

Programação Orientada por Objetos em C++ (alguns conceitos)

Algoritmos e Estruturas de Dados

2019/2020



Mestrado Integrado em Engenharia Informática e Computação

Passagem de parâmetros

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

Ex:

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



AEDA – 2019/20

Retorno em funções-membro

- Três possibilidades:
 - Por valor,
 - quando o objeto a retornar é construído numa variável automática
 - Por referência,
 - evita cópia
 - possível quando o objeto 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 array

OK



```
const string &findMax(string *a, int n)
{
    int maxIndex = 0;
    for (int i=1; i<n; i++)
        if (a[maxIndex] < a[i])
            maxIndex = i;
    return a[maxIndex];
}
```

```
const string &findMaxWrong(string *a, int n)
{
    string maxValue = a[0];
    for (int i=1; i<n; i++)
        if (maxValue < a[i])
            maxValue = a[i];
    return maxValue;
}
```

← errado



Os “3 grandes”

- Classes possuem 3 funções já definidas:
 - Destruitor
 - libertação recursos que foram alocados durante uso objeto
 - invocado quando objeto deixa de ser válido ou é sujeito a delete
 - por omissão: aplica destrutor a cada membro-dado
 - Construtor de cópia
 - construção de um novo objeto, 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 objeto para outro do mesmo tipo, quando = é aplicado a dois objetos
 - 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: definir os “3 grandes”

- 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

```
// 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; }
```



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 objeto `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
```



Objetos e membros constantes (`const`)

- Objeto constante:
 - declarado com prefixo **`const`**
 - especifica que o objeto 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 objeto constante



Objetos e membros constantes (`const`)

- Membro-dado constante:
 - declarado com prefixo **`const`**
 - especifica que não pode ser modificado (tem de ser inicializado)
 - no construtor usar a seguinte notação, após a lista de argumentos mas antes do corpo:
`: dado_constante(valor_inicializar)`
- Membro-função constante:
 - declarado com sufixo **`const`** (a seguir ao fecho de parêntesis)
 - especifica que a função não modifica o objeto 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 objeto

```
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 objetos da classe
 - tem uma única cópia (alocada estaticamente) (mesmo que não exista qualquer objeto da classe), em vez de uma cópia por cada objeto 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 objeto 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 objeto da classe, ou
 - com *nome-da-classe::nome-do-membro-estático*



Exemplo (membros estáticos)

```
class Fatura
{
public:
    Fatura(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 Fatura::ultimoNumero = 0; // definição

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



Exemplo (membros estáticos)

```
// Programa de teste
main()
{
    Fatura 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=" << Fatura::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 (membros estáticos)

```
// Correção da classe Fatura
class Fatura {
public:
    Fatura(Fatura & f);
    // ...
};
// copia o valor mas dá um novo número
Fatura::Fatura(Fatura & f) : numero(++ultimoNumero)
{ valor = f.valor; }
```

```
main()
{
    Fatura 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
```



Composição de classes

- Uma classe pode ter como membros objetos doutras classes
- Membros-objeto são inicializados antes dos objetos de que fazem parte
- Os argumentos para os construtores dos membros-objeto 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-objeto
};

Pessoa::Pessoa(char *n, int d, int m, int a) : nascimento(d,m,a)
{ /* ... */ }
```



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

```
#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

```
#include <iostream>
using namespace std;

class CSquare;
```

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

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

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



Classes friend

```
//Programa de Teste
```

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

