Escola de Engenharia
Universidade do Minho

Complementos de Programação de Computadores – Aula Teórica 2b
Classes e Abstracção de Dados

Mestrado Integrado em Electrónica Industrial e Computadores

Luís Paulo Reis

lpreis@dsi.uminho.pt

Professor Associado do Departamento de Sistemas de Informação, Escola de Engenharia, Universidade do Minho, Portugal

(Slides Baseados em L.P.Reis et al. 2006 e P.Cortez, 2011)

Escola de Engenharia
Universidade do Minho

Conceito de classe em C++

- **Classe em sentido lato:**
  - **Tipo de dados definido pelo utilizador (programador)**
  - 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`**
  - 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
- **Estrutura (struct) vs Classe (Class)**
  - 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

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 ⇒ ***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***
- **Outro Exemplo: tipo IntCell**
  - Um objecto exemplo 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>

class CData {
 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()
{   CData 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

- **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 (Cdata.h)
class CData {
  public:
    int dia, mes, ano;
    void escreve();
    void le();
};
```

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

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


Controlo de acesso a membros

```
class CData {
  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
};
...
CData d;
d.dia = 21;     /* Erro na compilação! */;
d.setDia(21);   /* OK! Utilização de função membro pública*/
```

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 (friend) da classe

Exemplo : Class CIntCell

```
#ifndef _IntCell_H_
#define _IntCell_H_

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

#endif
```

Interface

ficheiro "CIntCell.h"

Exemplo : Class CIntCell

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

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

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

Implementação

ficheiro "CIntCell.cpp"

Exemplo : Class CIntCell

```
#include "CIntCell.h"

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

Programa Teste

Exemplo : Class CMemoryCell

```
#ifndef MEMORY_CELL_H
#define MEMORY_CELL_H

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

Exemplo : Class CMemoryCell

```
#include “CMemoryCell.h"
// Construct the MemoryCell with initialValue

template <class Object>
CMemoryCell<Object>::CMemoryCell( const Object &initialValue )
   : storedValue( initialValue ) { }

// Return the stored value.
template <class Object>
const Object &CMemoryCell<Object>::read( ) const {
   return storedValue;
}

// Store x in CMemoryCell.
template <class Object>
void CMemoryCell<Object>::write( const Object &x ) {
   storedValue = x;
}
```


Exemplo : Class CMemoryCell

```
#include <iostream>
#include “CMemoryCell.h"
#include "mystring.h"

//Use String and Int CMemory cells
int main( )
{
   CMemoryCell<int> m1;
   CMemoryCell<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
- **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 <cstdio>      //para usar sscanf

class CData {
  public:
    CData(int d, int m, int a=2013); // um construtor
    CData(char *s);                  // outro construtor
    ...
  private:
    int dia, mes, ano;
};

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

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

```
CData d1 ("27/3/2013"); // OK - chama Data(char *)
CData d2 (27, 3, 2013); // OK - chama Data(int, int, int)
CData d3 (27, 3);       // OK - chama Data(int, int, int)c/ a=2006
CData d4;               // Erro: não há construtor sem argumentos
CData d5 = {27,3,2013}; // Erro: ilegal na presença de construtores
d1 = CData(1,1,2013);   // OK (chamada explícita de construtor)
f(CData(27,3,2013));    // 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 CPessoa {
  public:
    CPessoa(int, int);            // construtor
    long getIdade() const;        // função constante
  private:
    // ...
    int idade;
    const long BI;                // dado constante
};

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

long CPessoa::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
- **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 – Classe `CRectangle`

- **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 CRectangle

- 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 ::) :

```
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 CRectangle

- 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 CRectangle

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

```
#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;
}
```


Complementos de Programação de Computadores – Aula Teórica 2b

Classes e Abstracção de Dados

Mestrado Integrado em Electrónica Industrial e Computadores

Luís Paulo Reis

lpreis1970@gmail.com / lpreis@dsi.uminho.pt

Professor Associado do Departamento de Sistemas de Informação, Escola de Engenharia, Universidade do Minho, Portugal

(Slides Baseados em L.P.Reis et al. 2006 e P.Cortez, 2011)