criado um objecto da classe
  - para objectos globais, o construtor é chamado no início da execução do programa
  - para objectos locais (automáticos ou estáticos), o construtor é chamado quando a execução passa pelo ponto em que são definidos
- Construtores também podem ser invocados explicitamente



# Exemplo com construtores

```
#include <stdio.h>      //para usar sscanf

class Data {
  public:
    Data(int d, int m, int a=2006); // um construtor
    Data(char *s);                  // outro construtor
    // ...
  private:
    int dia, mes, ano;

};

Data::Data(int d, int m, int a) {
       dia = d; mes =m; ano = a; }

Data::Data(char *s) { // formato dia/mes/ano
       sscanf(s, "%d/%d/%d", &dia, &mes, &ano); }
```

```
void f(Data d);
```

```
Data d1 ("27/3/2006"); // OK - chama Data(char *)
Data d2 (27, 3, 2006); // OK - chama Data(int, int, int)
Data d3 (27, 3);       // OK - chama Data(int, int, int) c/ a=2006
Data d4;               // Erro: não há construtor sem argumentos
Data d5 = {27,3,2006}; // Erro: ilegal na presença de construtores
d1 = Data(1,1,2006);   // OK (chamada explícita de construtor)
f(Data(27,3,2006));    // OK (chamada explícita de construtor)
```

## Objectos e membros constantes (`const`)

- Aplicação do "princípio do privilégio mínimo" (Eng. de Software) aos objectos
- Objecto constante:
  - declarado com prefixo **`const`**
  - especifica que o objecto não pode ser modificado
  - como não pode ser modificado, tem de ser inicializado
  - exemplo: **`const Data nascBeethoven (16, 12, 1770);`**
  - não se pode chamar membro-função não constante sobre objecto constante
- Membro-dado constante:
  - declarado com prefixo **`const`**
  - especifica que não pode ser modificado (tem de ser inicializado)
- Membro-função constante:
  - declarado com sufixo **`const`** (a seguir ao fecho de parêntesis)
  - especifica que a função não modifica o objecto a que se refere a chamada



## Inicializadores de membros

- Quando um membro-dado é constante, um **inicializador de membro** (também utilizável com dados não constantes) tem de ser fornecido para dar ao construtor os valores iniciais do objecto

```
class Pessoa {
  public:
    Pessoa(int, int);           // construtor
    long getIdade() const;      // função constante
  private:
    // ...
    int idade;
    const long BI;              // dado constante
};

Pessoa::Pessoa(int i, long bi) : BI(bi)
    // inicializador de membro ^^^^^^^^ , ...
{ idade = i;}

long Pessoa::getIdade() const
{ return idade; }
```

## Membros estáticos (static)

- Membro-dado estático (declarado com prefixo `static`):
  - variável que faz parte da classe, mas não faz parte dos objectos da classe
  - tem uma única cópia (alocada estaticamente) (mesmo que não exista qualquer objecto da classe), em vez de uma cópia por cada objecto da classe
  - permite guardar um dado pertencente a toda a classe
  - parecido com variável global, mas possui âmbito (*scope*) de classe
  - tem de ser *declarado* dentro da classe (com `static`) e *definido* fora da classe (sem `static`), podendo ser inicializado onde é definido



## Membros estáticos (static)

- Membro-função estático (declarado com prefixo `static`):
  - função que faz parte da classe, mas não se refere a um objecto da classe (identificado por apontador `this` nas funções não estáticas)
  - só pode aceder a membros estáticos da classe
- Referência a membro estático (dado ou função):
  - sem qualquer prefixo, a partir de um membro-função da classe, ou
  - com operadores de acesso a membros a partir de um objecto da classe, ou
  - com *nome-da-classe::nome-do-membro-estático*

# Exemplo com membros estáticos

```
class Factura
{
  public:
    Factura(float v = 0, string no);
    long getNumero() const { return numero; }
    float getValor() const { return valor; }
    static long getUltimoNumero() { return ultimoNumero; }
  private:
    const long numero;
    float valor;
    string nome;
    static long ultimoNumero; // declaração
};

long Factura::ultimoNumero = 0; // definição

Factura::Factura(float v, string no) : numero(++ultimoNumero)
{
    valor = v; nome = no;
}
```



# Exemplo com membros estáticos

```
// Programa de teste
main()
{
  Factura f1(100,"luis"), f2(200,"paulo"), f3("joao");
  cout << "n=" << f1.getNumero() << " v=" << f1.getValor() << endl;
  cout << "n=" << f2.getNumero() << " v=" << f2.getValor() << endl;
  cout << "n=" << f3.getNumero() << " v=" << f3.getValor() << endl;
  cout << "ultimo numero=" << Factura::getUltimoNumero() << endl;
  return 0;
}
```

*Resultados produzidos pelo programa de teste:*

```
n=1 v=100
n=2 v=200
n=3 v=0
ultimo numero=3
```

# Exemplo – Classe Rectangulo

- **Declara uma classe CRectangle e um objecto (i.e. uma variável) desta classe chamado rect**
- **Classe contém quatro membros:**
  - **Dois membros dados de tipo int (x e y) com acesso privado (acesso por defeito)**
  - **Dois membros funções (set_values() e area()) com acesso público**
- **Notar diferença entre Classe e Objecto (semelhante a "int i" em que int é o tipo - classe e i é a variável - objecto)**

```
class CRectangle {
       int x, y;
public:
       void set_values (int,int);
       int area (void);
} rect;
```



# Exemplo – Classe Rectangulo

- **No programa podemos utilizar membros públicos**

```
rect.set_values (3,4);
myarea = rect.area();
```

- **x e y só podem ser referidos por outro membros da classe (são privados)**
- **Definição completa da classe (notar a utilização de ::) :**

```
#include <iostream>
using namespace std;
class CRectangle {
       int x, y;
public:
       void set_values (int,int);
       int area () {return (x*y);}
};
void CRectangle::set_values (int a, int b) {
       if (a>=0) x = a;
       y = b;
}
```

# Exemplo – Classe Rectangulo

- **Exemplo de utilização da Classe com 2 objectos:**

```
int main () {
        CRectangle rect1, rect2;
        rect1.set_values (3,4);
        rect2.set_values (5,10);
        cout << "areas: " << rect1.area() << " e ", rect2.area();
        return 0;
}
```

- **Operador de "Scope" :: utilizado para definir um membro de uma classe, fora da declaração da própria classe**
- **Diferença principal em definir funções dentro e fora da classe é que funções definidas dentro são automaticamente consideradas inline**



# Exemplo – Classe Rectangulo

- **Exemplo com Construtor – removendo set_values()**

```
// example: class constructor
#include <iostream>
using namespace std;

class CRectangle {
    int width, height;
  public:
    CRectangle (int,int);
    int area () {return (width*height);}
};
CRectangle::CRectangle (int a, int b) {
        width = a; height = b;
}
int main () {
  CRectangle rect1 (3,4), rect2 (5,6);
  cout << "rect1 area: " << rect1.area() << endl;
  cout << "rect2 area: " << rect2.area() << endl;
  return 0;
}
```