Programação com Classes em C++ (cont.)

João Pascoal Faria (versão original)
Ana Paula Rocha (versão 2004/2005)
Luís Paulo Reis (versão 2005/2006)

Passagem de parâmetros

- Três maneiras de passar parâmetros:
  - *Por valor*, para objectos pequenos que não são alterados pela função
  - *Por referência constante*, para objectos grandes que não são alterados pela função
  - *Por referência,*, para objectos que podem ser alterados pela função

Ex:

double avg(const vector<int> &a, int n, bool & errorF)

Retorno em funções-membro

- Três possibilidades:
  - *Por valor*,
    - quando o objecto a retornar é construído numa variável automática
  - *Por referência (pouco usado),*
    - evita cópia
    - possível quando o objecto a retornar não é local à função
  - *Por referência constante*
    - a referência retornada não pode ser modificada pelo chamador

Retorno por referência constante

- Exemplo: procurar a maior *string* num vector

OK →

```
const string &findMax(const vector<string> &a)
{
  int maxIndex = 0;
  for (int i=1; i<a.size(); i++)
    if (a[maxIndex] < a[i])
        maxIndex = i;
  return a[maxIndex];
}
```

```
const string &findMaxWrong(const vector<string> &a)
{
  string maxValue = a[0];
  for (int i=1; i<a.size(); i++)
    if (maxValue < a[i])
        maxValue = a[i];
  return maxValue;
}
```

← **errado**

Conversão de tipos e qualificador `explicit`

- C++ permite conversão implícita de tipos
  - *ex:* se `d` é do tipo `double` e `i` do tipo `int`, é válido:
    - `d = i;` ou `i = d;`
- Tipos definidos com classes
  - construtores de 1 parâmetro definem uma *conversão implícita de tipo*
  - Exemplo: `IntCell c1 = 54;`
    equivale a chamar o construtor `IntCell(int)` para criar objecto `c1`
- Para impedir conversão implícita
  - usar qualificador `explicit` nos construtores de 1 argumento

Qualificador `explicit`

```
class IntCell
{
  public:
    explicit IntCell (int initialValue=0);
    …
  private:
    int storedValue;
}
```

• Sem `explicit`

```
IntCell c1; c1=20;              // OK
IntCell c2 = IntCell(20);       // OK
IntCell c3, c4(20); c3 = c4;    // OK
```

• Com `explicit`

```
IntCell c1; c1=20;              // errado
IntCell c2 = IntCell(20);       // OK
IntCell c3, c4(20); c3 = c4;    // OK
```

Qualificador `friend`

- Declarar uma função como `friend`
  - função tem acesso aos membros privados da classe
  - Contraria encapsulamento
- Alternativa:
  - usar funções de acesso para obter os valores dos membros privados a usar na função externa
- Declarar uma classe como `friend`
  - todos os membros-função são `friend`

Qualificador `friend`

```
class MyVector
{
  float v[4];
  …
  friend MyVector multiplica(const MyMatrix &, const MyVector &);
};

class MyMatrix
{
  MyVector v[4];
  …
  friend MyVector multiplica(const MyMatrix &, const MyVector &);
};
```

```
MyVector multiplica(const MyMatrix &m, const MyVector &v)
{
  MyVector r;
  for (int i = 0; i<4; i++) {
    r.v[i] = 0;
    for (int j = 0; j<4; j++)
      r.v[i] += m.v[i].v[j] * v.v[j];
  }
  return r;
}
```

função que multiplica uma matriz por um vector

Funções Friend

```
// friend functions
#include <iostream>
using namespace std;

class CRectangle {
    int width, height;
  public:
    void set_values (int, int);
    int area () {return (width * height);}
    friend CRectangle duplicate (CRectangle);
};                      //função duplicate é friend da classe CRectangle

void CRectangle::set_values (int a, int b) {  width = a;  height =
   b; }

CRectangle duplicate (CRectangle rectparam) {
  CRectangle rectres;
  rectres.width = rectparam.width*2;
  rectres.height = rectparam.height*2;
  return (rectres);
}
```


Funções Friend

```
//Programa de Teste

int main () {
  CRectangle rect, rectb;
  rect.set_values (2,3);
  rectb = duplicate (rect);
  cout << rectb.area();
  return 0;
}
```

Classes Friend

```
//Friend class
#include <iostream>
using namespace std;

class CSquare;

class CRectangle {
    int width, height;
  public:
    int area () {return (width * height);}
    void convert (CSquare a);
};

class CSquare {
  private:
    int side;
  public:
    void set_side (int a) { side=a; }
    friend class CRectangle;      //Class CRectangle é freind de CSquare
};

void CRectangle::convert (CSquare a) {
  width = a.side;
  height = a.side;
}
```


Classes Friend

```
//Programa de Teste

int main () {
  CSquare sqr;
  CRectangle rect;
  sqr.set_side(4);
  rect.convert(sqr);
  cout << rect.area();
  return 0;
}
```

Composição de classes

- 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)
{ /* ... */ }
```


O apontador this

- 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.)

- 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 dataPtr;`
  - **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

```
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 p/ 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

- 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:

```
class Pessoa {
  public:
   Pessoa(Pessoa & p);
   // ...
};
```

```
Pessoa::Pessoa(Pessoa & p)
{
  nome = new char [strlen(p.nome)+1];
  strcpy(nome, p.nome);
}
```


Exemplo com construtor de cópia

```
// 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.)

```
// Correcção 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=4
```


Os "3 grandes"

- Classes possuem 3 funções já definidas:
  - Destrutor
    - libertação recursos que foram alocados durante uso objecto
    - invocado quando objecto deixa de ser válido ou é sujeito a `delete`
    - *por omissão*: aplica destrutor a cada membro-dado
  - Construtor de cópia
    - construção de um novo objecto, inicializado com uma cópia de outro do mesmo tipo
    - *por omissão*: aplica construtor de cópia a cada membro-dado; atribuição para tipos primitivos
  - Operador de atribuição (=)
    - copia o estado de um objecto para outro do mesmo tipo, quando = é aplicado a dois objectos
    - *por omissão*: aplica operador = a cada membro-dado

IntCell

- IntCell : membro-dado é um apontador

```
// Data member is a pointer; defaults are no good

class IntCell
{
  public:
    explicit IntCell( int initialValue = 0)
        { storedValue = new int(initialValue); }

    int read() const
    { return *storedValue; }
    void write(int x)
    { *storedValue = x }

  private:
    int *storedValue;
};
```


IntCell: comportamento indesejado

```
// simple function that exposes problems

int f
{
  IntCell a(2);
  IntCell b = a;
  IntCell c;

  c = b;
  a.write(4);
  cout << a.read() << endl << b.read() << endl
       << c.read() << endl;

  return 0;
}
```

Resultado ?

IntCell: comportamento indesejado

- Problema
  - construtor de cópia e operador = *por omissão*
  - o que é copiado é o apontador
    - `a.storedValue`, `b.storedValue`, e `c.storedValue` apontam para o mesmo valor inteiro

```
// the defaults for the big three

IntCell::~IntCell()
{ }
IntCell::IntCell(cont IntCell & rhs) :
storedValue(rhs.storedValue)
{ }
const IntCell & IntCell:: operator= (const IntCell & rhs)
{
  if (this!= & rhs)
    storedValue = rhs.storedValue;
  return *this;
}
```


IntCell: definir os "3 grandes"

```
// Data member is a pointer; big three needs to be written

class IntCell
{
  public:
    explicit IntCell( int initialValue = 0);
    IntCell (const IntCell & rhs);
    ~IntCell();
    const IntCell & operator= (const Intcell & rhs);

    int read() const;
    void write(int x);

  private:
    int *storedValue;
};
```

IntCell: definir os "3 grandes"

```
IntCell::IntCell (int initialValue)
{ storedValue = new int(initialvalue); }

IntCell::IntCell (const IntCell & rhs)
{ storedValue = new int(*rhs.storedValue); }

IntCell::~IntCell()
{ delete storedValue; }

const IntCell &Intcell::operator= (const IntCell &rhs)
{
  if (this!=&rhs)
    *storedValue=*rhs.storedValue;
  return *this;
}

int IntCell::read() const
{ return *storedValue; }

Void IntCell::write(int x)
{  *storedValue = x; }
```


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.