

FEUP/LEEC/AED/2001-2002

# Introdução à Programação com Classes em C++

João Pascoal Faria

<http://www.fe.up.pt/~jpf>

## 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++ (cont.)

3

- Com classe em C++ é possível criar novos tipos de dados que podem ser usados 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 uma 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
- Novos tipos de dados são projectados para representar conceitos da aplicação que não têm representação directa nos tipos *built-in*
  - Exemplo: em muitas aplicações interessa poder definir um novo tipo **Data**
    - Interessa poder usar os operadores -, ==, +=, <<, >>, etc. para subtrair, comparar, incrementar, escrever e ler datas ? **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) ? **encapsulamento**

## Conceito de objecto em C++

4

- 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-função

5

```
#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.imprime();
```

```
return 0;
```

```
}
```

*membros-dados*

*Nomes de funções mais simples!*

*Acesso mais simples a membros!*

*Maior coesão de dados e funções!*

*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 (cont.)

6

- 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.C)
```

```
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*

*Facilita manutenção da integridade dos dados!*

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

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

# 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
- 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
- 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)

## Exemplo com construtores

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

class Data {
public:
    Data(int d, int m, int a=2000); // 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 ("27/3/2000"); // OK - chama Data(char *)
Data d2 (27, 3, 2000); // OK - chama Data(int, int, int)
Data d3 (27, 3);        // OK - chama Data(int, int, int) c/ a=2000
Data d4;                // Erro: não há construtor sem argumentos
Data d5 = {27,3,2000}; // Erro: ilegal na presença de construtores
d1 = Data(1,1,2000);    // OK (chamada explícita de construtor)
f(Data(27,3,2000));     // 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-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
  - exercício: verificar que funções nos exemplos anteriores devem levar **const**
- Membro-dado constante:
  - declarado com prefixo **const**
  - especifica que não pode ser modificado (tem de ser inicializado)

## 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; }
```

## Composição de classes

13

- Uma classe pode ter como membros objectos doutras classes
- Membros-objecto são inicializados antes dos objectos de que fazem parte
- Os argumentos para os construtores dos membros-objecto são indicados através da sintaxe de inicializadores de membros
- Exemplo

```
class Pessoa {  
    public:  
        Pessoa(char *n, int d, int m, int a); // construtor  
        // ...  
    private:  
        char *nome;  
        Data nascimento;    // membro-objecto  
};  
  
Pessoa::Pessoa(char *n, int d, int m, int a) : nascimento(d, m, a)  
{ /* ... */ }
```



## Membros estáticos (static)

14

- 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
- 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:
  - 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

15

```
#include <iostream.h>

class Factura
{
public:
    Factura(float v = 0);
    long getNumero() const { return numero; }
    float getValor() const { return valor; }
    static long getUltimoNumero() { return ultimoNumero; }
private:
    const long numero;
    float valor;
    static long ultimoNumero; // declaração
};

long Factura::ultimoNumero = 0; // definição

Factura::Factura(float v) : numero(++ultimoNumero)
{
    valor = v;
}
```

## Exemplo com membros estáticos (cont.)

16

```
// Programa de teste
main()
{
    Factura f1(100), f2(200), f3;
    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~~



# O apontador **this**

17

- Cada membro-função (não estático) tem como argumento implícito um apontador para o objecto para o qual a função é chamada, designado **this**
  - usado implicitamente em todas as referências a membros do objecto
  - também pode ser usado explicitamente
- O apontador **this** não pode ser alterado pela função
  - tipo: **nome-da-classe \* const this** (apontador constante)
  - Se a função for constante, o objecto apontado não é alterado pela função
    - tipo: **nome-da-classe const \* const this**  
(apontador constante para objecto constante)
- Exemplo:
  - a seguinte função da classe Data

```
int Data::getDia() const { return dia; }
```

pode escrever-se de forma equivalente:

```
int Data::getDia() const { return this->dia; }
```

# O apontador **this** (cont.)

18

- Caso de uso de **this**: para permitir encadear chamadas de funções sobre um dado objecto

```
class Data {  
    public:  
        Data & setDia(int d);  
        Data & setMes(int m);  
        Data & setAno(int a);  
        //...  
    private:  
        int dia, mes, ano;  
};  
  
// actualiza o dia e devolve referência para mesmo  
// objecto, para poder ser usada de forma encadeada  
Data & Data::setDia(int d)  
{ dia = d; return *this; }  
  
Data d;  
d.setDia(27).setMes(3).setAno(2000);
```

## Criação e destruição de objectos com `new` e `delete`

- Criação de objecto:
 

```
Tipo *tipoPtr; tipoPtr = new Tipo;
```

  - cria um objecto do tamanho apropriado, **chama o construtor** para o objecto (se existir) e devolve um apontador do tipo correcto ou **0** (se não há espaço)
  - podem ser usados inicializadores (são passados ao construtor):
 

```
Data *dataPtr = new Data (27, 3, 2000);
```
- Destruição de objecto:
 

```
delete tipoPtr;
```

  - **invoca o destrutor** para o objecto (se existir) e só depois liberta a memória
- Criação de array de objectos:
 

```
Data *p = new Data[12];
```

  - **invoca o construtor por omissão** (sem argumentos) para cada objecto do array
  - não admite inicializadores
- Destruição de array de objectos:
 

```
delete [] p;
```

  - **invoca o destrutor** para cada objecto do array

## Destrutores

- Um membro-função com o mesmo nome da classe precedido do til (~) é um destrutor
- Destrutores não recebem parâmetros e não retornam valor
- O destrutor da classe é invocado automaticamente sempre que um objecto deixa de existir
  - para objectos globais e objectos locais estáticos, o destrutor é chamado no fim da execução do programa
  - para objectos locais automáticos, o destrutor é chamado quando estes saem do âmbito (*scope*) (quando a execução sai do bloco em que são definidos)
  - para objectos temporários embebidos em expressões, o destrutor é chamado no final da avaliação da expressão completa
- Destrutores servem normalmente para libertar recursos (memória dinâmica, etc.) associados ao objecto

## Exemplo com destrutor

21

```
class Pessoa {
public:
    Pessoa(const char *nm);           // construtor
    ~Pessoa();                       // destrutor
    const char *getNome();           // consulta o nome (devolve
                                    // apontador só para consulta)
    void setNome(const char *);      // muda o nome
private:
    char *nome; // apontador para array de caracteres alocado à medida
};

Pessoa::Pessoa(const char *nm)
{ nome = new char [strlen(nm)+1]; strcpy(nome, nm); }

Pessoa::~~Pessoa()
{ delete [] nome; }

const char *Pessoa::getNome()
{ return nome; }

void Pessoa::setNome(const char *n)
{ delete [] nome; nome = new char [strlen(n)+1]; strcpy(nome, n); }
```

## Construtor de cópia

22

- Um objecto de uma classe pode ser inicializado com uma cópia doutro objecto da mesma classe
  - Exemplo: `Data d1 (27, 3, 2000); Data d2 = d1;`
- O comportamento por omissão é uma cópia membro a membro
- Para especificar outro comportamento: escrever um construtor que tem como argumento uma referência para um objecto da mesma classe (construtor de cópia), o qual é automaticamente usado
  - para tratar atribuição sem ser na inicialização - com sobrecarga de operador =
- Por exemplo, na classe `Pessoa` já apresentada, o comportamento por omissão não é seguro, devido à alocação dinâmica do nome.

### Alternativa:

<pre>class Pessoa { public:     Pessoa(Pessoa &amp; p);     // ... };</pre>	<pre>Pessoa::Pessoa(Pessoa &amp; p) {     nome = new char [strlen(p.nome)+1];     strcpy(nome, p.nome); }</pre>
---	---

## Exemplo com construtor de cópia

23

```
// Programa de teste da classe Factura
main()
{
    Factura f1(100), f2(200), f3, f4 = f2 /*legal apesar de const*/;
    /* f4 = f2; ilegal devido a const */
    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 << "n=" << f4.getNumero() << " v=" << f4.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
n=2 v=200 -> Não gerou um novo número para f4!!
ultimo numero=3
```

## Exemplo com construtor de cópia (cont.)

24

```
// Correção da classe factura
class Factura {
public:
    Factura(Factura & f);
    // ...
};

// copia o valor mas dá um novo número
Factura::Factura(Factura & f) : numero(++ultimoNumero)
{ valor = f.valor; }
```

```
main()
{
    Factura f1(100), f2(200), f3, f4 = f2 /*usa constr. de cópia*/;
    /* f4 = f2; ilegal devido a const e não usa constr. de cópia*/
    // ...
}
```

*Resultados produzidos pelo programa de teste:*

```
n=1 v=100
n=2 v=200
n=3 v=0
n=4 v=200 -> Gerou um novo número para f4!!
ultimo numero=3
```

## Templates de classes

25

- *Template* de classes: "classe genérica" (também chamada classe parametrizada) definida em função de parâmetros a instanciar para se ter uma "classe ordinária"
- Precede-se a definição da "classe genérica" por:  
`template <tipo-de-parâmetro nome-de-parâmetro, ... >`
- **Tipo-de-parâmetro** pode ser o nome de um tipo de dados, ou, mais frequentemente, a palavra-chave **class** para significar que *nome-de-parâmetro* é o nome de um tipo de dados!
- Para se obter uma classe ordinária, têm de se instanciar os parâmetros, numa lista entre <> a seguir ao nome da classe genérica

## Exemplo

26

```
template <class T>
class Pair { // par de valores do tipo T
public:
    Pair(T a, T b);
    void output();
    T first, second;
};

template <class T>
void Pair<T>::Pair(T a, T b)
{first = a; second = b; }

template <class T>
void Pair<T>::output()
{ cout << '(' << _first << ', ' << _second << ')'; }

main()
{
    Pair<double> p1(1.5, 2.5);
    Pair<int> p2(3, 5);
    // ...
    return 0;
}
```

## Conclusões

---

- A classe é a unidade de **ocultação de dados** e de **encapsulamento**
- Classe como **tipo abstracto de dados**: a classe é o mecanismo que suporta **abstracção de dados**, ao permitir que os detalhes de representação sejam escondidos e acedidos exclusivamente através de um conjunto de operações definidas como parte da classe
- A classe proporciona uma unidade de **modularidade**. Em particular, uma classe apenas com membros estáticos proporciona uma facilidade semelhante ao conceito de "módulo": um conjunto nomeado de objectos e funções no seu próprio espaço de nomes.