

Classes II

BCC 221 - Programação Orientada a Objectos(POO)

Guillermo Cámara-Chávez

Departamento de Computação - UFOP
Baseado nos slides do Prof. Marco Antônio Carvalho



Construtores I

nome: @#\$%\$*&%#@
idade: -895555
altura: 856454



Construtores II

```
nome: ""  
idade: 0  
altura: 0
```



Construtores III

- ▶ Quando um **objeto** de uma classe é **criado**, o atributo é **inicializado como vazio**
- ▶ Mas e se quiséssemos que o atributo fosse **inicializado com um valor padrão**?
 - ▶ Podemos **criar um método construtor** para inicializar cada objeto criado.

Construtores IV

- ▶ **Construtor:** Como o encapsulamento de dados é comum, o C++ permite aos objetos serem “inicializados” (iniciados) por si mesmo quando criados.
 - ▶ É um método que possui o **mesmo nome da classe** onde ela está declarado
 - ▶ O construtor **não possui retorno de nenhum tipo**, nem mesmo *void*.
 - ▶ Deve ser declarado como **público**.

Construtores V

- ▶ Se não especificarmos um construtor, o compilador utilizará o construtor padrão
 - ▶ O construtor padrão da classe `string` inicializa com uma cadeia vazia.

Construtores VI

```
#include <iostream>
#include <string>
using namespace std;
class DiarioClasse{
    string nomeDaDisciplina;
public:
    DiarioClasse(string nome){
        setNomeDaDisciplina(nome);
    }
    void setNomeDaDisciplina(string name){
        nomeDaDisciplina = name;
    }
}
```

Construtores VII

```
string getNomeDaDisciplina(){  
    return nomeDaDisciplina;  
}  
void mostraMensagem(){  
    cout << "Seja Bem-vindo ao Diario de Classe de"  
        << getNomeDaDisciplina() << endl;  
} // fim da função  
}; // fim da classe
```


Construtores VIII

```
int main(){
    DiarioClasse meuDiario1("BCC221 - P00");
    DiarioClasse meuDiario2("BCC326 - PDI");

    cout << "Diario de Classe da disciplina: "
         << meuDiario1.getNomeDaDisciplina() << endl
         << "Diario de Classe da disciplina: "
         << meuDiario2.getNomeDaDisciplina() << endl

    return 0;
}
```

Construtores IX

- ▶ Notem que um construtor **pode possuir parâmetros ou não**
 - ▶ Por exemplo, poderíamos **não passar** nenhum **parâmetro e definir um valor padrão** dentro do próprio construtor.
- ▶ Todas **nossas classes** devem **possuir construtores**, para **evitarmos lixo em nossos atributos**

Construtores X

- ▶ É possível criarmos mais de um construtor na mesma classe
 - ▶ Sobrecarga de construtores
 - ▶ Da mesma forma que sobrecarregamos funções
 - ▶ O construtor *default* não possui parâmetros.

Construtores XI

- ▶ A **diferenciação** é feita pelo **número de parâmetros** enviados no momento da criação do objeto
- ▶ Diferentes objetos de uma mesma classe podem ser inicializados por construtores diferentes.
- ▶ **Escolhemos qual construtor é mais adequado** a cada momento.

Construtores XII

```
class Ponto{  
    double x, y;  
    ...  
};
```

- ▶ Métodos especiais de construtores

- ▶ Atribuição:

```
Ponto(){ x = y = 0.0; }
```

- ▶ Ponteiro *this*:

```
Ponto(){ this->x = 0.0; this->y = 0.0; }
```

- ▶ Lista de inicialização:

```
Ponto():x(0.0), y(0.0){}
```

Construtores XIII

- ▶ Podemos ainda ter **construtores com parâmetros padronizados**
 - ▶ O construtor recebe parâmetros para inicializar atributos;
 - ▶ Porém, define parâmetros padronizados, caso não receba nenhum parâmetro.

Construtores XIV

- ▶ Suponha uma classe *Venda*, em que temos os atributos *valor* e *peças*
 - ▶ Ao criar um objeto, o programador pode definir a quantidade de peças e o valor da venda;
 - ▶ Porém, se nada for informado, inicializaremos os atributos com o valor -1, usando o mesmo construtor;
 - ▶ É uma forma de economizar o trabalho de sobrecarregar um construtor.

Construtores XV

```
class Vendas{  
    float valor;  
    int pecas;  
public:  
    Vendas(int p=-1, float v=-1.0){  
        valor = v;  
        pecas = p;  
    }  
    float getValor(){  
        return valor;  
    }  
    int getPecas(){  
        return pecas;  
    }  
}; // fim da classe
```


Construtores XVI

```
int main(){  
    //inicializa um objeto com -1 e outro com 10  
    Vendas a, b(10,10);  
  
    cout << a.getPecas() << endl  
         << a.getValor() << endl  
         << b.getPecas() << endl  
         << b.getValor();  
  
    return 0;  
}
```

- ▶ Como fica o diagrama de classe UML para a classe GradeBook agora que temos um construtor?
- ▶ **Nota:** geralmente construtores são omitidos em diagramas de classes.

Destruutores I

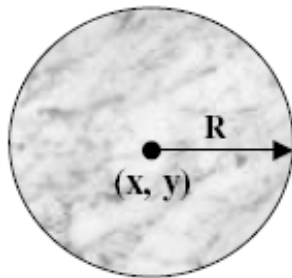
- ▶ De forma análoga aos construtores, que inicializam objetos, temos os destrutores, que finalizam objetos
 - ▶ São chamados automaticamente quando um objeto for destruído, por exemplo, ao terminar o seu bloco de código;
 - ▶ São indicados por um ~ antes do nome do método, que deve ser igual ao da classe.

Destruítores II

- ▶ Destruítores:
 - ▶ Não possuem valor de retorno;
 - ▶ Não podem receber argumentos;
 - ▶ Não podem ser chamados explicitamente pelo programador.
- ▶ **Atenção!**
 - ▶ Se um programa terminar por uma chamada `exit()` ou `abort()`, o destrutor não será chamado

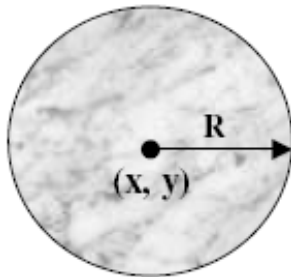
Exemplo

Círculo



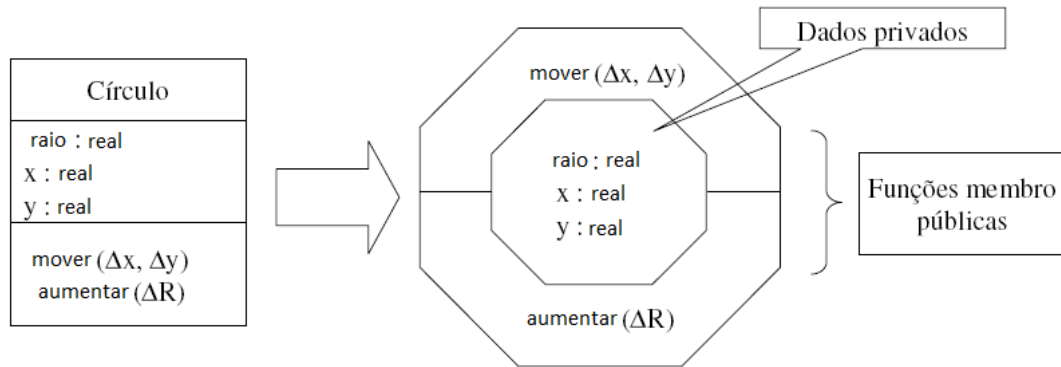
Exemplo

Círculo



Círculo
raio: real x: real y: real
+Circulo() +~ Circulo() +setPonto(x:real, y:real):void +setRaio(raio:real):void +move(dx:real, dy:real):void +aumentar(valor:real):void +mostrar():void

Exemplo I



Geralmente, funções são o único meio de acesso aos atributos da classe!

Exemplo I

```
class Circulo{
    double x, y, raio;
public:
    Circulo(){ setPonto(0.0, 0.0); setRaio(0.0); }
    Circulo(double raio){ setPonto(0.0, 0.0);
        setRaio(raio); }
    Circulo(double x, double y){ setPonto(x, y);
        setRaio(0.0); }
    ~Circulo(){};
    void setPonto(double x, double y);
    void setRaio(double raio);
    void mover(double delta_x, double delta_y);
    void aumentar(double delta_r);
    void mostrar();
};
```

Exemplo II

```
void Circulo::setPonto(double x, double y){
    this->x = x;
    this->y = y;
}
void Circulo::setRaio(double raio){
    this->raio = raio;
}
void Circulo::mover(double delta_x, double delta_y){
    this->x += delta_x;
    this->y += delta_y;
}
void Circulo::aumentar(double delta_r){
    this->raio += delta_r;
}
void Circulo::mostrar(){
    cout << x << "," << y << endl;
    cout << raio << endl;
}
```


Exemplo III

```
int main(){  
    Circulo A, B(10), C(3, 5);  
    A.mostrar();  
    B.mostrar();  
    C.mostrar();  
    cin.get();  
    return 0;  
}
```

Exemplo IV

0, 0
0

0, 0
10

3, 5
0

Exemplo V

- ▶ Vejamos um exemplo em que o construtor de uma classe incrementa um atributo a cada vez que um objeto é criado e o destrutor decrementa o mesmo atributo a cada vez que um objeto é destruído;
- ▶ Como seria possível se cada objeto possui uma cópia diferente de cada atributo?
- ▶ Usamos o modificador *static*, que faz com que haja apenas um atributo compartilhado por todos os objetos

Exemplo VI

```
class Rec{  
    // cria um único item para todos os objetos  
    static int n;  
public:  
    Rec(){ n++; }  
    ~Rec(){ n--; }  
    int getRec(){  
        return n;  
    }  
};
```

Exemplo VII

```
// necessário para que o compilador crie a variável
int Rec::n = 0;

int main(){
    Rec r1, r2, r3;
    cout << r1.getRec() << endl;
    {
        // somente existem dentro deste bloco
        Rec r4, r5, r6;
        cout << r1.getRec() << endl;
    }
    cout << r1.getRec();
    return 0;
}
```

Exemplo VIII

3

6

3

Exemplo IX

- ▶ Construtores e destrutores são especialmente úteis quando os objetos utilizam alocação dinâmica de memória
- ▶ Alocamos a memória no construtor;
- ▶ Desalocamos a memória no destrutor.

Construtores Parametrizados e Vetores de Objetos I

- ▶ No caso de termos um vetor de objetos, recomenda-se não utilizar construtores parametrizados
 - ▶ Ou então utilizar construtores com parâmetros padronizados.
- ▶ Caso seja realmente necessário, no momento da declaração do vetor é necessário inicializá-lo, fazendo a atribuição de objetos anônimos
 - ▶ De forma parecida com a inicialização de vetores de tipos primitivos;
 - ▶ Cada objeto anônimo deve enviar seus parâmetros para o construtor

Construtores Parametrizados e Vetores de Objetos II

```
class Numero{
    int valor;
public:
    Numero(int n){ valor = n;}
    int getNumero(){
        return valor;
    }
};

int main(){
    Numero vet[3] = {Numero(0), Numero(1), Numero(3)};
    for (int i = 0; i < 3; i++)
    {
        cout << vet[i].getNumero() << endl;
    }
    return 0;
}
```

Objetos como Parâmetros de Métodos I

- ▶ Entre os parâmetros que um método pode receber, podemos incluir objetos
 - ▶ Como dito anteriormente, um método só possui acesso aos atributos do objeto que o chamou;
 - ▶ E se precisarmos acessar os atributos de outros objetos?
 - ▶ Podemos passá-los como parâmetros.
 - ▶ Note que para o método acessar os atributos de outros objetos é necessário a utilização do operador .

Objetos como Parâmetros de Métodos II

- ▶ Suponha uma classe Venda, em que temos os atributos valor e peças.
- ▶ Deseja-se totalizar os valores e as peças de uma venda.

Objetos como Parâmetros de Métodos III

```
class Vendas{
    float valor;
    int pecas;
public:
    void setValor(float preco){
        valor = preco;
    }
    void setPecas(int quantidade){
        pecas = quantidade;
    }
    float getValor(){
        return valor;
    }
    int getPecas(){
        return pecas;
    }
    void init(){setValor(0.0); setPecas(0);}
    void totaliza(Vendas v[], int n);
};
```

Objetos como Parâmetros de Métodos IV

```
void Vendas::totaliza(Vendas v[], int n){  
    // evita lixo  
    this->init();  
    for(int i = 0; i < n; i++){  
        valor += v[i].getValor();  
        pecas += v[i].getPecas();  
    }  
}
```

Objetos como Parâmetros de Métodos V

```
int main(){
    Vendas total , v[5];
    v[0].setPecas(1);
    v[1].setPecas(2);
    v[2].setPecas(3);
    v[3].setPecas(4);
    v[4].setPecas(5);
    v[0].setValor(1.0);
    v[1].setValor(2.0);
    v[2].setValor(3.0);
    v[3].setValor(4.0);
    v[4].setValor(5.0);
    total.totaliza(v, 5);
    cout << total.getPecas() << endl
         << total.getValor();
}
```

Métodos de retornam Objetos I

- ▶ Podemos modificar nosso exemplo anterior para retornar um objeto com a totalização dos valores
- ▶ Devemos definir o tipo de retorno como sendo um objeto da classe;
- ▶ Algum objeto deve receber o valor retornado.

Métodos de retornam Objetos II

```
Vendas totaliza(Vendas v[], int n){  
    // evita lixo  
    Vendas tmp;  
    tmp.init();  
    for(int i = 0; i < n; i++){  
        tmp.valor += v[i].getValor();  
        tmp.pecas += v[i].getPecas();  
    }  
    return tmp;  
}
```


Métodos de retornam Objetos III

```
int main(){
    Vendas total , v[5];
    v[0].setPecas(1);
    v[1].setPecas(2);
    v[2].setPecas(3);
    v[3].setPecas(4);
    v[4].setPecas(5);
    v[0].setValor(1.0);
    v[1].setValor(2.0);
    v[2].setValor(3.0);
    v[3].setValor(4.0);
    v[4].setValor(5.0);
    total = v[0].totaliza(v, 5);
    cout << total.getPecas() << endl
         << total.getValor();
}
```

Separando a Interface da Implementação I

- ▶ Uma vantagem de definir classes é que, quando empacotadas apropriadamente, elas podem ser reutilizadas
 - ▶ Como por exemplo, a classe **string**.
- ▶ Nosso exemplo não pode ser reutilizado em outro programa
 - ▶ Já contém um *main*, e todo programa deve possuir apenas um *main*.
- ▶ Para resolver isto, separamos os arquivos

Headers I

- ▶ Arquivos de cabeçalho (ou *header*) possuem extensão `.h`
- ▶ Servem para melhorar a organização do código.
- ▶ Tipicamente contém apenas
 - ▶ definições de tipos
 - ▶ protótipos de funções.

Headers II

- ▶ Normalmente, procura-se agrupar no mesmo arquivo funções e tipos que possuem algo em comum
- ▶ Exemplo: biblioteca de funções matemáticas (`cmath`)
- ▶ Sintaxe:

```
// arquivo.h  
#ifndef ARQUIVO_H  
#define ARQUIVO_H  
  
// Declarações de tipos  
  
#endif
```

Headers III

- ▶ Estes arquivos não são compilados a menos que sejam incluídos por meio da diretiva `#include`

```
#include <iostream>  
#include "arquivo.h"
```

```
int main()  
{  
    return 0;  
}
```

Headers IV

- ▶ Usamos a diretiva `#include "arquivo.h"` com aspas duplas quando se trata de um arquivo `.h` no diretório corrente (do nosso projeto).
- ▶ Usamos os sinais de menor/maior `#include <iostream>` para incluir bibliotecas-padrão do sistema.
- ▶ A diretiva `#include` faz com que o arquivo referido seja incluído inteiramente naquele ponto do código.

Headers V

- ▶ É boa prática de programação “proteger” o código do arquivo *header* usando as diretivas `#ifndef` e `#endif`
- ▶ A lógica é a seguinte:
 - ▶ Se `IMC_H` (nome escolhido ao acaso) não estiver definido, defina `IMC_H`
 - ▶ Se `IMC_H` já estiver definida é porque a estrutura e as funções também já foram definidas e carregadas, não sendo necessário fazê-lo novamente.
- ▶ Isto impede que acidentalmente um programa **.cpp** inclua duas ou mais vezes as definições do arquivo **.h**.

Headers VI

```
// Arquivo: imprime.h
#ifndef IMPRIME_H
#define IMPRIME_H

#include "imc.h"

void imprime(float altura , float peso );
void imprime(DadosBiometricos *db);

#endif
```

- Neste caso, `imprime.h` inclui `imc.h`

Headers VII

- ▶ No `main.cpp` só haveria a necessidade de se incluir `imprime.h` (pois `imc.h` já viria incluído)

```
// Arquivo: main.cpp
// ...
#include "imprime.h"

// etc.
```

Headers VIII

- ▶ A classe fica em um arquivo de cabeçalhos **.h** (*header*)
- ▶ O main fica em um arquivo de código-fonte **.cpp** (*source*);
- ▶ Desta forma, o arquivo **.cpp** deve incluir o arquivo **.h** para reutilizar o código

DiarioClasse.h I

```
class DiarioClasse {
    string nomeDisciplina;
public:
    DiarioClasse(string nome ){
        setNomeDisciplina( nome );
    }
    void setNomeDisciplina( string nome ){
        courseName = name;
    }
    string getNomeDisciplina(){
        return nomeDisciplina;
    }
    void mostraMensagem(){
        cout<<"Bem-vindo ao diario de classe:\n"
            << getNomeDisciplina <<"!"<<endl;
    }
};
```

Main.cpp I

```
#include <iostream>
#include "DiarioClasse.h"
using namespace std;
int main(){
    DiarioClasse meuDiario1("BCC221 - P00");
    DiarioClasse meuDiario2("BCC326 - PDI");

    cout << "Diario de Classe da disciplina: "
         << meuDiario1.getNomeDaDisciplina() << endl
         << "Diario de Classe da disciplina: "
         << meuDiario2.getNomeDaDisciplina() << endl

    return 0;
}
```

Separando a Interface da Implementação I

- ▶ Um problema relacionado a esta divisão de arquivos é que o usuário da classe vai conhecer a implementação
 - ▶ O que não é recomendável.
- ▶ Permite que o usuário escreva programas baseado em detalhes da implementação da classe
 - ▶ Quando na verdade deveria apenas saber quais métodos chamar, sem saber seu funcionamento;
 - ▶ Se a implementação da classe for alterada, o usuário também precisará alterar seu programa.

Separando a Interface da Implementação II

- ▶ Podemos então separar a interface da implementação.
- ▶ Geralmente acompanhando cada arquivo `.h` (em particular os que contém protótipos de funções) existe um arquivo `.cpp`
- ▶ No arquivo `.cpp` são implementadas as funções declaradas no `.h`.
- ▶ Para associar um `.h` com o `.cpp` correspondente, normalmente damos aos dois arquivos o mesmo nome (embora com extensões diferentes).

Interface I

- ▶ A interface de uma classe especifica quais serviços podem ser utilizados e como requisitar estes serviços
 - ▶ E não como os serviços são realizados.
- ▶ A interface pública de uma classe consiste dos métodos públicos
 - ▶ Em nosso exemplo, o construtor, o getter, o setter e o método *mostraMensagem*.

Interface II

- ▶ Separamos então nossa classe em dois arquivos:
 - ▶ A definição da classe e protótipos dos métodos são feitos no arquivo `.h`;
 - ▶ A implementação dos métodos é definida em um arquivo `.cpp` separado;
 - ▶ Por convenção os arquivos possuem o mesmo nome, diferenciados apenas pela extensão.

Interface III

- ▶ O *main* é criado em um terceiro arquivo, também com extensão **.cpp**
 - ▶ *DiarioClasse.h*
 - ▶ *DiarioClasse.cpp*
 - ▶ *Main.cpp*

DiarioClasse.h I

```
#ifndef DIARIOCLASSE_H
#define DIARIOCLASSE_H
#include <string>
class DiarioClasse
{
    string nomeDaDisciplina;
public:
    DiarioClasse( string="" );
    ~DiarioClasse();
    void setNome( string nome);
    std::string getNome();
    void mostraMensagem();
};
#endif
```

- ▶ Na definição da classe, temos apenas a declaração dos atributos e dos protótipos dos métodos
 - ▶ Apenas o cabeçalho dos métodos
 - ▶ Sempre terminados com ;
 - ▶ Note que não é necessário definir um nome para os atributos, apenas o tipo.

DiarioClasse.cpp I

```
#include "DiarioClasse.h"
using namespace std;

DiarioClasse::DiarioClasse(string nomeDaDisciplina) :
    nomeDaDisciplina(nomeDaDisciplina) {}
DiarioClasse::~DiarioClasse(){}
void DiarioClasse::setNome(string nomeDaDisciplina){
    this->nomeDaDisciplina = nomeDaDisciplina;
}
string DiarioClasse::getNome(){
    return nomeDaDisciplina;
}
void DiarioClasse::mostraMensagem(){
    cout << "Bem-vindo : " << getName() << endl;
}
```

- ▶ No arquivo de definição dos métodos deve ser incluído o arquivo **.h** com a definição da classe;
- ▶ Note que após o tipo de cada método, incluímos o nome da classe seguido de **::**
- ▶ Na definição dos métodos é necessário dar nomes aos parâmetros.

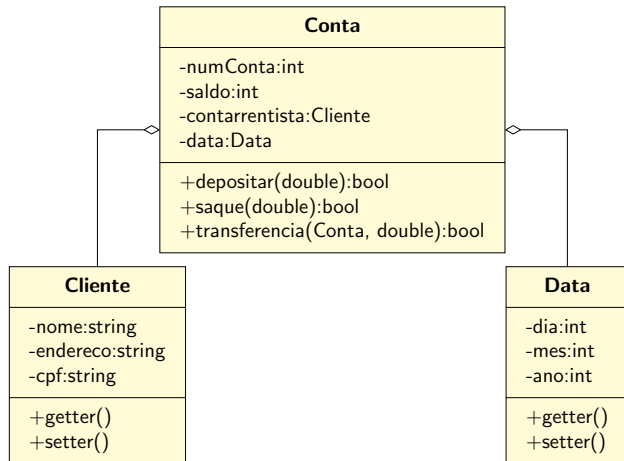
Main.cpp I

```
#include <iostream>
#include <string>
#include "DiarioClasse.h"
using namespace std;
int main(){
    DiarioClasse meuDiario1("BCC221 - P00");
    DiarioClasse meuDiario2("BCC326 - PDI");

    cout << "Diario de Classe da disciplina: "
         << meuDiario1.getNome() << endl
         << "Diario de Classe da disciplina: "
         << meuDiario2.getNome() << endl

    return 0;
}
```

Composição: Objetos como Membros de Classes I



Cliente.h

```
#ifndef CLIENTE_H
#define CLIENTE_H

#include <iostream>
using namespace std;
class Cliente{
    string nome, endereco , cpf;
public:
    Cliente( string="", string="", string="" );
    void setNome( string );
    void setEndereco( string );
    void setCpf( string );
    string getNome() const;
    string getEndereco() const;
    string getCpf() const;
};

#endif
```


Cliente.cpp

```
#include "Cliente.h"
Cliente::Cliente(string nome, string ende, string cpf):
    nome(nome), endereco(ende), cpf(cpf){}
void Cliente::setNome(string nome){
    this->nome = nome;
}
string Cliente::getNome() const{
    return nome;
}
void Cliente::setEndereco(string ende){
    this->endereco = ende;
}
string Cliente::getEndereco() const{
    return endereco;
}
...
```

Data.h

```
#ifndef DATA_H
#define DATA_H

class Data{
    int dia , mes, ano;
public:
    Data(int=0, int=0, int=0);
    void setDia(int);
    int getDia() const;
    void setMes(int);
    int getMes() const;
    void setAno(int);
    int getAno() const;
};

#endif
```

Data.cpp

```
include "Data.h"

Data::Data(int dia , int mes, int ano) :
    dia(dia), mes(mes), ano(ano){}
void Data::setDia(int dia){
    this->dia = dia;
}
int Data::getDia() const{
    return dia;
}
void Data::setMes(int mes){
    this->mes = mes;
}
int Data::getMes() const{
    return mes;
}
...
```

Conta.h

```
#ifndef CONTA_H
#define CONTA_H

class Conta{
    int numConta;
    double saldo;
    Cliente cliente;
    Data data;
public:
    Conta(int , double , Cliente&, Data&);
    Conta();
    bool depositar(double);
    bool saque(double);
    bool transferencia(Conta&, double);
    void mostrarDados() const;
    //getters e setters
};
#endif
```

Conta.cpp

```
#include "Conta.h"

Conta::Conta(int numConta, double saldo,
             Cliente& cliente, Data& data) :
    numConta(numConta), saldo(saldo), cliente(cliente),
    data(data){}

bool Conta::depositar(double valor){
    if (valor > 0){
        saldo += valor;
        return true;
    }
    return false;
}

...
```

main.cpp

```
#include "Conta.h"
using namespace std;
int main(){
    Cliente cli1("Manoel Dias", "Rua A", "111.111.111-11"),
              cli2("Alex Vargas", "Rua B", "222.222.222-22");
    Data data1(1,3,2019), data2(23,5,2019);
    Conta conta1(100, 500.0, cli1, data1);
    Conta conta2(101, 1000.0, cli2, data2);
    conta1.mostrarDados();
    conta2.mostrarDados();
    conta1.transferencia(conta2, 200.0);
    cout << "Apos a transferencia"
    conta1.mostrarDados();
    conta2.mostrarDados();
    return 0;
}
```

FIM