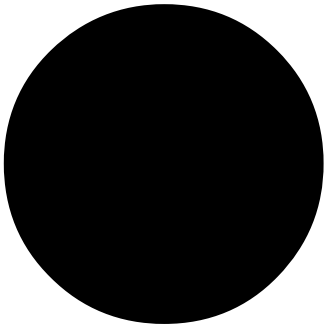
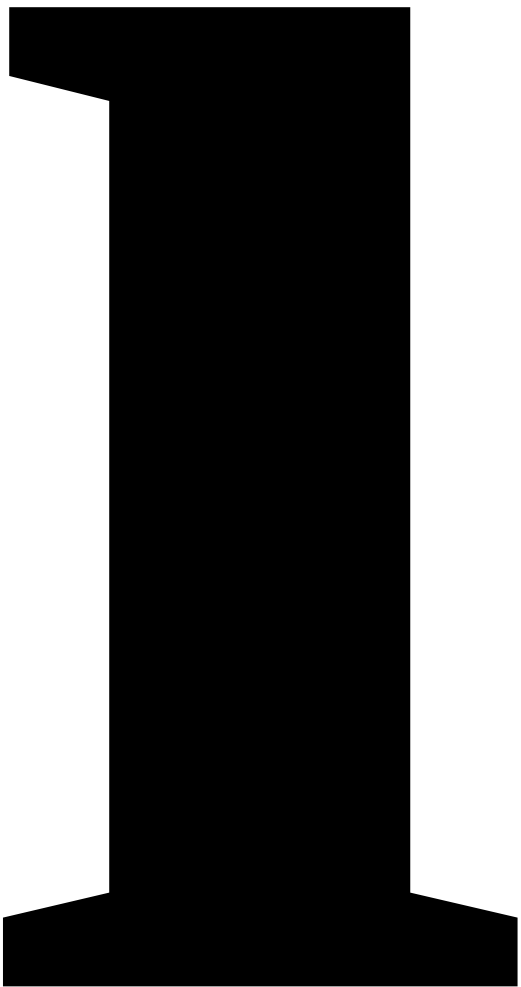
C++

Orientação a Objectos

Índice





Introdução a classes

Teoria

No Curso C++ Básico, aprendemos sobre diversos tipos de variáveis, como `int`, `long` e `char`. O tipo da variável diz muito sobre ela. Por exemplo, se declararmos `x` e `y` como sendo `unsigned int`, sabemos que cada uma delas pode armazenar apenas valores positivos ou zero, dentro de uma faixa bem definida de valores. É esse o significado de dizer que uma variável é `unsigned int`: tentar colocar um valor de outro tipo causa um erro de compilação.

Assim, a declaração do tipo de uma variável indica:

(a) O tamanho da variável na memória

(b) Que tipo de informação a variável pode conter

(c) Que operações podem ser executadas com ela

Mais genericamente, um tipo é uma categoria. No mundo real, temos tipos familiares como carro, casa, pessoa, fruta e forma. Em C++, um programador pode criar qualquer tipo de que precise, e cada novo tipo pode ter funcionalidade similar à dos tipos embutidos na linguagem.

A construção `class` (classe) define as características de um novo tipo de objeto, criado pelo programador.

Exemplo

```
// InClass.cpp

// Ilustra o uso

// de uma classe simples.

#include <iostream.h>

// Define uma classe.

class Cliente

{

public:

        int numCliente;

        float saldo;

}; // Fim de class Cliente.

int main()

{

        // Cria um objeto

        // da classe cliente.

        Cliente objCliente;

        // Atribui valores às

        // variáveis do objeto

        // cliente.

        objCliente.numCliente = 25;
```

```
        objCliente.saldo = 49.95;

        // Exibe valores.

        cout << "\nSaldo do cliente "

                << objCliente.numCliente

                << " = "

                << objCliente.saldo

                << "\n";

} // Fim de main()
```

Exercício

Modifique o programa `InClass.cpp`, de maneira que os valores usados para inicializar o objeto da classe `Cliente` sejam solicitados do usuário.

Private e public

Teoria

Todos os membros de uma classe - dados e métodos - são `private` por default. Isso significa que eles somente podem ser acessados por métodos da própria classe. Os membros `public` podem ser acessados através de qualquer objeto da classe.

Como princípio geral de projeto, devemos tornar os membros de dados de uma classe `private`. Portanto, é preciso criar funções públicas, conhecidas como métodos de acesso, para definir e acessar os valores das variáveis `private`.

Exemplo

```
// PrivPub.cpp

// Ilustra o uso

// de membros private

// e public.

#include <iostream.h>

// Define uma classe.

class Cliente
```

```
{
        // Por default, estes membros
        // são
        //private:
        int numCliente;
        float saldo;
public:
        void defineNumCliente(int num);
        int acessaNumCliente();
        void defineSaldo(float);
        float acessaSaldo();
}; // Fim de class Cliente.
int main()
{
        // Cria um objeto
        // da classe cliente.
        Cliente objCliente;
        // Atribui valores às
        // variáveis do objeto
        // cliente.
        objCliente.defineNumCliente(49);
```

```
        objCliente.defineSaldo(6795.97);
        // Exibe valores.
        cout << "\nSaldo do cliente "
                << objCliente.acessaNumCliente()
                << " = "
                << objCliente.acessaSaldo()
                << "\n";
} // Fim de main()
// Implementação dos métodos.
void Cliente::defineNumCliente(int num)
{
        numCliente = num;
} // Fim de Cliente::defineNumCliente()
int Cliente::acessaNumCliente()
{
        return numCliente;
} // Fim de Cliente::acessaNumCliente()
void Cliente::defineSaldo(float s)
{
        saldo = s;
} // Fim de Cliente::defineSaldo()
```

```
float Cliente::acessaSaldo()

{

        return saldo;

} // Fim de Cliente::acessaSaldo()
```

Exercício

Modifique o programa `PrivPub.cpp` de maneira que os valores de inicialização do objeto da classe `Cliente` sejam solicitados do usuário.

Métodos private

Teoria

Embora o procedimento mais comum seja tornar os membros de dados `private` e os métodos `public`, nada impede que existam métodos `private`. Isso é ditado pelas necessidades do projeto, e não por uma regra fixa. O exemplo abaixo ilustra esse fato.

Exemplo

```
// MetPriv.cpp

// Ilustra o uso

// de métodos private.

#include <iostream.h>

// Define uma classe.

class Cliente

{

        // Por default, estes membros

        // são

        //private:

        int numCliente;

        float saldo;
```

```
        // Estes métodos também
        // são private.
        void defineNumCliente(int num);
        int acessaNumCliente();
        void defineSaldo(float);
        float acessaSaldo();
public:
        void inicializa(int, float);
        void exibe();
}; // Fim de class Cliente.
int main()
{
        // Cria um objeto
        // da classe cliente.
        Cliente objCliente;
        // Atribui valores às
        // variáveis do objeto
        // cliente.
        objCliente.inicializa(52, 99.09);
        // Exibe valores.
        objCliente.exibe();
```

```
} // Fim de main()

// Implementação dos métodos.

void Cliente::defineNumCliente(int num)

{

        numCliente = num;

} // Fim de Cliente::defineNumCliente()

int Cliente::acessaNumCliente()

{

        return numCliente;

} // Fim de Cliente::acessaNumCliente()

void Cliente::defineSaldo(float s)

{

        saldo = s;

} // Fim de Cliente::defineSaldo()

float Cliente::acessaSaldo()

{

        return saldo;

} // Fim de Cliente::acessaSaldo()

void Cliente::inicializa(int num, float sal)

{

        defineNumCliente(num);
```

```
        defineSaldo(sal);

} // Fim de Cliente::inicializa()

void Cliente::exibe()

{

        cout << "\nCliente = "

                << acessaNumCliente()

                << ", Saldo = "

                << acessaSaldo()

                << "\n";

} // Fim de Cliente::exibe()
```

Exercício

Modifique o programa `MetPriv.cpp` de maneira que os valores de inicialização do objeto da classe `Cliente` sejam solicitados do usuário.

Construtores e destrutores

Teoria

Como os membros de dados de uma classe são inicializados? As classes têm uma função membro especial, chamada de construtor. O construtor recebe os parâmetros necessários, mas não pode retornar um valor, nem mesmo `void`. O construtor é um método de classe que tem o mesmo nome que a própria classe.

Quando declaramos um construtor, devemos também declarar um destrutor. Da mesma forma que os construtores criam e inicializam os objetos da classe, os destrutores fazem a limpeza depois que o objeto não é mais necessário, liberando a memória que tiver sido alocada. Um destrutor sempre tem o nome da classe, precedido por um til ~. Os destrutores não recebem argumentos, nem retornam valores.

Se não declararmos um construtor ou um ~destrutor, o compilador cria um automaticamente.

Exemplo

```
// Constr.cpp

// Ilustra o uso

// de métodos construtores

// e destrutores.

#include <iostream.h>
```

```
// Define uma classe.
class Cliente
{
        // Por default, estes membros
        // são
        //private:
        int numCliente;
        float saldo;
        // Estes métodos também
        // são private.
        int acessaNumCliente();
        float acessaSaldo();
public:
        // Construtor default.
        Cliente();
        // Outro construtor.
        Cliente(int, float);
        // Destrutor.
        ~Cliente();
        // Um método public.
        void exibe();
```

```
}; // Fim de class Cliente.

int main()

{

        // Cria um objeto

        // da classe cliente

        // sem definir valores.

        Cliente objCliente1;

        // Cria um objeto

        // da classe cliente

        // inicializado.

        Cliente objCliente2(572, 777.77);

        // Exibe valores.

        cout << "\n*** Valores p/ objcliente1 ***";

        objCliente1.exibe();

        cout << "\n*** Valores p/ objcliente2 ***";

        objCliente2.exibe();

} // Fim de main()

// Implementação dos métodos.

// Construtores.

Cliente::Cliente()

{
```

```
        numCliente = 0;
        saldo = 0.0;
} // Fim de Cliente::Cliente()
Cliente::Cliente(int i, float f)
{
        numCliente = i;
        saldo = f;
} // Fim de Cliente::Cliente(int, float)
// Destrutor.
Cliente::~Cliente()
{
        cout << "\nDestruindo cliente...";
} // Fim de Cliente::~Cliente()
int Cliente::acessaNumCliente()
{
        return numCliente;
} // Fim de Cliente::acessaNumCliente()
float Cliente::acessaSaldo()
{
        return saldo;
} // Fim de Cliente::acessaSaldo()
```

```
void Cliente::exibe()

{

        cout << "\nCliente = "

                << acessaNumCliente()

                << ", Saldo = "

                << acessaSaldo()

                << "\n";

} // Fim de Cliente::exibe()
```

Exercício

Modifique o programa `Constr.cpp`, de maneira que os valores das variáveis membro dos objetos sejam modificados após a criação dos objetos. Faça com que os novos valores sejam exibidos na tela.

Métodos const

Teoria

Quando declaramos um método como sendo `const`, estamos indicando que esse método não alterará o valor de nenhum dos membros da classe. Para declarar um método de classe como sendo `const`, colocamos esta palavra-chave após os parênteses `( )`, mas antes do caractere de ponto e vírgula `;`

Em geral, os métodos de acesso são declarados como `const`.

Quando declaramos uma função como `const`, qualquer tentativa que façamos na implementação dessa função de alterar um valor do objeto será sinalizada pelo compilador como sendo um erro. Isso faz com que o compilador detecte erros durante o processo de desenvolvimento, evitando a introdução de bugs no programa.

Exemplo

```
// ConstMt.cpp

// Ilustra o uso

// de métodos const.

#include <iostream.h>

// Define uma classe.

class Cliente
```

```
{
        // Por default, estes membros
        // são
        //private:
        int numCliente;
        float saldo;
        // Estes métodos
        // são private e const.
        int acessaNumCliente() const;
        float acessaSaldo() const;
public:
        // Construtor default.
        Cliente();
        // Outro construtor.
        Cliente(int, float);
        // Destrutor.
        ~Cliente();
        // Métodos public.
        void exibe() const;
        void defineNumCliente(int);
        void defineSaldo(float);
```

```
}; // Fim de class Cliente.

int main()

{

        // Cria um objeto

        // da classe cliente

        // sem definir valores.

        Cliente objCliente1;

        // Cria um objeto

        // da classe cliente

        // inicializado.

        Cliente objCliente2(572, 777.77);

        // Exibe valores.

        cout << "\n*** Valores p/ objcliente1 ***";

        objCliente1.exibe();

        cout << "\n*** Valores p/ objcliente2 ***";

        objCliente2.exibe();

        // Modifica valores.

        cout << "\nModificando valores...\n";

        objCliente1.defineNumCliente(1000);

        objCliente1.defineSaldo(300.00);

        objCliente2.defineNumCliente(2000);
```

```
        objCliente2.defineSaldo(700.00);
        // Exibe novos valores.
        cout << "\n*** Novos valores p/ objcliente1 ***";
        objCliente1.exibe();
        cout << "\n*** Novos valores p/ objcliente2 ***";
        objCliente2.exibe();
} // Fim de main()
// Implementação dos métodos.
// Construtores.
Cliente::Cliente()
{
        numCliente = 0;
        saldo = 0.0;
} // Fim de Cliente::Cliente()
Cliente::Cliente(int i, float f)
{
        numCliente = i;
        saldo = f;
} // Fim de Cliente::Cliente(int, float)
// Destrutor.
```

```
Cliente::~Cliente()
{
        cout << "\nDestruindo cliente...";
} // Fim de Cliente::~Cliente()
int Cliente::acessaNumCliente() const
{
        return numCliente;
} // Fim de Cliente::acessaNumCliente()
float Cliente::acessaSaldo() const
{
        return saldo;
} // Fim de Cliente::acessaSaldo()
void Cliente::exibe() const
{
        cout << "\nCliente = "
                << acessaNumCliente()
                << ", Saldo = "
                << acessaSaldo()
                << "\n";
} // Fim de Cliente::exibe()
void Cliente::defineNumCliente(int iVal)
```

```
{
        numCliente = iVal;
} // Fim de Cliente::defineNumCliente()
void Cliente::defineSaldo(float fVal)
{
        saldo = fVal;
} // Fim de Cliente::defineSaldo()
```

Exercício

No exemplo `ConstMt.cpp`, experimente (a) declarar como `const` um método que modifica o objeto e (b) tentar modificar um objeto em um método `const`.

Interface e implementação

Teoria

Fazer com que o compilador detecte erros é sempre uma boa idéia. Isso impede que esses erros venham a se manifestar mais tarde, na forma de bugs no programa.

Por isso, convém definir a interface de cada classe em um arquivo de cabeçalho separado da implementação. Quando incluimos esse arquivo de cabeçalho no código do programa, utilizando a diretiva `#include`, permitimos que o compilador faça essa checagem para nós, o que ajuda bastante em nosso trabalho. Na verdade, esse é o procedimento padrão em C++.

O exemplo abaixo ilustra esse fato.

Exemplo

```
// IntImpl.h

// Ilustra a separação

// de interface e

// implementação em diferentes

// arquivos.

#include <iostream.h>

// Define uma classe.
```

```
class Cliente
{
        // Por default, estes membros
        // são
        //private:
        int numCliente;
        float saldo;
        // Estes métodos
        // são private e const.
        int acessaNumCliente() const;
        float acessaSaldo() const;
public:
        // Construtor default.
        Cliente();
        // Outro construtor.
        Cliente(int, float);
        // Destrutor.
        ~Cliente();
        // Métodos public.
        void exibe() const;
```

```
        void defineNumCliente(int);
        void defineSaldo(float);
}; // Fim de class Cliente.
//-------------------------------------------------
//-------------------------------------------------
// IntImpl.cpp
// Ilustra a separação
// de interface e
// implementação em diferentes
// arquivos.
#include "IntImpl.h"
int main()
{
        // Cria um objeto
        // da classe cliente
        // sem definir valores.
        Cliente objCliente1;
        // Cria um objeto
        // da classe cliente
        // inicializado.
        Cliente objCliente2(572, 777.77);
```

```
        // Exibe valores.
        cout << "\n*** Valores p/ objcliente1 ***";
        objCliente1.exibe();
        cout << "\n*** Valores p/ objcliente2 ***";
        objCliente2.exibe();
        // Modifica valores.
        cout << "\nModificando valores...\n";
        objCliente1.defineNumCliente(1000);
        objCliente1.defineSaldo(300.00);
        objCliente2.defineNumCliente(2000);
        objCliente2.defineSaldo(700.00);
        // Exibe novos valores.
        cout << "\n*** Novos valores p/ objcliente1 ***";
        objCliente1.exibe();
        cout << "\n*** Novos valores p/ objcliente2 ***";
        objCliente2.exibe();
} // Fim de main()
// Implementação dos métodos.
// Construtores.
Cliente::Cliente()
{
```

```
        numCliente = 0;
        saldo = 0.0;
} // Fim de Cliente::Cliente()
Cliente::Cliente(int i, float f)
{
        numCliente = i;
        saldo = f;
} // Fim de Cliente::Cliente(int, float)
// Destrutor.
Cliente::~Cliente()
{
        cout << "\nDestruindo cliente...";
} // Fim de Cliente::~Cliente()
int Cliente::acessaNumCliente() const
{
        return numCliente;
} // Fim de Cliente::acessaNumCliente()
float Cliente::acessaSaldo() const
{
        return saldo;
} // Fim de Cliente::acessaSaldo()
```

```
void Cliente::exibe() const

{

        cout << "\nCliente = "

                << acessaNumCliente()

                << ", Saldo = "

                << acessaSaldo()

                << "\n";

} // Fim de Cliente::exibe()

void Cliente::defineNumCliente(int iVal)

{

        numCliente = iVal;

} // Fim de Cliente::defineNumCliente()

void Cliente::defineSaldo(float fVal)

{

        saldo = fVal;

} // Fim de Cliente::defineSaldo()

//-------------------------------------------------
```

Exercício

Modifique o exemplo `IntImpl.cpp/IntImpl.h`, definindo os seguintes métodos como `inline`: `acessaNumCliente()`, `acessaSaldo()`, `defineNumCliente()`, `defineSaldo()`

Objetos como membros

Teoria

Muitas vezes, construimos uma classe complexa declarando classes mais simples e incluindo objetos dessas classes simples na declaração da classe mais complicada. Por exemplo, poderíamos declarar uma classe `roda`, uma classe `motor`, uma classe `transmissão`, e depois combinar objetos dessas classes para criar uma classe `carro`, mais complexa. Chamamos esse tipo de relacionamento tem-um, porque um objeto tem dentro de si outro objeto. Um carro tem um `motor`, quatro `rodas` e uma `transmissão`.

Exemplo

```
//-------------------------------------------------

// ObjMemb.h

// Ilustra uma classe

// que tem objetos

// de outra classe como

// membros.

#include <iostream.h>

class Ponto

{
```

```
        int coordX;
        int coordY;
public:
        void defineX(int vlrX)
        {
                coordX = vlrX;
        } // Fim de defineX()
        void defineY(int vlrY)
        {
                coordY = vlrY;
        } // Fim de defineY()
        int acessaX() const
        {
                return coordX;
        } // Fim de acessaX()
        int acessaY() const
        {
                return coordY;
        } // Fim de acessaY()
}; // Fim de class Ponto.
```

```
// No sistema de coords considerado,
// a origem (0, 0) fica no canto
// superior esquerdo.
class Retangulo
{
        Ponto supEsq;
        Ponto infDir;
public:
        // Construtor.
        Retangulo(int esq, int topo,
                int dir, int base);
        // Destrutor.
        ~Retangulo()
        {
                cout << "Destruindo retangulo...";
        } // Fim de ~Retangulo()
        // Funções de acesso.
        Ponto acessaSupEsq() const
        {
                return supEsq;
        } // Fim de acessaSupEsq() const
```

```
        Ponto acessaInfDir() const
        {
                return infDir;
        } // Fim de acessainfDir() const
        // Funções para definir
        // valores.
        void defineSupEsq(Ponto se)
        {
                supEsq = se;
        } // Fim de defineSupEsq()
        void defineInfDir(Ponto id)
        {
                infDir = id;
        } // Fim de defineInfDir()
        // Função para calcular
        // a área do retângulo.
        int calcArea() const;
        }; // Fim de class Retangulo.
//-------------------------------------------------
//-------------------------------------------------
// ObjMemb.cpp
```

```
// Ilustra uma classe
// que tem objetos
// de outra classe como
// membros.
#include "ObjMemb.h"
// Implementações.
Retangulo::Retangulo(int esq, int topo,
                int dir, int base)
{
        supEsq.defineY(topo);
        supEsq.defineX(esq);
        infDir.defineY(base);
        infDir.defineX(dir);
} // Fim de Retangulo::Retangulo()
int Retangulo::calcArea() const
{
        int xd, xe, ys, yi;
        xd = infDir.acessaX();
        xe = supEsq.acessaX();
        yi = infDir.acessaY();
        ys = supEsq.acessaY();
```

```
        return (xd - xe) * (yi - ys);
} // Fim de Retangulo::calcArea() const
int main()
{
        // Solicita coords para
        // o retangulo.
        int xse, yse, xid, yid;
        cout << "\nDigite x sup. esq.: ";
        cin >> xse;
        cout << "\nDigite y sup. esq.: ";
        cin >> yse;
        cout << "\nDigite x inf. dir.: ";
        cin >> xid;
        cout << "\nDigite y inf. dir.: ";
        cin >> yid;
        // Cria um objeto
        // da classe Retangulo.
        Retangulo ret(xse, yse, xid, yid);
        int areaRet = ret.calcArea();
        cout << "\nArea do retangulo = "
                << areaRet
```

```
                << "\n";

} // Fim de main()

//-------------------------------------------------
```

Exercício

Acrescente ao exemplo `ObjMemb.h/ObjMemb.cpp` uma classe `Coord`, que representa uma coordenada `x` ou `y`. Faça com que a classe `Ponto` utilize objetos da classe `Coord` para definir suas coordenadas.

Classes internas

Teoria

Vimos que C++ permite que uma classe contenha objetos de outra classe. Além disso, C++ permite também que uma classe seja declarada dentro de outra classe. Dizemos que a classe interna está aninhada dentro da classe externa.

Exemplo

```
//----------------------------------------

// ClMemb.cpp

// Ilustra uma classe

// como membro de outra

// classe.

#include <iostream.h>

class Externa

{

        int dadoExt;

public:
```

```
// Construtor.
Externa()
{
        cout << "\nConstruindo obj. Externa...";
        dadoExt = 25;
} // Fim do construtor Externa()
// Destrutor.
~Externa()
{
        cout << "\nDestruindo obj. Externa...";
} // Fim do destrutor ~Externa()
// Uma classe aninhada.
class Interna
{
        int dadoInt;
public:
        // Construtor.
        Interna()
        {
                cout << "\nConstruindo obj Interna...";
                dadoInt = 50;
```

```
                } // Fim do constr. Interna()

                // Destrutor.

                ~Interna()

                {

                        cout << "\nDestruindo obj Interna...";

                } // Fim do destr. ~Interna()

                // Um método public.

                void mostraDadoInt()

                {

                        cout << "\ndadoInt = "

                                << dadoInt;

                } // Fim de mostraDadoInt()

        } objInt; // Um objeto de class Interna.

        // Um método public.

        void mostraTudo()

        {

                objInt.mostraDadoInt();

                cout << "\ndadoExt = "

                        << dadoExt;

        } // Fim de mostraTudo()

}; // Fim de class Externa.
```

```
int main()

{

        // Um objeto de

        // class Externa.

        Externa objExt;

        objExt.mostraTudo();

} // Fim de main()

//-----------------------------------------
```

Exercício

Modifique o exemplo `ClMemb.cpp`, de maneira que o programa solicite do usuários valores a serem atribuídos aos membros de dados `dadoExt` e `dadoInt`.

new e delete com objetos

Teoria

Da mesma forma que podemos criar um ponteiro para um inteiro, podemos criar um ponteiro para qualquer objeto. Se declararmos uma classe chamada `Cliente`, por exemplo, podemos declarar um ponteiro para essa classe e instanciar um objeto `Cliente` no free store, da mesma forma que fazemos na pilha. A sintaxe é a mesma que para um tipo simples:

```
Cliente* pCliente = new Cliente;
```

Isso faz com que o construtor default seja chamado - o construtor que não recebe nenhum parâmetro. O construtor é chamado sempre que um objeto é criado, seja na pilha ou no free store.

Quando chamamos `delete` com um ponteiro para um objeto, o destrutor desse objeto é chamado antes que a memória seja liberada. Isso permite que o próprio objeto faça os preparativos para a limpeza, da mesma forma que acontece com os objetos criados na pilha.

Exemplo

```
//------------------------------------------------

// NewObj.cpp

// Ilustra o uso
```

```
// de new e delete
// com objetos.
#include <iostream.h>
// Define uma classe.
class Cliente
{
        int idCliente;
        float saldo;
public:
        // Construtor.
        Cliente()
        {
                cout << "\nConstruindo obj. Cliente...\n";
                idCliente = 100;
                saldo = 10.0;
        } // Fim de Cliente()
        // Destrutor.
        ~Cliente()
        {
                cout << "\nDestruindo obj. Cliente...\n";
        } // Fim de ~Cliente()
```

```
}; // Fim de class Cliente.

int main()

{

        // Um ponteiro para

        // Cliente.

        Cliente* pCliente;

        // Cria um objeto Cliente com new.

        cout << "\nCriando Cliente com new...\n";

        pCliente = new Cliente;

        // Checa se alocação foi

        // bem sucedida.

        if(pCliente == 0)

        {

                cout << "\nErro alocando memoria...\n";

                return 1;

        } // Fim de if.

        // Cria uma variável local Cliente.

        cout << "\nCriando Cliente local...\n";

        Cliente clienteLocal;

        // Deleta Cliente dinâmico.

        delete pCliente;
```

```
        return 0;

} // Fim de main()

//-----------------------------------------------
```

Exercício

Modifique o exemplo `NewObj.cpp`, de maneira que o construtor da classe `Cliente` aceite dois argumentos, `int` e `float`. Esses valores deverão ser usados para inicializar as variáveis membro de `Cliente`.

Acessando membros via ponteiro

Teoria

Para acessar os membros de dados e métodos de um objeto, usamos o operador ponto `.`
Para acessar os membros de um objeto criado no free store, precisamos de-referenciar o ponteiro e chamar o operador ponto para o objeto apontado. Isso é feito da seguinte forma:

```
(*pCliente).getSaldo();
```

O uso de parênteses é obrigatório, para assegurar que `pCliente` seja de-referenciado, antes do método `getSaldo()` ser acessado.

Como essa sintaxe é muito desajeitada, C++ oferece um operador que permite o acesso indireto: o operador aponta-para `->`, que é criado digitando-se um hífen `-` seguido pelo símbolo maior do que `>`

C++ considera essa combinação como um único símbolo.

Exemplo

```
//------------------------------------------------------

// ViaPont.cpp

// Ilustra o acesso a

// membros de uma classe
```

```
// via ponteiro.
#include <iostream.h>
// Define uma classe.
class Cliente
{
        int idCliente;
        float saldo;
public:
        // Construtor.
        Cliente(int id, float sal)
        {
                cout << "\nConstruindo obj. Cliente...\n";
                idCliente = id;
                saldo = sal;
        } // Fim de Cliente()
        // Destrutor.
        ~Cliente()
        {
                cout << "\nDestruindo obj. Cliente "
                        << idCliente
                        << " ...\n";
```

```
        } // Fim de ~Cliente()

        // Um método para

        // exibir valores do

        // cliente.

        void mostraCliente()

        {

                cout << "\nidCliente = "

                        << idCliente

                        << "\tSaldo = "

                        << saldo;

        } // Fim de mostraCliente()

}; // Fim de class Cliente.

int main()

{

        // Um ponteiro para

        // Cliente.

        Cliente* pCliente;

        // Cria um objeto Cliente com new.

        cout << "\nCriando Cliente com new...";

        pCliente = new Cliente(10, 25.0);

        // Checa se alocação foi
```

```
        // bem sucedida.
        if(pCliente == 0)
        {
                cout << "\nErro alocando memoria...\n";
                return 1;
        } // Fim de if.
        // Cria uma variável local Cliente.
        cout << "\nCriando Cliente local...";
        Cliente clienteLocal(20, 50.0);
        // Exibe os objetos.
        pCliente->mostraCliente();
        // O mesmo que:
        //(*pCliente).mostraCliente();
        clienteLocal.mostraCliente();
        // Deleta Cliente dinâmico.
        delete pCliente;
        return 0;
} // Fim de main()
//-------------------------------------------------
```

Exercício

Modifique o exemplo `ViaPont.cpp`, de maneira que os valores das variáveis membro dos objetos criados sejam alterados após a criação. Para isso, inclua na classe `Cliente` dois métodos para modificar os valores das variáveis: `void defineId(int id)` e `void defineSaldo(float sal)`. Faça com que os novos valores sejam exibidos na tela.

Ponteiros como membros de uma classe

Teoria

Um ou mais membros de uma classe podem ser ponteiros para objetos do free store. A memória pode ser alocada no construtor da classe, ou em um de seus métodos, e pode ser liberada no construtor.

Exemplo

```
//-----------------------------------------------------

// MembPnt.cpp

// Ilustra ponteiros como

// membros de dados

// de uma classe.

#include <iostream.h>

// Define uma classe.

class Cliente

{

        // Estes membros
```

```
        // são ponteiros.
        int* pIdCliente;
        float* pSaldo;
public:
        // Construtor.
        Cliente(int id, float sal);
        // Destrutor.
        ~Cliente();
        // Um método para
        // exibir valores do
        // cliente.
        void mostraCliente()
        {
                cout << "\nidCliente = "
                        << *pIdCliente
                        << "\tSaldo = "
                        << *pSaldo;
        } // Fim de mostraCliente()
}; // Fim de class Cliente.
// Definições dos métodos.
// Construtor.
```

```
Cliente::Cliente(int id, float sal)
{
        cout << "\nConstruindo obj. Cliente...\n";
        // Aloca dinamicamente.
        pIdCliente = new int;
        if(pIdCliente == 0)
        {
                cout << "\nErro construindo objeto!\n";
                return;
        } // Fim de if(pIdCliente...
        *pIdCliente = id;
        pSaldo = new float;
        if(pSaldo == 0)
        {
                cout << "\nErro construindo objeto!\n";
                return;
        } // Fim de if(pSaldo...
        *pSaldo = sal;
} // Fim de Cliente::Cliente()
// Destrutor.
Cliente::~Cliente()
```

```
{
        cout << "\nDestruindo obj. Cliente "
                << *pIdCliente
                << " ...\n";
        if(pIdCliente != 0)
                delete pIdCliente;
        if(pSaldo != 0)
                delete pSaldo;
} // Fim de Cliente::~Cliente()
int main()
{
        // Um ponteiro para
        // Cliente.
        Cliente* pCliente;
        // Cria um objeto Cliente com new.
        cout << "\nCriando Cliente com new...";
        pCliente = new Cliente(10, 25.0);
        // Checa se alocação foi
        // bem sucedida.
        if(pCliente == 0)
        {
```

```
                cout << "\nErro alocando memoria...\n";

                return 1;

        } // Fim de if.

        // Cria uma variável local Cliente.

        cout << "\nCriando Cliente local...";

        Cliente clienteLocal(20, 50.0);

        // Exibe os objetos.

        pCliente->mostraCliente();

        // O mesmo que:

        //(*pCliente).mostraCliente();

        clienteLocal.mostraCliente();

        // Deleta Cliente dinâmico.

        delete pCliente;

        return 0;

} // Fim de main()

//-------------------------------------------------
```

Exercício

Modifique o exemplo `MembPnt.cpp`, de maneira que os valores das variáveis membro dos objetos criados sejam alterados após a criação. Para isso, inclua na classe `Cliente` o seguinte método para modificar os valores das variáveis: `void modificaCliente(int id, float sal)`. Faça com que os novos valores sejam exibidos na tela.

O ponteiro this

Teoria

Toda função membro de classe tem um parâmetro oculto: o ponteiro `this`. O ponteiro `this` aponta para o próprio objeto. Portanto, em cada chamada a uma função membro, ou em cada acesso a um membro de dados, o ponteiro `this` é passado como um parâmetro oculto. Veja o exemplo.

Exemplo

```
//--------------------------------------------------------

// ThisPnt.cpp

// Ilustra o uso do

// ponteiro this.

#include <iostream.h>

// Define uma classe.

class Cliente

{

        int idCliente;

        float saldo;
```

```
public:
        // Construtor.
        Cliente(int id, float sal)
        {
                cout << "\nConstruindo obj. Cliente...\n";
                this->idCliente = id;
                this->saldo = sal;
        } // Fim de Cliente()
        // Destrutor.
        ~Cliente()
        {
                cout << "\nDestruindo obj. Cliente "
                        << this->idCliente
                        << " ...\n";
        } // Fim de ~Cliente()
        // Um método para
        // exibir valores do
        // cliente.
        void mostraCliente() const
        {
```

```
                cout << "\nidCliente = "
                        << this->idCliente
                        << "\tSaldo = "
                        << this->saldo;
        } // Fim de mostraCliente()
        // Métodos para alterar
        // os valores do cliente.
        void defineId(int id)
        {
                this->idCliente = id;
        } // Fim de defineId()
        void defineSaldo(float sal)
        {
                this->saldo = sal;
        } // Fim de defineSaldo()
}; // Fim de class Cliente.
int main()
{
        // Um ponteiro para
        // Cliente.
        Cliente* pCliente;
```

```
// Cria um objeto Cliente com new.
cout << "\nCriando Cliente com new...";
pCliente = new Cliente(10, 25.0);
// Checa se alocação foi
// bem sucedida.
if(pCliente == 0)
{
        cout << "\nErro alocando memoria...\n";
        return 1;
} // Fim de if.
// Cria uma variável local Cliente.
cout << "\nCriando Cliente local...";
Cliente clienteLocal(20, 50.0);
// Exibe os objetos.
pCliente->mostraCliente();
// O mesmo que:
//(*pCliente).mostraCliente();
clienteLocal.mostraCliente();
// Altera valores.
cout << "\nAlterando valores...";
pCliente->defineId(40);
```

```
        pCliente->defineSaldo(400.0);

        clienteLocal.defineId(80);

        clienteLocal.defineSaldo(800.0);

        // Exibe os novos

        // valores dos objetos.

        cout << "\n\nNovos valores...";

        pCliente->mostraCliente();

        clienteLocal.mostraCliente();

        // Deleta Cliente dinâmico.

        delete pCliente;

        return 0;

} // Fim de main()

//-----------------------------------------------------
```

Exercício

Modifique o exemplo `ThisPnt.cpp`, declarando uma classe chamada `Retangulo`. A classe `Retangulo` deve conter uma função membro para calcular a área do retângulo. Utilize esse método para calcular a área de um objeto `Retangulo`, usando o ponteiro `this` em todos os acessos a funções membro e variáveis membro da classe `Retangulo`.

Referências a objetos

Teoria

Podemos ter referêndia a qualquer tipo de variável, inclusive variáveis de tipos definidos pelo usuário. Observe que podemos criar uma referência a um objeto, mas não a uma classe. Não poderíamos escrever:

```
int & refInt = int; // Erro!!!
```

É preciso inicializar `refInt` com uma determinada variável inteira, como por exemplo:

```
int intVar = 100;

int & refInt = intVar;
```

Da mesma forma, não podemos inicializar uma referência com a classe `Cliente`:

```
Cliente & refCliente = Cliente; // Erro!!!
```

Precisamos inicializar `refCliente` com um objeto `Cliente` em particular.

```
Cliente fulano;

Cliente & refCliente = fulano;
```

As referências a objetos são usadas da mesma forma que o próprio objeto. Os membros de dados e os métodos são acessados usando-se o operador ponto `.`

Da mesma forma que no caso dos tipos simples, a referência funciona como um sinônimo para o objeto.

Exemplo

```
//---------------------------------------------------------

// RefClas.cpp

// Ilustra o uso de referências

// a objetos de uma classe.

#include <iostream.h>

// Declara uma classe.

class Cliente

{

        int idCliente;

        int saldo;

public:

        // Construtor.

        Cliente(int id, int sal);

        // Destrutor.

        ~Cliente()

        {

                cout << "\nDestruindo cliente...\n";

        } // Fim de ~Cliente()
```

```
        int acessaId()
        {
                return idCliente;
        } // Fim de acessaId()
        int acessaSaldo()
        {
                return saldo;
        } // Fim de acessaSaldo()
}; // Fim de class Cliente.
// Implementação.
// Construtor.
Cliente::Cliente(int id, int sal)
{
        cout << "\nConstruindo Cliente...\n";
        idCliente = id;
        saldo = sal;
} // Fim de Cliente::Cliente()
int main()
{
        // Um objeto da classe Cliente.
        Cliente umCliente(18, 22);
```

```
        // Uma referência a um objeto

        // da classe Cliente.

        Cliente &refCliente = umCliente;

        // Exibe valores.

        cout << "\n*** Usando objeto ***\n";

        cout << "\nidCliente = "

                << umCliente.acessaId()

                << "\tSaldo = "

                << umCliente.acessaSaldo();

        cout << "\n\n*** Usando referencia ***\n";

        cout << "\nidCliente = "

                << refCliente.acessaId()

                << "\tSaldo = "

                << refCliente.acessaSaldo();

        return 0;

} // Fim de main()

//-------------------------------------------------
```

Exercício

No exemplo `RefClass.cpp`, modifique os valores do objeto `umCliente` usando a referência. Em seguida, exiba os novos valores (a) usando o objeto (b) usando a referência.

Funções membro sobrecarregadas

Teoria

Vimos no Curso C++ Básico que podemos implementar a sobrecarga de funções, escrevendo duas ou mais funções com o mesmo nome, mas com diferentes listas de parâmetros. As funções membros de classes podem também ser sobrecarregadas, de forma muito similar, conforme mostrado no exemplo abaixo.

Exemplo

```
//-----------------------------------------------

// SbrMemb.cpp

// Ilustra sobrecarga

// de funções membro.

#include <iostream.h>

class Retangulo

{

        int altura;

        int largura;

public:
```

```
        // Construtor.
        Retangulo(int alt, int larg);
        // Função sobrecarregada.
        void desenha();
        void desenha(char c);
}; // Fim de class Retangulo.
// Implementação.
// Construtor.
Retangulo::Retangulo(int alt, int larg)
{
        altura = alt;
        largura = larg;
} // Fim de Retangulo::Retangulo()
// Função sobrecarregada.
void Retangulo::desenha()
// Desenha o retângulo preenchendo-o
// com o caractere '*'
{
        for(int i = 0; i < altura; i++)
        {
                for(int j = 0; j < largura; j++)
```

```
                        cout << '*';
                cout << "\n";
        } // Fim de for(int i = 0...
} // Fim de Retangulo::desenha()
void Retangulo::desenha(char c)
// Desenha o retângulo preenchendo-o
// com o caractere c recebido como
// argumento.
{
        for(int i = 0; i < altura; i++)
        {
                for(int j = 0; j < largura; j++)
                        cout << c;
                cout << "\n";
        } // Fim de for(int i = 0...
} // Fim de Retangulo::desenha(char c)
int main()
{
        // Cria um objeto
        // da classe Retangulo.
```

```
        Retangulo ret(8, 12);

        // Desenha usando

        // as duas versões

        // de desenha()

        ret.desenha();

        cout << "\n\n";

        ret.desenha('A');

        return 0;

} // Fim de main()

//-------------------------------------------------
```

Exercício

Modifique o exemplo `SbrMemb.cpp`, acrescentando a seguinte versão sobrecarregada:

```
void desenha(char c, bool preenche);
```

Se o parâmetro `bool preenche` for verdadeiro, o retângulo deverá ser preenchido; caso contrário, deverá ter apenas a borda desenhada.

Funções membro com valores default

Teoria

Da mesma forma que as funções globais podem ter valores default, o mesmo acontece com funções membro de uma classe. O exemplo abaixo ilustra esse fato.

Exemplo

```
//--------------------------------------------------------

// MembDef.cpp

// Ilustra uso de

// valores default em

// funções membro.

#include <iostream.h>

class Retangulo

{

        int altura;

        int largura;

public:
```

```
        // Construtor.
        Retangulo(int alt, int larg);
        // Função com valor
        // default.
        void desenha(char c = '*');
}; // Fim de class Retangulo.
// Implementação.
// Construtor.
Retangulo::Retangulo(int alt, int larg)
{
        altura = alt;
        largura = larg;
} // Fim de Retangulo::Retangulo()
// Função com valor default.
void Retangulo::desenha(char c)
// Desenha o retângulo preenchendo-o
// com o caractere c
{
        for(int i = 0; i < altura; i++)
        {
                for(int j = 0; j < largura; j++)
```

```
                        cout << c;

                cout << "\n";

        } // Fim de for(int i = 0...

} // Fim de Retangulo::desenha()

int main()

{

        // Cria um objeto

        // da classe Retangulo.

        Retangulo ret(8, 12);

        // Desenha usando

        // valor default.

        ret.desenha();

        cout << "\n\n";

        // Desenha especificando

        // caractere.

        ret.desenha('C');

        return 0;

} // Fim de main()

//----------------------------------------
```

Exercício

Modifique o exemplo `MembDef.cpp`, definindo a função membro da classe `Retangulo`

```
void desenha(char c, bool preenche);
```

com valores default para ambos os parâmetros.

Sobrecarregando construtores

Teoria

Conforme sugere o nome, o objetivo de um construtor é construir um objeto. Por exemplo, o construtor da classe `Retangulo` no exemplo abaixo deve construir um retângulo. Antes do construtor ser executado, não existe um retângulo, apenas uma área na memória. Depois que o construtor termina seu trabalho, o objeto `Retangulo` existe, e está pronto para ser usado.

Os construtores, tal como as outras funções membro, podem ser sobrecarregados. A possibilidade de sobrecarregar construtores representa um recurso poderoso e flexível.

Por exemplo, podemos ter uma classe `Retangulo` com dois construtores: o primeiro recebe argumentos para a altura e a largura; o segundo não recebe nenhum argumento, construindo um retângulo com um tamanho padrão. Veja o exemplo.

Exemplo

```
//------------------------------------------------------

// SbrCtr.cpp

// Ilustra sobrecarga

// de construtores.

#include <iostream.h>
```

```
class Retangulo
{
        int altura;
        int largura;
public:
        // Construtores sobrecarregados.
        // Default.
        Retangulo();
        Retangulo(int alt, int larg);
        // Função com valor
        // default.
        void desenha(char c = '*');
}; // Fim de class Retangulo.
// Implementação.
// Construtor default.
Retangulo::Retangulo()
{
        altura = 7;
        largura = 11;
} // Fim de Retangulo::Retangulo()
Retangulo::Retangulo(int alt, int larg)
```

```
{
        altura = alt;
        largura = larg;
} // Fim de Retangulo::Retangulo()
// Função com valor default.
void Retangulo::desenha(char c)
// Desenha o retângulo preenchendo-o
// com o caractere c
{
        for(int i = 0; i < altura; i++)
        {
                for(int j = 0; j < largura; j++)
                        cout << c;
                cout << "\n";
        } // Fim de for(int i = 0...
} // Fim de Retangulo::desenha()
int main()
{
        // Cria um objeto
        // da classe Retangulo
        // usando construtor default.
```

```
        Retangulo ret1;

        // Cria outro objeto

        // especificando as

        // dimensões.

        Retangulo ret2(8, 12);

        // Desenha ret1.

        ret1.desenha('1');

        cout << "\n\n";

        // Desenha ret2.

        ret2.desenha('2');

        return 0;

} // Fim de main()

//-------------------------------------------------
```

Exercício

Acrescente à classe `Retangulo` um construtor sobrecarregado que receba somente um argumento e construa um quadrado com o lado dado por esse argumento.

Inicializando variáveis membro

Teoria

Até agora, temos definido os valores das variáveis membro dentro do corpo do construtor. Porém os construtores são chamados em dois estágios: o estágio de inicialização e o corpo da função.

A maioria das variáveis pode ter seus valores definidos em qualquer dos dois estágios. Porém é mais eficiente, e mais elegante, inicializar as variáveis membro no estágio de inicialização. O exemplo abaixo ilustra como isso é feito.

Exemplo

```
//--------------------------------------------------------

// InicVar.cpp

// Ilustra inicialização

// de variáveis membro.

#include <iostream.h>

class Retangulo

{

        int altura;
```

```
        int largura;
public:
        // Construtores sobrecarregados.
        // Default.
        Retangulo();
        Retangulo(int alt, int larg);
        // Função com valor
        // default.
        void desenha(char c = '*');
}; // Fim de class Retangulo.
// Implementação.
// Construtor default.
Retangulo::Retangulo() :
                altura(7), largura(11)
{
        cout << "\nConstrutor default...\n";
} // Fim de Retangulo::Retangulo()
Retangulo::Retangulo(int alt, int larg) :
                altura(alt), largura(larg)
{
        cout << "\nConstrutor (int, int)...\n";
```

```
} // Fim de Retangulo::Retangulo(int, int)

// Função com valor default.

void Retangulo::desenha(char c)

// Desenha o retângulo preenchendo-o

// com o caractere c

{

        for(int i = 0; i < altura; i++)

        {

                for(int j = 0; j < largura; j++)

                        cout << c;

                cout << "\n";

        } // Fim de for(int i = 0...

} // Fim de Retangulo::desenha()

int main()

{

        // Cria um objeto

        // da classe Retangulo

        // usando construtor default.

        Retangulo ret1;

        // Cria outro objeto

        // especificando as
```

```
        // dimensões.

        Retangulo ret2(8, 12);

        // Desenha ret1.

        ret1.desenha('1');

        cout << "\n\n";

        // Desenha ret2.

        ret2.desenha('2');

        return 0;

} // Fim de main()

//-------------------------------------------------
```

Exercício

Modifique o exemplo `InicVar.cpp`, acrescentando o construtor sobrecarregado para construir um quadrado a partir de um único parâmetro. Utilize a notação de inicialização ilustrada em `InicVar.cpp`.

Construtor de cópia

Teoria

Além de criar automaticamente um construtor e um destrutor default, o compilador fornece também um construtor de cópia default. O construtor de cópia é chamado sempre que fazemos uma cópia do objeto.

Quando um objeto é passado por valor, seja para uma função ou como valor retornado por uma função, uma cópia temporária desse objeto é criada. Quando se trata de um objeto definido pelo usuário, o construtor de cópia da classe é chamado.

Todo construtor de cópia recebe um parâmetro: uma referência para um objeto da mesma classe. Uma sábia providência é fazer essa referência constante, já que o construtor de cópia não precisará alterar o objeto passado.

O construtor de cópia default simplesmente copia cada variável membro do objeto passado como parâmetro para as variáveis membro do novo objeto. Isso é chamado cópia membro a membro, ou cópia rasa. A cópia membro a membro funciona na maioria dos casos, mas pode criar sérios problemas quando uma das variáveis membro é um ponteiro para um objeto do free store.

A cópia membro a membro copia os valores exatos de cada membro do objeto para o novo objeto. Os ponteiros de ambos os objetos ficam assim apontando para a mesma memória. Já a chamada cópia profunda copia os valores alocados no heap para memória recém-alocada.

Se a classe copiada incluir uma variável membro, digamos `pontInt`, que aponta para memória no free store, o construtor de cópia default copiará a variável membro do

objeto recebido para a variável membro correspondente do novo objeto. Os dois objetos apontarão então para a mesma área de memória.

Se qualquer dos dois objetos sair de escopo, teremos um problema. Quando o objeto sai de escopo, o destrutor é chamado e libera a memória alocada. Se o destrutor do objeto original liberar a memória e o novo objeto continuar apontando para essa memória, teremos um ponteiro solto, o que representa um grande risco para o programa.

A solução é o programador criar seu próprio construtor de cópia, e alocar a memória conforme necessário.

Exemplo

```
//-----------------------------------------------------

// CopyCnt.cpp

// Ilustra uso do

// construtor de cópia.

#include <iostream.h>

class Retangulo

{

        int altura;

        int largura;

public:

        // Construtores sobrecarregados.

        // Default.

        Retangulo();

        // Cópia.
```

```
        Retangulo(const Retangulo&);
        Retangulo(int alt, int larg);
        // Função com valor
        // default.
        void desenha(char c = '*');
}; // Fim de class Retangulo.
// Implementação.
// Construtor default.
Retangulo::Retangulo() :
                altura(7), largura(11)
{
        cout << "\nConstrutor default...\n";
} // Fim de Retangulo::Retangulo()
// Construtor de cópia.
Retangulo::Retangulo(const Retangulo& umRet)
{
        cout << "\nConstrutor de copia...\n";
        altura = umRet.altura;
        largura = umRet.largura;
} // Fim de Retangulo::Retangulo(const Retangulo&)
Retangulo::Retangulo(int alt, int larg) :
```

```
                altura(alt), largura(larg)
{
        cout << "\nConstrutor (int, int)...\n";
} // Fim de Retangulo::Retangulo(int, int)
// Função com valor default.
void Retangulo::desenha(char c)
// Desenha o retângulo preenchendo-o
// com o caractere c
{
        for(int i = 0; i < altura; i++)
        {
                for(int j = 0; j < largura; j++)
                        cout << c;
                cout << "\n";
        } // Fim de for(int i = 0...
} // Fim de Retangulo::desenha()
int main()
{
        // Cria um retângulo
        // especificando as
        // duas dimensões.
```

```
        Retangulo retOrig(8, 12);

        // Cria uma cópia usando

        // o construtor de cópia.

        Retangulo retCopia(retOrig);

        // Desenha retOrig.

        cout << "\nRetangulo original\n";

        retOrig.desenha('O');

        // Desenha retCopia.

        cout << "\nRetangulo copia\n";

        retCopia.desenha('C');

        return 0;

} // Fim de main()

//-------------------------------------------------
```

Exercício

Modifique o exemplo `CopyCnt.cpp` utilizando no construtor de cópia a notação que inicializa as variáveis antes do início do corpo do construtor.

Sobrecarregando o operador ++

Teoria

Cada um dos tipos embutidos de C++, como `int`, `float` e `char`, tem diversos operadores que se aplicam a esse tipo, como o operador de adição (`+`) e o operador de multiplicação (`*`). C++ permite que o programador crie também operadores para suas próprias classes, utilizando a sobrecarga de operadores.

Para ilustrar o uso da sobrecarga de operadores, começaremos criando uma nova classe, chamada `Contador`. Um objeto `Contador` poderá ser usado em loops e outras situações nas quais um número deve ser incrementado, decrementado e ter seu valor acessado.

Por enquanto, os objetos de nossa classe `Contador` não podem ser incrementados, decrementados, somados, atribuídos nem manuseados de outras formas. Nos próximos passos, acrescentaremos essa funcionalidade a nossa classe.

Exemplo

```
//--------------------------------------------------------

// SobreO.cpp

// Ilustra sobrecarga

// de operadores.

#include <iostream.h>
```

```
// Declara a classe Contador.
class Contador
{
        unsigned int vlrCont;
public:
        // Construtor.
        Contador();
        // Destrutor.
        ~Contador();
        unsigned int acessaVal() const;
        void defineVal(unsigned int val);
}; // Fim de class Contador.
// Implementação.
// Construtor.
Contador::Contador() :
                vlrCont(0)
{
        cout << "\nConstruindo Contador...\n";
} // Fim de Contador::Contador()
// Destrutor.
Contador::~Contador()
```

```
{
        cout << "\nDestruindo Contador...\n";
} // Fim de Contador::~Contador()
unsigned int Contador::acessaVal() const
{
        return vlrCont;
} // Fim de Contador::acessaVal()
void Contador::defineVal(unsigned int val)
{
        vlrCont = val;
} // Fim de Contador::defineVal()
int main()
{
        // Um objeto contador.
        Contador umCont;
        // Exibe valor.
        cout << "\nValor de contador = "
                << umCont.acessaVal();
        return 0;
} // Fim de main()
//--------------------------------------------------
```

Exercício

Modifique o exemplo `SobreO.cpp`, acrescentando à classe `Contador` o método

```
void mostraVal() const
```

para exibir na tela o valor da variável `vlrCont`.

Teoria

Podemos acrescentar a possibilidade de incrementar um objeto `Contador` de duas maneiras. A primeira é escrever um método `incrementar()`. Esse método é ilustrado no exemplo abaixo.

Exemplo

```
//-------------------------------------------------

// SobreO2.cpp

// Ilustra sobrecarga

// de operadores.

// Acrescenta função

// incrementar()

#include <iostream.h>

// Declara a classe Contador.

class Contador

{

        unsigned int vlrCont;

public:
```

```
        // Construtor.
        Contador();
        // Destrutor.
        ~Contador();
        unsigned int acessaVal() const;
        void defineVal(unsigned int val);
        void mostraVal() const;
        void incrementar();
}; // Fim de class Contador.
// Implementação.
// Construtor.
Contador::Contador() :
vlrCont(0)
{
        cout << "\nConstruindo Contador...\n";
} // Fim de Contador::Contador()
// Destrutor.
Contador::~Contador()
{
        cout << "\nDestruindo Contador...\n";
} // Fim de Contador::~Contador()
```

```
unsigned int Contador::acessaVal() const
{
        return vlrCont;
} // Fim de Contador::acessaVal()
void Contador::defineVal(unsigned int val)
{
        vlrCont = val;
} // Fim de Contador::defineVal()
void Contador::mostraVal() const
{
        cout << "\nValor = "
                << acessaVal();
} // Fim de Contador::mostraVal()
void Contador::incrementar()
{
        ++vlrCont;
} // Fim de Contador::incrementar()
int main()
{
        // Um objeto contador.
        Contador umCont;
```

```
        // Exibe valor.

        umCont.mostraVal();

        // Incrementa.

        umCont.incrementar();

        // Exibe novo valor.

        umCont.mostraVal();

        return 0;

} // Fim de main()

//-------------------------------------------------
```

Exercício

Modifique o exemplo `SobreO2.cpp`, de maneira que a função `incrementar()` exiba uma mensagem na tela ao ser executada.

Teoria

Agora, acrescentaremos a nossa classe `Contador` o operador `++` em prefixo.

Os operadores em prefixo podem ser sobrecarregados declarando-se uma função da forma:

```
tipoRetornado operator op(parametros);
```

Aqui, `op` é o operador a ser sobrecarregado. Assim, o operador `++` pode ser sobrecarregado com a seguinte sintaxe:

```
void operator++();
```

Isso é mostrado no exemplo abaixo.

Exemplo

```
//-----------------------------------------------------

// SobreO4.cpp

// Ilustra sobrecarga

// de operadores.

// Acrescenta operador++

#include <iostream.h>

// Declara a classe Contador.

class Contador

{

        unsigned int vlrCont;

public:

        // Construtor.

        Contador();

        // Destrutor.

        ~Contador();

        unsigned int acessaVal() const;

        void defineVal(unsigned int val);

        void mostraVal() const;

        // Sobrecarrega operador.

        void operator++();

}; // Fim de class Contador.
```

```
// Implementação.

// Construtor.

Contador::Contador() :

vlrCont(0)

{

        cout << "\nConstruindo Contador...\n";

} // Fim de Contador::Contador()

// Destrutor.

Contador::~Contador()

{

        cout << "\nDestruindo Contador...\n";

} // Fim de Contador::~Contador()

unsigned int Contador::acessaVal() const

{

        return vlrCont;

} // Fim de Contador::acessaVal()

void Contador::defineVal(unsigned int val)

{

        vlrCont = val;

} // Fim de Contador::defineVal()
```

```
void Contador::mostraVal() const

{

        cout << "\nValor = "

                << acessaVal();

} // Fim de Contador::mostraVal()

void Contador::operator++()

{

        ++vlrCont;

} // Fim de Contador::operator++()

int main()

{

        // Um objeto contador.

        Contador umCont;

        // Exibe valor.

        umCont.mostraVal();

        // Incrementa.

        ++umCont;

        // Exibe novo valor.

        umCont.mostraVal();

        return 0;

} // Fim de main()
```

```
//------------------------------------------------
```

Exercício

Modifique o exemplo `Sobre04.cpp`, de maneira que o operador++ exiba uma mensagem na tela ao ser executado.

Teoria

O operador ++ em prefixo de nossa classe `Contador` está agora funcionando. Porém, ele tem uma séria limitação. Se quisermos colocar um `Contador` no lado direito de uma atribuição, isso não funcionará. Por exemplo:

```
Contador c = ++i;
```

O objetivo deste código é criar um novo `Contador`, `c`, e depois atribuir a ele o valor de `i`, depois de `i` ter sido incrementado. O construtor de cópia default cuidará da atribuição, mas atualmente, o operador de incremento não retorna um objeto `Contador`. Não podemos atribuir um objeto `void` a um objeto `Contador`.

É claro que o que precisamos é fazer com que o operador ++ retorne um objeto `Contador`.

Exemplo

```
//------------------------------------------------

// SobreO6.cpp

// Ilustra sobrecarga

// de operadores.

// Agora operador++

// retorna um objeto

// temporário.

#include <iostream.h>
```

```
// Declara a classe Contador.
class Contador
{
        unsigned int vlrCont;
public:
        // Construtor.
        Contador();
        // Destrutor.
        ~Contador();
        unsigned int acessaVal() const;
        void defineVal(unsigned int val);
        void mostraVal() const;
        Contador operator++();
}; // Fim de class Contador.
// Implementação.
// Construtor.
Contador::Contador() :
vlrCont(0)
{
        cout << "\nConstruindo Contador...\n";
```

```
} // Fim de Contador::Contador()

// Destrutor.

Contador::~Contador()

{

        cout << "\nDestruindo Contador...\n";

} // Fim de Contador::~Contador()

unsigned int Contador::acessaVal() const

{

        return vlrCont;

} // Fim de Contador::acessaVal()

void Contador::defineVal(unsigned int val)

{

        vlrCont = val;

} // Fim de Contador::defineVal()

void Contador::mostraVal() const

{

        cout << "\nValor = "

                << acessaVal();

} // Fim de Contador::mostraVal()

Contador Contador::operator++()

{
```

```
        cout << "\nIncrementando...";
        ++vlrCont;
        // Cria um objeto temporário.
        Contador temp;
        temp.defineVal(vlrCont);
        return temp;
} // Fim de Contador::operator++()
int main()
{
        // Dois objetos Contador.
        Contador cont1, cont2;
        // Exibe valor.
        cout << "\nObjeto cont1";
        cont1.mostraVal();
        // Incrementa e atribui.
        cont2 = ++cont1;
        // Exibe novo objeto.
        cout << "\nObjeto cont2";
        cont2.mostraVal();
        return 0;
} // Fim de main()
```

```
//---------------------------------------
```

Exercício

Acrescente ao exemplo `SobreO6.cpp` um construtor de cópia que exiba uma mensagem na tela ao ser chamado.

Teoria

Na verdade, não há necessidade do objeto `Contador` temporário criado no exemplo anterior. Se `Contador` tiver um construtor que recebe um valor, podemos simplesmente retornar o resultado desse construtor, como sendo o valor retornado pelo operador ++. O exemplo abaixo mostra como fazer isso.

Exemplo

```
//-----------------------------------------------------

// SobreO8.cpp

// Ilustra sobrecarga

// de operadores.

// Agora operador++

// retorna um objeto

// temporário sem nome.

#include <iostream.h>

// Declara a classe Contador.

class Contador

{

        unsigned int vlrCont;
```

```
public:
        // Construtor.
        Contador();
        // Construtor de cópia.
        Contador(Contador&);
        // Construtor com
        // inicialização.
        Contador(int vlr);
        // Destrutor.
        ~Contador();
        unsigned int acessaVal() const;
        void defineVal(unsigned int val);
        void mostraVal() const;
        Contador operator++();
}; // Fim de class Contador.
// Implementação.
// Construtor.
Contador::Contador() :
vlrCont(0)
{
        cout << "\nConstruindo Contador...\n";
```

```
} // Fim de Contador::Contador()

// Construtor de cópia.

Contador::Contador(Contador& umCont) :

                vlrCont(umCont.vlrCont)

{

        cout << "\nCopiando Contador...\n";

} // Fim de Contador::Contador(Contador&)

// Construtor com

// inicialização.

Contador::Contador(int vlr) :

                vlrCont(vlr)

{

        cout << "\nConstruindo e inicializando...\n";

} // Fim de Contador::Contador(int)

// Destrutor.

Contador::~Contador()

{

        cout << "\nDestruindo Contador...\n";

} // Fim de Contador::~Contador()

unsigned int Contador::acessaVal() const

{
```

```
        return vlrCont;
} // Fim de Contador::acessaVal()
void Contador::defineVal(unsigned int val)
{
        vlrCont = val;
} // Fim de Contador::defineVal()
void Contador::mostraVal() const
{
        cout << "\nValor = "
                << acessaVal();
} // Fim de Contador::mostraVal()
Contador Contador::operator++()
{
        cout << "\nIncrementando...";
        ++vlrCont;
        // Retorna um objeto temporário
        // sem nome.
        return Contador(vlrCont);
} // Fim de Contador::operator++()
int main()
{
```

```
        // Dois objetos Contador.

        Contador cont1, cont2;

        // Exibe valor.

        cout << "\nObjeto cont1";

        cont1.mostraVal();

        // Incrementa e atribui.

        cont2 = ++cont1;

        // Exibe novo objeto.

        cout << "\nObjeto cont2";

        cont2.mostraVal();

        return 0;

} // Fim de main()

//----------------------------------------
```

Exercício

Modifique o exemplo `SobreO8.cpp`, removendo as mensagens dos construtores e destrutores.

Teoria

Vimos que o ponteiro `this` é passado para todas as funções membro de uma classe. Portanto, ele é passado também para o operador ++. Esse ponteiro aponta para o próprio objeto `Contador`, de modo que se for de-referenciado, ele retornará o objeto, já com o valor correto na variável `vlrCont`. O ponteiro `this` pode então ser de-referenciado, evitando assim a necessidade de criação de um objeto temporário.

Exemplo

```
//-------------------------------------------------------
// SobreO10.cpp
// Ilustra sobrecarga
// de operadores.
// Agora operador++
// utiliza o ponteiro this
// para retornar um objeto.
#include <iostream.h>
// Declara a classe Contador.
class Contador
{
        unsigned int vlrCont;
public:
        // Construtor.
        Contador();
        // Destrutor.
        ~Contador();
        unsigned int acessaVal() const;
        void defineVal(unsigned int val);
        void mostraVal() const;
        const Contador& operator++();
```

```
}; // Fim de class Contador.

// Implementação.

// Construtor.

Contador::Contador() :

                vlrCont(0)

{

        cout << "\nConstruindo Contador...\n";

} // Fim de Contador::Contador()

// Destrutor.

Contador::~Contador()

{

        cout << "\nDestruindo Contador...\n";

} // Fim de Contador::~Contador()

unsigned int Contador::acessaVal() const

{

        return vlrCont;

} // Fim de Contador::acessaVal()

void Contador::defineVal(unsigned int val)

{

        vlrCont = val;
```

```
} // Fim de Contador::defineVal()

void Contador::mostraVal() const

{

        cout << "\nValor = "

                << acessaVal();

} // Fim de Contador::mostraVal()

const Contador& Contador::operator++()

{

        cout << "\nIncrementando...";

        ++vlrCont;

        // Utiliza o ponteiro this

        // para retornar uma

        // referência a este objeto.

        return *this;

} // Fim de Contador::operator++()

int main()

{

        // Dois objetos Contador.

        Contador cont1, cont2;

        // Exibe valor.

        cout << "\nObjeto cont1";
```

```
        cont1.mostraVal();

        // Incrementa e atribui.

        cont2 = ++cont1;

        // Exibe novo objeto.

        cout << "\nObjeto cont2";

        cont2.mostraVal();

        return 0;

} // Fim de main()

//-------------------------------------------------------
```

Exercício

Modifique o exemplo `SobreO10.cpp`, removendo as mensagens dos construtores e destrutores.

Teoria

Neste exemplo, vamos escrever o operador ++ em sufixo. Apenas para relembrar, a diferença entre o operador ++ em prefixo e em sufixo é a seguinte:

O operador em prefixo diz "incremente, depois acesse o valor"; o operador em sufixo diz "acesse o valor, depois incremente".

Assim, enquanto o operador em prefixo pode simplesmente incrementar o valor e depois retornar o próprio objeto, o operador em sufixo deve retornar o valor que existe antes de ser incrementado. Para isso, é preciso criar um objeto temporário que conterá o valor original, depois incrementar o valor do objeto original, e finalmente retornar o objeto temporário.

Por exemplo, observe a linha:

```
y = x++;
```

Se x for igual a 10, depois dessa linha y será igual a 10, e x será igual a 11. Assim, retornamos o valor de x e o atribuímos a y, e depois incrementamos o valor de x. Se x for um objeto, o operador de incremento em sufixo precisa guardar o valor original (10) em um objeto temporário, incrementar o valor de x para 11, e depois retornar o valor temporário e atribui-lo a y.

Observe que como estamos retornando um valor temporário, precisamos retorná-lo por valor, e não por referência, já que o objeto temporário sairá de escopo tão logo a função retorne.

Observe que o operador ++ em sufixo é declarado como recebendo um parâmetro int. Esse parâmetro é apenas para indicar que se trata de um operador em sufixo. Na prática, esse valor nunca é utilizado.

Exemplo

```
//--------------------------------------------------------

// SobreO12.cpp

// Ilustra sobrecarga

// de operadores.

// Ilustra operador++

// em sufixo.

#include <iostream.h>

// Declara a classe Contador.

class Contador

{

        unsigned int vlrCont;

public:
```

```
        // Construtor.
        Contador();
        // Destrutor.
        ~Contador();
        unsigned int acessaVal() const;
        void defineVal(unsigned int val);
        void mostraVal() const;
        // Sufixo.
        const Contador operator++(int);
}; // Fim de class Contador.
// Implementação.
// Construtor.
Contador::Contador() :
                vlrCont(0)
{
} // Fim de Contador::Contador()
// Destrutor.
Contador::~Contador()
{
} // Fim de Contador::~Contador()
unsigned int Contador::acessaVal() const
```

```
{
        return vlrCont;
} // Fim de Contador::acessaVal()
void Contador::defineVal(unsigned int val)
{
        vlrCont = val;
} // Fim de Contador::defineVal()
void Contador::mostraVal() const
{
        cout << "\nValor = "
                << acessaVal();
} // Fim de Contador::mostraVal()
// operador++ sufixo.
const Contador Contador::operator++(int)
{
        cout << "\nIncrementando com sufixo...";
        // Cria contador temporário.
        Contador temp(*this);
        // Incrementa.
        ++vlrCont;
        // Retorna contador temporário.
```

```
        return temp;
} // Fim de Contador Contador::operator++(int i)
int main()
{
        // Dois objetos Contador.
        Contador cont1, cont2;
        // Exibe valor.
        cout << "\nObjeto cont1";
        cont1.mostraVal();
        // Incrementa com sufixo
        // e atribui.
        cont2 = cont1++;
        // Exibe valores.
        cout << "\n*** Novos valores ***\n";
        cout << "\nObjeto cont1";
        cont1.mostraVal();
        cout << "\nObjeto cont2";
        cont2.mostraVal();
        return 0;
} // Fim de main()
//-------------------------------------------------
```

Exercício

Modifique o exemplo `Sobre012.cpp`, para que o operador `++` seja usado nas duas posições, prefixo e sufixo.

Sobrecarregando o operador +

Teoria

Vimos como sobrecarregar o operador ++, que é unário. Ele opera somente sobre um objeto. O operador de adiçao + é um operador binário, que trabalha com dois objetos.

O objetivo aqui é poder declarar duas variáveis `Contador` e poder somá-las como no exemplo:

```
Contador c1, c2, c3;

c3 = c2 + c1;
```

Começaremos escrevendo uma função `soma()`, que recebe um `Contador` como argumento, soma os valores e depois retorna um `Contador` como resultado.

Exemplo

```
//--------------------------------------------------------

// SobrePl.cpp

// Ilustra sobrecarga

// de operadores.

// Ilustra sobrecarga do
```

```
// operador +
// Inicialmente, função soma()
#include <iostream.h>
// Declara a classe Contador.
class Contador
{
        unsigned int vlrCont;
public:
        // Construtor.
        Contador();
        // Construtor com inicialização.
        Contador(unsigned int vlr);
        // Destrutor.
        ~Contador();
        unsigned int acessaVal() const;
        void defineVal(unsigned int val);
        void mostraVal() const;
        // A função soma()
        Contador soma(const Contador&);
}; // Fim de class Contador.
// Implementação.
```

```
// Construtor.
Contador::Contador() :
                vlrCont(0)
{
} // Fim de Contador::Contador()
// Construtor com inicialização.
Contador::Contador(unsigned int vlr) :
vlrCont(vlr)
{
} // Fim de Contador::Contador(unsigned int)
// Destrutor.
Contador::~Contador()
{
} // Fim de Contador::~Contador()
unsigned int Contador::acessaVal() const
{
        return vlrCont;
} // Fim de Contador::acessaVal()
void Contador::defineVal(unsigned int val)
{
        vlrCont = val;
```

```
} // Fim de Contador::defineVal()

void Contador::mostraVal() const

{

        cout << "\nValor = "

                << acessaVal();

} // Fim de Contador::mostraVal()

// Função soma()

Contador Contador::soma(const Contador& parcela)

{

        return Contador(vlrCont +

                parcela.acessaVal());

} // Fim de Contador::soma(const Contador&)

int main()

{

        // Três objetos Contador.

        Contador parc1(4), parc2(3), result;

        // Soma contadores.

        result = parc1.soma(parc2);

        // Exibe valores.

        cout << "\n*** Valores finais ***";

        cout << "\nParcela1: ";
```

```
        parc1.mostraVal();

        cout << "\nParcela2: ";

        parc2.mostraVal();

        cout << "\nResultado: ";

        result.mostraVal();

        return 0;

} // Fim de main()

//-------------------------------------------------
```

Exercício

Modifique o exemplo `SobrePl.cpp`, de maneira que sejam somadas três parcelas, ao invés de duas.

Teoria

A função `soma()` funciona, mas seu uso é pouco intuitivo. A sobrecarga do operador `+` tornará o uso da classe `Contador` mais natural. Eis como isso é feito.

Exemplo

```
//-------------------------------------------------

// SobrePl2.cpp

// Ilustra sobrecarga

// de operadores.

// Implementa operador +

#include <iostream.h>
```

```
// Declara a classe Contador.
class Contador
{
        unsigned int vlrCont;
public:
        // Construtor.
        Contador();
        // Construtor com inicialização.
        Contador(unsigned int vlr);
        // Destrutor.
        ~Contador();
        unsigned int acessaVal() const;
        void defineVal(unsigned int val);
        void mostraVal() const;
        // O operador +
        Contador operator+(const Contador&);
}; // Fim de class Contador.
// Implementação.
// Construtor.
Contador::Contador() :
                vlrCont(0)
```

```
{
} // Fim de Contador::Contador()
// Construtor com inicialização.
Contador::Contador(unsigned int vlr) :
vlrCont(vlr)
{
} // Fim de Contador::Contador(unsigned int)
// Destrutor.
Contador::~Contador()
{
} // Fim de Contador::~Contador()
unsigned int Contador::acessaVal() const
{
        return vlrCont;
} // Fim de Contador::acessaVal()
void Contador::defineVal(unsigned int val)
{
        vlrCont = val;
} // Fim de Contador::defineVal()
void Contador::mostraVal() const
{
```

```
        cout << "\nValor = "
                << acessaVal();
} // Fim de Contador::mostraVal()
// Operador +
Contador Contador::operator+(const Contador& parcela)
{
        return Contador(vlrCont +
                parcela.acessaVal());
} // Fim de Contador::operator+(const Contador&)
int main()
{
        // Três objetos Contador.
        Contador parc1(4), parc2(3), result;
        // Soma dois contadores.
        result = parc1 + parc2;
        // Exibe valores.
        cout << "\n*** Valores finais ***";
        cout << "\nParcela1: ";
        parc1.mostraVal();
        cout << "\nParcela2: ";
        parc2.mostraVal();
```

```
        cout << "\nResultado: ";

        result.mostraVal();

        return 0;

} // Fim de main()

//-----------------------------------------
```

Exercício

Modifique o exemplo `SobrePl2.cpp`, de maneira que sejam somadas três parcelas, ao invés de duas.

Sobrecarregando o operador =

Teoria

O operador = é um dos operadores fornecidos por default pelo compilador, mesmo para os tipos (classes) definidos pelo programador. Mesmo assim, pode surgir a necessidade de sobrecarregá-lo.

Quando sobrecarregamos o operador =, precisamos levar em conta um aspecto adicional.

Digamos que temos dois objetos `Contador`, `c1` e `c2`.

Com o operador de atribuição, podemos atribuir `c2` a `c1`, da seguinte forma:

```
c1 = c2;
```

O que acontece se uma das variáveis membro for um ponteiro? E o que acontece com os valores originais de `c1`?

Lembremos o conceito de cópia rasa e cópia profunda. Uma cópia rasa apenas copia os membros, e os dois objetos acabam apontando para a mesma área do free store. Uma cópia profunda aloca a memória necessária.

Há ainda outra questão. O objeto `c1` já existe na memória, e tem sua memória alocada. Essa memória precisa ser deletada, para evitar vazamentos de memória. Mas o que acontece se atribuirmos um objeto a si mesmo, da seguinte forma:

```
c1 = c1;
```

Ninguém vai fazer isso de propósito, mas se acontecer, o programa precisa ser capaz de lidar com isso. E mais importante, isso pode acontecer por acidente, quando referências e ponteiros de-referenciados ocultam o fato de que a atribuição está sendo feita ao próprio objeto.

Se essa questão não tiver sido tratada com cuidado, `c1` poderá deletar sua memória alocada. Depois, no momento de copiar a memória do lado direito da atribuição, haverá um problema: a memória terá sido deletada.

Para evitar esse problema, o operador de atribuição deve checar se o lado direito do operador de atribuição é o próprio objeto. Isso é feito examinando o ponteiro `this`. O exemplo abaixo ilustra esse procedimento.

Exemplo

```
//--------------------------------------------------------

// SobreAtr.cpp

// Ilustra sobrecarga

// de operadores.

// Implementa operador =

#include <iostream.h>

// Declara a classe Contador.

class Contador

{

        unsigned int vlrCont;

public:

        // Construtor.

        Contador();
```

```
        // Construtor com inicialização.
        Contador(unsigned int vlr);
        // Destrutor.
        ~Contador();
        unsigned int acessaVal() const;
        void defineVal(unsigned int val);
        void mostraVal() const;
        // O operador =
        Contador& operator=(const Contador&);
}; // Fim de class Contador.
// Implementação.
// Construtor.
Contador::Contador() :
                vlrCont(0)
{
} // Fim de Contador::Contador()
// Construtor com inicialização.
Contador::Contador(unsigned int vlr) :
                vlrCont(vlr)
{
```

```
} // Fim de Contador::Contador(unsigned int)

// Destrutor.

Contador::~Contador()

{

} // Fim de Contador::~Contador()

unsigned int Contador::acessaVal() const

{

        return vlrCont;

} // Fim de Contador::acessaVal()

void Contador::defineVal(unsigned int val)

{

        vlrCont = val;

} // Fim de Contador::defineVal()

void Contador::mostraVal() const

{

        cout << "\nValor = "

                << acessaVal();

} // Fim de Contador::mostraVal()

// Operador =

Contador& Contador::operator=(const Contador& outro)

{
```

```
        vlrCont = outro.acessaVal();
        return *this;
} // Fim de Contador::operator=(const Contador&)
int main()
{
        // Dois objetos Contador.
        Contador cont1(40), cont2(3);
        // Exibe valores iniciais.
        cout << "\n*** Valores iniciais ***";
        cout << "\ncont1: ";
        cont1.mostraVal();
        cout << "\ncont2: ";
        cont2.mostraVal();
        // Atribui.
        cont1 = cont2;
        // Exibe novos valores.
        cout << "\n*** Apos atribuicao ***";
        cout << "\ncont1: ";
        cont1.mostraVal();
        cout << "\ncont2: ";
        cont2.mostraVal();
```

```
        return 0;

} // Fim de main()

//-------------------------------------------------
```

Exercício

Modifique o exemplo `SobreAtr.cpp`, de maneira que a variável membro `vlrCont` da classe `Contador` seja um ponteiro. Faça as mudanças necessárias na implementação do operador =

Conversão entre objetos e tipos simples

Teoria

O que acontece quando tentamos converter uma variável de um tipo simples, como `int`, em um objeto de uma classe definida pelo programador?

A classe para a qual desejamos converter o tipo simples precisará ter um construtor especial, com essa finalidade. Esse construtor deverá receber como argumento o tipo simples a ser convertido.

O exemplo abaixo ilustra essa situação.

Exemplo

```
//---------------------------------------------------------

// ConvObj.cpp

// Ilustra conversão de

// um tipo simples

// em um objeto.

// ATENÇÃO: ESTE PROGRAMA

// CONTÉM UM ERRO DELIBERADO.

#include <iostream.h>
```

```
// Declara a classe Contador.
class Contador
{
        unsigned int vlrCont;
public:
        // Construtor.
        Contador();
        // Destrutor.
        ~Contador();
        unsigned int acessaVal() const;
        void defineVal(unsigned int val);
        void mostraVal() const;
}; // Fim de class Contador.
// Implementação.
// Construtor.
Contador::Contador() :
                vlrCont(0)
{
} // Fim de Contador::Contador()
// Destrutor.
```

```
Contador::~Contador()
{
} // Fim de Contador::~Contador()
unsigned int Contador::acessaVal() const
{
        return vlrCont;
} // Fim de Contador::acessaVal()
void Contador::defineVal(unsigned int val)
{
        vlrCont = val;
} // Fim de Contador::defineVal()
void Contador::mostraVal() const
{
        cout << "\nValor = "
                << acessaVal();
} // Fim de Contador::mostraVal()
int main()
{
        // Uma variável unsigned int.
        unsigned int uiVar = 50;
        // Um objeto Contador.
```

```
        Contador cont;

        // Tenta converter unsigned int

        // em contador.

        cont = uiVar;

        // Exibe valor.

        cout << "\nValor de cont: ";

        cont.mostraVal();

        return 0;

} // Fim de main()

//-------------------------------------------------
```

Exercício

Modifique o exemplo `ConvObj.cpp`, implementando o construtor necessário para realizar a conversão.

Teoria

Como é feita a conversão em sentido oposto, de um objeto para um tipo simples?

Para solucionar esse tipo de problema, C++ oferece a possibilidade de acrescentar a uma classe os operadores de conversão. Isso permite que uma classe especifique como devem ser feitas conversões implícitas para tipos simples.

Exemplo

```
//-----------------------------------------------------

// ConvObj2.cpp

// Ilustra conversão
```

```
// de um objeto
// em um tipo simples.
// ATENÇÃO: ESTE PROGRAMA
// CONTÉM UM ERRO PROPOSITAL.
#include <iostream.h>
// Declara a classe Contador.
class Contador
{
        unsigned int vlrCont;
public:
        // Construtor.
        Contador();
        // Construtor com inicialização.
        Contador(unsigned int vlr);
        // Destrutor.
        ~Contador();
        unsigned int acessaVal() const;
        void defineVal(unsigned int val);
        void mostraVal() const;
}; // Fim de class Contador.
// Implementação.
```

```
// Construtor.
Contador::Contador() :
                vlrCont(0)
{
} // Fim de Contador::Contador()
// Construtor com inicialização.
Contador::Contador(unsigned int vlr) :
                vlrCont(vlr)
{
        cout << "\nConstruindo e inicializando...\n";
} // Fim de Contador::Contador(unsigned int)
// Destrutor.
Contador::~Contador()
{
} // Fim de Contador::~Contador()
unsigned int Contador::acessaVal() const
{
        return vlrCont;
} // Fim de Contador::acessaVal()
void Contador::defineVal(unsigned int val)
{
```

```
        vlrCont = val;
} // Fim de Contador::defineVal()
void Contador::mostraVal() const
{
        cout << "\nValor = "
                << acessaVal();
} // Fim de Contador::mostraVal()
int main()
{
        // Uma variável unsigned int.
        unsigned int uiVar;
        // Um objeto Contador.
        Contador cont(500);
        // Tenta converter contador
        // em unsigned int.
        uiVar = cont;
        // Exibe valores.
        cout << "\nValor de cont: ";
        cont.mostraVal();
        cout << "\nValor de uiVar = "
                << uiVar;
```

```
        return 0;

} // Fim de main()

//---------------------------------------------------------
```

Exercício

No exemplo `ConvObj2.cpp` implemente o operador necessário para a conversão de `Contador` para `unsigned int`.

Arrays de objetos

Teoria

Qualquer objeto, seja de um tipo simples ou de uma classe definida pelo programador, pode ser armazenado em um array. Quando declaramos o array, informamos ao compilador qual o tipo de objeto a ser armazenado, bem como o tamanho do array. O compilador sabe então quanto espaço alocar, dependendo do tipo de objeto. No caso de objetos de uma classe, o compilador sabe quanto espaço precisa ser alocado para cada objeto com base na declaração da classe. A classe deve ter um construtor default, que não recebe nenhum argumento, de modo que os objetos possam ser criados quando o array é definido.

O acesso aos membros de dados em um array de objetos é um processo em dois passos. Primeiro, identificamos o membro do array, com o auxílio do operador de índice `[]`, e depois aplicamos o operador ponto `.` para acessar o membro.

Exemplo

```
//--------------------------------------------------------

// ArrObj.cpp

// Ilustra o uso de

// arrays de objetos.

#include <iostream.h>
```

```
class Cliente
{
        int numCliente;
        float saldo;
public:
        // Construtor.
        Cliente();
        int acessaNum() const;
        float acessaSaldo() const;
        void defineNum(int num);
        void defineSaldo(float sal);
}; // Fim de class Cliente.
// Definições.
Cliente::Cliente()
{
        numCliente = 0;
        saldo = 0.0;
} // Fim de Cliente::Cliente()
int Cliente::acessaNum() const
{
        return numCliente;
```

```
} // Fim de Cliente::acessaNum()

float Cliente::acessaSaldo() const

{

        return saldo;

} // Fim de Cliente::acessaSaldo()

void Cliente::defineNum(int num)

{

        numCliente = num;

} // Fim de Cliente::defineNum()

void Cliente::defineSaldo(float sal)

{

        saldo = sal;

} // Fim de Cliente::defineSaldo()

int main()

{

        // Um array de clientes.

        Cliente arrayClientes[5];

        // Inicializa.

        for(int i = 0; i < 5; i++)

        {

                arrayClientes[i].defineNum(i + 1);
```

```
                arrayClientes[i].defineSaldo((float)(i + 1) * 25);

        } // Fim de for(int i...

        // Exibe.

        for(int i = 0; i < 5; i++)

        {

                cout << "\nCliente: "

                        << arrayClientes[i].acessaNum()

                        << "\tSaldo = "

                        << arrayClientes[i].acessaSaldo();

        } // Fim de for(int i = 1...

        return 0;

} // Fim de main()

//-------------------------------------------------------
```

Exercício

Modifique o exemplo `ArrObj.cpp` acrescentando à classe `Cliente` uma variável membro `bool` para indicar se o cliente é ou não cliente preferencial.

Uma classe string

Teoria

Atualmente, todos os compiladores C++ em conformidade com o padrão ANSI/ISO vêm com uma classe `string`, o que facilita bastante a manipulação de strings.

Entretanto, como exercício de programação vamos implementar nossa própria classe string. Para evitar confusões, vamos chamá-la de `ClString`.

Exemplo

```
//-------------------------------------------------------------

// ClStr.cpp

// Ilustra um exemplo

// de classe string.

#include <iostream.h>

#include <string.h>

// Declara a classe.

class ClString

{
```

```
        // A string propriamente dita.
        char* str;
        // O comprimento da string.
        unsigned int compr;
        // Um construtor private.
        ClString(unsigned int);
public:
        // Construtores.
        ClString();
        ClString(const char* const);
        ClString(const ClString&);
        // Destrutor.
        ~ClString();
        // Operadores sobrecarregados.
        char& operator[](unsigned int posicao);
        char operator[](unsigned int posicao) const;
        ClString operator+(const ClString&);
        void operator +=(const ClString&);
        ClString& operator =(const ClString&);
        // Métodos de acesso.
        unsigned int acessaCompr() const
```

```
        {
                return compr;
        } // Fim de acessaCompr()
        const char* acessaStr() const
        {
                return str;
        } // Fim de acessaStr()
}; // Fim de class ClString.
// Implementações.
// Construtor default.
// Cria uma string de
// comprimento zero.
ClString::ClString()
{
        // cout << "\nConstrutor default...\n";
        str = new char[1];
        str[0] = '\0';
        compr = 0;
} // Fim de ClString::ClString()
// Construtor private.
// Usado somente pelos
```

```
// métodos da classe.
// Cria uma string com o
// comprimento especificado
// e a preenche com o
// caractere '\0'
ClString::ClString(unsigned int comp)
{
        // cout << "\nConstrutor private...\n";
        str = new char[comp + 1];
        for(unsigned int i = 0; i <= comp; i++)
        str[i] = '\0';
        compr = comp;
} // Fim de ClString::ClString(unsigned int)
// Constroi um objeto ClString
// a partir de um array de caracteres.
ClString::ClString(const char* const cArray)
{
        // cout << "\nConstruindo de array...\n";
        compr = strlen(cArray);
        str = new char[compr + 1];
        for(unsigned int i = 0; i < compr; i++)
```

```
                str[i] = cArray[i];
        str[compr] = '\0';
} // Fim de ClString::ClString(const char* const)
// Construtor de cópia.
ClString::ClString(const ClString& strRef)
{
        // cout << "\nConstrutor de copia...\n";
        compr = strRef.acessaCompr();
        str = new char[compr + 1];
        for(unsigned int i = 0; i < compr; i++)
                str[i] = strRef[i];
        str[compr] = '\0';
} // Fim de ClString::ClString(const String&)
// Destrutor.
ClString::~ClString()
{
        // cout << "\nDestruindo string...\n";
        delete[] str;
        compr = 0;
} // Fim de ClString::~ClString()
// Operador =
```

```
// Libera memória atual;
// Copia string e compr.
ClString& ClString::operator=(const ClString& strRef)
{
        if(this == &strRef)
        return *this;
        delete[] str;
        compr = strRef.acessaCompr();
        str = new char[compr + 1];
        for(unsigned int i = 0; i < compr; i++)
                str[i] = strRef[i];
        str[compr] = '\0';
        return *this;
} // Fim de ClString::operator=(const ClString&)
// Operador de posição não-constante.
// Referência permite que char
// seja modificado.
char& ClString::operator[](unsigned int pos)
{
        if(pos > compr)
                return str[compr - 1];
```

```
        else
                return str[pos];
} // Fim de ClString::operator[]()
// Operador de posição constante.
// Para uso em objetos const.
char ClString::operator[](unsigned int pos) const
{
        if(pos > compr)
                return str[compr - 1];
        else
                return str[pos];
} // Fim de ClString::operator[]()
// Concatena duas strings.
ClString ClString::operator+(const ClString& strRef)
{
        unsigned int comprTotal = compr + strRef.acessaCompr();
        ClString tempStr(comprTotal);
        unsigned int i;
        for(i = 0; i < compr; i++)
                tempStr[i] = str[i];
```

```
        for(unsigned int j = 0;
                        j < strRef.acessaCompr();
                        j++, i++)
                tempStr[i] = strRef[j];
        tempStr[comprTotal] = '\0';
        return tempStr;
} // Fim de ClString::operator+()
void ClString::operator+=(const ClString& strRef)
{
        unsigned int comprRef = strRef.acessaCompr();
        unsigned int comprTotal = compr + comprRef;
        ClString tempStr(comprTotal);
        unsigned int i;
        for(i = 0; i < compr; i++)
                tempStr[i] = str[i];
        for(unsigned int j = 0;
                j < strRef.acessaCompr();
                j++, i++)
                tempStr[i] = strRef[i - compr];
        tempStr[comprTotal] = '\0';
        *this = tempStr;
```

```
} // Fim de ClString::operator+=()

int main()

{

        // Constroi string a partir

        // de array de char.

        ClString str1("Tarcisio Lopes");

        // Exibe.

        cout << "str1:\t" << str1.acessaStr() << '\n';

        // Atribui array de chars

        // a objeto ClString.

        char* arrChar = "O rato roeu ";

        str1 = arrChar;

        // Exibe.

        cout << "str1:\t" << str1.acessaStr() << '\n';

        // Cria um segundo

        // array de chars.

        char arrChar2[64];

        strcpy(arrChar2, "a roupa do rei.");

        // Concatena array de char

        // com objeto ClString.

        str1 += arrChar2;
```

```
        // Exibe.
        cout << "arrChar2:\t" << arrChar2 << '\n';
        cout << "str1:\t" << str1.acessaStr() << '\n';
        // Coloca R maiúsculo.
        str1[2] = str1[7] = str1[14] = str1[23] = 'R';
        // Exibe.
        cout << "str1:\t" << str1.acessaStr() << '\n';
        return 0;
} // Fim de main()
//--------------------------------------------------
```

Exercício

Modifique o exemplo `ClStr.cpp`, utilizando a classe `ClString` para:

(a) exibir os caracteres de uma string separados por um espaço, usando o operador `[ ]`

(b) transformar todas as letras da string em MAIÚSCULAS

(c) concatenar duas strings usando o operador `+`

Exemplo de lista encadeada

[

Teoria

Os arrays são muito práticos, mas têm uma séria limitação: seu tamanho é fixo. Ao definir o tamanho do array, se errarmos pelo excesso, estaremos desperdiçando espaço de armazenamento. Se errarmos pela falta, o conteúdo pode estourar o tamanho do array, criando sérios problemas.

Uma forma de contornar essa limitação é com o uso de uma lista encadeada. Uma lista encadeada é uma estrutura de dados que consiste de pequenos containers, que podem ser encadeados conforme necessário. A idéia da lista encadeada é que ela pode conter um objeto de uma determinada classe. Se houver necessidade, podemos acrescentar mais objetos, fazendo com que o último objeto da lista aponte para o novo objeto, recém acrescentado. Ou seja, criamos um container para cada objeto e encadeamos esses containers conforme a necessidade.

Os containers são chamados de nós. O primeiro nó é chamado cabeça da lista; o último nó é chamado de cauda. As listas podem ser de três tipos:

(a) Simplesmente encadeadas

(b) Duplamente encadeadas

(c) Árvores

Em uma lista simplesmente encadeada, cada nó aponta para o nó seguinte, mas o nó seguinte não aponta para o nó anterior. Para encontrar um determinado nó, começamos da cabeça da lista e seguimos nó por nó. Uma lista duplamente encadeada permite

movimentar-se para a frente e para trás na cadeia. Uma árvore é uma estrutura complexa, construída com nós, sendo que cada um deles aponta para dois ou três outros nós.

O exemplo abaixo ilustra a construção de uma lista simplesmente encadeada.

Exemplo

```
//-------------------------------------------------

// ListEnc.cpp

// Ilustra a criação

// de uma lista encadeada.

#include <iostream.h>

#include <assert.h>

// Define a classe de

// objetos que formarão

// a lista.

class Cliente

{

        int numCliente;

public:

        // Construtores.

        Cliente() {numCliente = 1;}

        Cliente(int num) : numCliente(num){}

        // Destrutor.
```

```
        ~Cliente(){}
        // Método de acesso.
        int acessaNum() const {return numCliente;}
}; // Fim de class Cliente.
// Uma classe para gerenciar e
// ordenar a lista.
class No
{
        Cliente* pCliente;
        No* proxNo;
public:
        // Construtor.
        No(Cliente*);
        // Destrutor.
        ~No();
        // Outros métodos.
        void defineProx(No* pNo) {proxNo = pNo;}
        No* acessaProx() const {return proxNo;}
        Cliente* acessaCli() const {return pCliente;}
        void insere(No*);
        void Exibe();
```

```
}; // Fim de class No.

// Implementação.

// Construtor.

No::No(Cliente* pCli): pCliente(pCli), proxNo(0)

{

} // Fim de No::No(Cliente*)

// Destrutor.

No::~No()

{

        cout << "Deletando No...\n";

        delete pCliente;

        pCliente = 0;

        delete proxNo;

        proxNo = 0;

} // Fim de No::~No()


// Método insere()

// Ordena clientes pelo número.

// Algoritmo: Se este cliente é o último da fila,

// acrescenta o novo cliente. Caso contrário,

// se o novo cliente tem número maior que o
```

```
// cliente atual e menor que o próximo da fila,
// insere-o depois deste. Caso contrário,
// chama insere() para o próximo cliente da fila.
void No::insere(No* novoNo)
{
        if(!proxNo)
                proxNo = novoNo;
        else
        {
                int numProxCli = proxNo->acessaCli()->acessaNum();
                int novoNum = novoNo->acessaCli()->acessaNum();
                int numDeste = pCliente->acessaNum();
                assert(novoNum >= numDeste);
                if(novoNum < numProxCli)
                {
                        novoNo->defineProx(proxNo);
                        proxNo = novoNo;
                } // Fim de if(novoNum < numProxCli)
                else
                        proxNo->insere(novoNo);
        } // Fim de else (externo)
```

```
} // Fim de No::insere(No*)

void No::Exibe()

{

        if(pCliente->acessaNum() > 0)

        {

                cout << "Num. do Cliente = ";

                cout << pCliente->acessaNum() << "\n";

        } // Fim de if(pCliente->...

        if(proxNo)

                proxNo->Exibe();

} // Fim de No::Exibe()

int main()

{

        // Um ponteiro para nó.

        No* pNo;

        // Um ponteiro para

        // Cliente, inicializado.

        Cliente* pontCli = new Cliente(0);

        // Um array de ints para

        // fornecer números de clientes.
```

```
        int numeros[] = {19, 48, 13, 17, 999, 18, 7, 0};
        // Exibe números.
        cout << "\n*** Nums. fornecidos ***\n";
        for(int i = 0; numeros[i]; i++)
                cout << numeros[i] << " ";
        cout << "\n";
        No* pCabeca = new No(pontCli);
        // Cria alguns nós.
        for(int i = 0; numeros[i] != 0; i++)
        {
                pontCli = new Cliente(numeros[i]);
                pNo = new No(pontCli);
                pCabeca->insere(pNo);
        } // Fim de for(int i = 0...
        cout << "\n*** Lista ordenada ***\n";
        pCabeca->Exibe();
        cout << "\n*** Destruindo lista ***\n";
        delete pCabeca;
        cout << "\n*** Encerrando... ***\n";
        return 0;
} // Fim de main()
```

```
//-------------------------------------------------
```

Exercício

Modifique o exemplo `ListEnc.cpp` de maneira que os números usados para inicializar os objetos `Cliente` contidos na lista encadeada sejam solicitados do usuário.

Introdução a herança

Teoria

Uma das formas que usamos para simplificar a organização e o desenvolvimento de programas complexos é procurar simular nos programas a organização do mundo real.

No mundo real, muitas vezes as coisas são organizadas em hierarquias. Eis alguns exemplos de hierarquias do mundo real:

Um carro e um caminhão são veículos automotores; um veículo automotor é um meio de transporte; um meio de transporte é uma máquina.

Dálmata, pequinês e pastor alemão são raças de cães; um cão é um mamífero, tal como um gato e uma baleia; um mamífero é um animal; um animal é um ser vivo.

Um planeta é um astro; astros podem ou não ter luz própria; todo astro é um corpo celeste.

Uma hierarquia estabelece um relacionamento do tipo é-um. Um cachorro é um tipo de canino. Um fusca é um tipo de carro, que é um tipo de veículo automotor. Um sorvete é um tipo de sobremesa, que é um tipo de alimento.

O que significa dizer que uma coisa é um tipo de outra coisa? Significa que uma coisa é uma forma mais especializada de outra. Um carro é um tipo especializado de veículo automotor. Um caminhão é outra forma especializada de veículo automotor. Um trator é ainda outra forma especializada de veículo automotor.

Exemplo

```
//-------------------------------------------------------
// IntrHer.cpp
// Apresenta o uso
// de herança.
#include <iostream.h>
class ClasseBase
{
protected:
        int m_propr_base1;
        int m_propr_base2;
public:
        // Construtores.
        ClasseBase() : m_propr_base1(10),
                m_propr_base2(20) {}
        // Destrutor.
        ~ClasseBase() {}
        // Métodos de acesso.
        int acessaPropr1() const {return m_propr_base1;}
        void definePropr1(int valor){ m_propr_base1 = valor;}
        int acessaPropr2() const {return m_propr_base2;}
        void definePropr2(int valor){m_propr_base2 = valor;}
```

```
        // Outros métodos.
        void met_base1() const
        {
                cout << "\nEstamos em met_base1...\n";
        } // Fim de met_base1()
        void met_base2() const
        {
                cout << "\nEstamos em met_base2...\n";
        } // Fim de met_base2()
}; // Fim de class ClasseBase
class ClasseDeriv : public ClasseBase
{
private:
        int m_propr_deriv;
public:
        // Construtor.
        ClasseDeriv() : m_propr_deriv(1000){}
        // Destrutor.
        ~ClasseDeriv() {};
        // Métodos de acesso.
        int acessaPropr_deriv() const
```

```
        {
                return m_propr_deriv;
        } // Fim de acessaPropr_deriv()
        void definePropr_deriv(int valor)
        {
                m_propr_deriv = valor;
        } // Fim de definePropr_deriv()
        // Outros métodos.
        void metodoDeriv1()
        {
                cout << "Estamos em metodoDeriv1()...\n";
        } // Fim de metodoDeriv1()
        void metodoDeriv2()
        {
                cout << "Estamos em metodoDeriv2()...\n";
        } // Fim de metodoDeriv2()
}; // Fim de class ClasseDeriv.
int main()
{
        // Cria um objeto
```

```
        // de ClasseDeriv.

        ClasseDeriv objDeriv;

        // Chama métodos da

        // classe base.

        objDeriv.met_base1();

        objDeriv.met_base2();

        // Chama métodos da

        // classe derivada.

        objDeriv.metodoDeriv1();

        cout << "Valor de m_propr_deriv = "

                << objDeriv.acessaPropr_deriv()

                << "\n";

        return 0;

} // Fim de main()

//-------------------------------------------------
```

Exercício

Escreva uma hierarquia de herança composta de uma classe base `Mamifero` e uma classe derivada `Cachorro`. Utilize a classe `Cachorro` em um programa, fazendo com que um objeto da classe derivada chame um método da classe base.

Ordem de chamada a construtores

Teoria

Vimos que a herança estabelece um relacionamento do tipo é um. Por exemplo, um cão é um mamífero; um carro é um veículo automotor; uma maçã é uma fruta.

Por isso, quando chamamos o construtor de uma classe derivada, antes é preciso executar o construtor da classe base. O exemplo abaixo ilustra esse fato.

Exemplo

```
//-------------------------------------------------------

// OrdCstr.cpp

// Ilustra a ordem

// de chamada a

// construtores.

#include <iostream.h>

class ClasseBase

{

protected:
```

```
        int m_propr_base1;
        int m_propr_base2;
public:
        // Construtores.
        ClasseBase() : m_propr_base1(10),
                m_propr_base2(20)
        {
                cout << "\nConstrutor de ClasseBase()...\n";
        } // Fim de ClasseBase()
        // Destrutor.
        ~ClasseBase()
        {
                cout << "\nDestrutor de ClasseBase()...\n";
        } // Fim de ~ClasseBase()
        // Métodos de acesso.
        int acessaPropr1() const {return m_propr_base1;}
        void definePropr1(int valor){ m_propr_base1 = valor;}
        int acessaPropr2() const {return m_propr_base2;}
        void definePropr2(int valor){m_propr_base2 = valor;}
        // Outros métodos.
        void met_base1() const
```

```
        {
                cout << "\nEstamos em met_base1...\n";
        } // Fim de met_base1()
        void met_base2() const
        {
                cout << "\nEstamos em met_base2...\n";
        } // Fim de met_base2()
}; // Fim de class ClasseBase
class ClasseDeriv : public ClasseBase
{
private:
        int m_propr_deriv;
public:
        // Construtor.
        ClasseDeriv() : m_propr_deriv(1000)
        {
                cout << "\nConstrutor de ClasseDeriv()...\n";
        } // Fim de ClasseDeriv()
        // Destrutor.
        ~ClasseDeriv()
        {
```

```
                cout << "\nDestrutor de ClasseDeriv()...\n";
        } // Fim de ~ClasseDeriv()
        // Métodos de acesso.
        int acessaPropr_deriv() const
        {
                return m_propr_deriv;
        } // Fim de acessaPropr_deriv()
        void definePropr_deriv(int valor)
        {
                m_propr_deriv = valor;
        } // Fim de definePropr_deriv()
        // Outros métodos.
        void metodoDeriv1()
        {
                cout << "Estamos em metodoDeriv1()...\n";
        } // Fim de metodoDeriv1()
        void metodoDeriv2()
        {
                cout << "Estamos em metodoDeriv2()...\n";
        } // Fim de metodoDeriv2()
}; // Fim de class ClasseDeriv.
```

```
int main()

{

        // Cria um objeto

        // de ClasseDeriv.

        ClasseDeriv objDeriv;

        // Chama métodos da

        // classe base.

        objDeriv.met_base1();

        objDeriv.met_base2();

        // Chama métodos da

        // classe derivada.

        objDeriv.metodoDeriv1();

        cout << "Valor de m_propr_deriv = "

                << objDeriv.acessaPropr_deriv()

                << "\n";

        return 0;

} // Fim de main()

//-------------------------------------------------------
```

Exercício

Crie uma hierarquia simples, composta pela classe base `Mamifero` e pela classe derivada `Cachorro`. Utilize essa hierarquia para ilustrar a ordem de chamada aos

construtores em um programa.

Argumentos para construtores da classe base

Teoria

Muitas vezes, ao criar um objeto, utilizamos um construtor que recebe parâmetros. Já vimos que, quando se trata de uma classe derivada, o construtor da classe base sempre é chamado. E se o construtor da classe base precisar receber parâmetros?
Como faremos para passar os parâmetros certos para o construtor da classe base?

O exemplo abaixo ilustra como isso é feito.

Exemplo

```
//--------------------------------------------------------------------------

// ArgCstr.cpp

// Ilustra a passagem

// de args para

// construtores da

// classe base.

#include <iostream.h>

enum VALORES {VLR1, VLR2, VLR3, VLR4, VLR5, VLR6};

class ClasseBase
```

```
{
protected:
        int m_propr_base1;
        int m_propr_base2;
public:
        // Construtores.
        ClasseBase();
        ClasseBase(int valor);
        // Destrutor.
        ~ClasseBase();
        // Métodos de acesso.
        int acessaPropr1() const {return m_propr_base1;}
        void definePropr1(int valor){ m_propr_base1 = valor;}
        int acessaPropr2() const {return m_propr_base2;}
        void definePropr2(int valor){m_propr_base2 = valor;}
        // Outros métodos.
        void met_base1() const
        {
                cout << "\nEstamos em met_base1...\n";
        } // Fim de met_base1()
        void met_base2() const
        {
```

```
                cout << "\nEstamos em met_base2...\n";
        } // Fim de met_base2()
}; // Fim de class ClasseBase
class ClasseDeriv : public ClasseBase
{
private:
        VALORES m_propr_deriv;
public:
        // Construtores.
        ClasseDeriv();
        ClasseDeriv(int propBase1);
        ClasseDeriv(int propBase1, int propBase2);
        // Destrutor.
        ~ClasseDeriv();
        // Métodos de acesso.
        VALORES acessaPropr_deriv() const
        {
                return m_propr_deriv;
        } // Fim de acessaPropr_deriv()
        void definePropr_deriv(VALORES valor)
        {
                m_propr_deriv = valor;
```

```
        } // Fim de definePropr_deriv()

        // Outros métodos.

        void metodoDeriv1()

        {

                cout << "Estamos em metodoDeriv1()...\n";

        } // Fim de metodoDeriv1()

        void metodoDeriv2()

        {

                cout << "Estamos em metodoDeriv2()...\n";

        } // Fim de metodoDeriv2()

}; // Fim de class ClasseDeriv.

// Implementações.

ClasseBase::ClasseBase() : m_propr_base1(10),

                m_propr_base2(20)

{

        cout << "\nConstrutor ClasseBase()...\n";

} // Fim de ClasseBase::ClasseBase()

ClasseBase::ClasseBase(int propr1) : m_propr_base1(propr1),

                m_propr_base2(20)

{

        cout << "\nConstrutor ClasseBase(int)...\n";

} // Fim de ClasseBase::ClasseBase(int)
```

```
// Destrutor.
ClasseBase::~ClasseBase()
{
        cout << "\nDestrutor ~ClasseBase()...\n";
} // Fim de ClasseBase::~ClasseBase()
// Construtores ClasseDeriv()
ClasseDeriv::ClasseDeriv() :
                ClasseBase(), m_propr_deriv(VLR3)
{
        cout << "\nConstrutor ClasseDeriv()\n";
} // Fim de ClasseDeriv::ClasseDeriv()
ClasseDeriv::ClasseDeriv(int propBase1) :
                ClasseBase(propBase1), m_propr_deriv(VLR3)
{
        cout << "\nConstrutor ClasseDeriv(int)\n";
} // Fim de ClasseDeriv::ClasseDeriv(int)
ClasseDeriv::ClasseDeriv(int propBase1, int propBase2) :
                ClasseBase(propBase1), m_propr_deriv(VLR3)
{
        m_propr_base2 = propBase2;
        cout << "\nConstrutor ClasseDeriv(int, int)\n";
} // Fim de ClasseDeriv::ClasseDeriv(int, int)
```

```
// Destrutor.
ClasseDeriv::~ClasseDeriv()
{
        cout << "\nDestrutor ~ClasseDeriv()\n";
} // Fim de ClasseDeriv::~ClasseDeriv()
int main()
{
        // Cria 3 objetos
        // de ClasseDeriv.
        ClasseDeriv objDeriv1;
        ClasseDeriv objDeriv2(2);
        ClasseDeriv objDeriv3(4, 6);
        // Chama métodos da
        // classe base.
        objDeriv1.met_base1();
        objDeriv2.met_base2();
        // Exibe valores.
        cout << "\nValores de objDeriv3: "
                << objDeriv3.acessaPropr1()
                << ", "
                << objDeriv3.acessaPropr2()
                << ", "
```

```
                << objDeriv3.acessaPropr_deriv();

        return 0;

} // Fim de main()

//-------------------------------------------------
```

Exercício

Reescreva a hierarquia classe base `Mamifero` classe derivada `Cachorro` de maneira a ilustrar a passagem de argumentos para o construtor da classe base.

Superposição de métodos

Teoria

Consideremos uma hierarquia composta de uma classe base, chamada classe `Mamifero`, e uma classe derivada, chamada classe `Cachorro`.

Um objeto da classe `Cachorro` tem acesso às funções membro da classe `Mamifero`. Além disso, a classe `Cachorro` pode acrescentar suas próprias funções membros, como por exemplo, `abanarCauda()`.

A classe cachorro pode ainda superpôr (override) uma função da classe base. Superpôr uma função significa mudar a implementação de uma função da classe base na classe derivada. Quando criamos um objeto da classe derivada, a versão correta da função é chamada.

Observe que, para que haja superposição, a nova função deve retornar o mesmo tipo e ter a mesma assinatura da função da classe base. Assinatura, refere-se ao protótipo da função, menos o tipo retornado: ou seja, o nome, a lista de parâmetros e a palavra-chave `const`, se for usada.

Exemplo

```
//--------------------------------------------------------

// Overrd.cpp

// Apresenta o uso
```

```
// da superposição
// de métodos (overriding)
#include <iostream.h>
class ClasseBase
{
protected:
        int m_propr_base1;
        int m_propr_base2;
public:
        // Construtores.
        ClasseBase() : m_propr_base1(10),
                m_propr_base2(20) {}
        // Destrutor.
        ~ClasseBase() {}
        // Métodos de acesso.
        int acessaPropr1() const {return m_propr_base1;}
        void definePropr1(int valor){ m_propr_base1 = valor;}
        int acessaPropr2() const {return m_propr_base2;}
        void definePropr2(int valor){m_propr_base2 = valor;}
        // Outros métodos.
        void met_base1() const
```

```
        {
                cout << "\nEstamos em met_base1...\n";
        } // Fim de met_base1()
        void met_base2() const
        {
                cout << "\nEstamos em met_base2...\n";
        } // Fim de met_base2()
}; // Fim de class ClasseBase
class ClasseDeriv : public ClasseBase
{
private:
        int m_propr_deriv;
public:
        // Construtor.
        ClasseDeriv() : m_propr_deriv(1000){}
        // Destrutor.
        ~ClasseDeriv() {};
        // Métodos de acesso.
        int acessaPropr_deriv() const
        {
                return m_propr_deriv;
```

```
        } // Fim de acessaPropr_deriv()

        void definePropr_deriv(int valor)

        {

                m_propr_deriv = valor;

        } // Fim de definePropr_deriv()

        // Outros métodos.

        void metodoDeriv1()

        {

                cout << "Estamos em metodoDeriv1()...\n";

        } // Fim de metodoDeriv1()

        void metodoDeriv2()

        {

                cout << "Estamos em metodoDeriv2()...\n";

        } // Fim de metodoDeriv2()

        // Superpôe (overrides)

        // métodos da classe base.

        void met_base1() /*const*/;

        void met_base2() /*const*/;

}; // Fim de class ClasseDeriv.

// Implementações.

void ClasseDeriv::met_base1() /*const*/
```

```
{
        cout << "\nmet_base1() definido na classe derivada...\n";
} // Fim de ClasseDeriv::met_base1()
void ClasseDeriv::met_base2() /*const*/
{
        cout << "\nmet_base2() definido na classe derivada...\n";
} // Fim de ClasseDeriv::met_base2()
int main()
{
        // Cria um objeto
        // de ClasseDeriv.
        ClasseDeriv objDeriv;
        // Chama métodos superpostos.
        objDeriv.met_base1();
        objDeriv.met_base2();
        // Chama métodos da
        // classe derivada.
        objDeriv.metodoDeriv1();
        cout << "Valor de m_propr_deriv = "
                << objDeriv.acessaPropr_deriv()
                << "\n";
```

```
        return 0;

} // Fim de main()

//-------------------------------------------------------
```

Exercício

Utilize a hierarquia classe base `Mamifero`, classe derivada `Cachorro` para ilustrar a superposição de métodos.

Ocultando métodos da classe base

Teoria

Quando superpomos um método na classe derivada, o método de mesmo nome da classe base fica inacessível. Dizemos que o método da classe base fica oculto. Acontece que muitas vezes, a classe base base tem várias versões sobrecarregadas de um método, com um único nome. Se fizermos a superposição de apenas um desses métodos na classe derivada, todas as outras versões da classe base ficarão inacessíveis. O exemplo abaixo ilustra esse fato.

Exemplo

```
//------------------------------------------------

// OculMet.cpp

// Ilustra ocultação

// de métodos da

// classe base.

#include <iostream.h>

class ClasseBase

{

protected:
```

```
        int m_propr_base1;
        int m_propr_base2;
public:
        void met_base() const
        {
                cout << "\nClasseBase::met_base()...\n";
        } // Fim de met_base1()
        void met_base(int vlr) const
        {
                cout << "\nClasseBase::met_base(int)...\n";
                cout << "\nValor = "
                        << vlr
                        << "\n";
        } // Fim de met_base1(int)
}; // Fim de class ClasseBase
class ClasseDeriv : public ClasseBase
{
private:
        int m_propr_deriv;
public:
        // Superpôe (overrides)
```

```
        // método da classe base.
        void met_base() const;
}; // Fim de class ClasseDeriv.
// Implementações.
void ClasseDeriv::met_base() const
{
        cout << "\nClasseDeriv::met_base1()...\n";
} // Fim de ClasseDeriv::met_base1()
int main()
{
        // Cria um objeto
        // de ClasseDeriv.
        ClasseDeriv objDeriv;
        // Chama método superposto.
        objDeriv.met_base();
        // Tenta chamar método
        // da classe base.
        //objDeriv.met_base(10);
        return 0;
} // Fim de main()
```

```
//-------------------------------------------------------------
```

Acessando métodos superpostos da classe base

Teoria

C++ oferece uma sintaxe para acessar métodos da classe base que tenham ficado ocultos pela superposição na classe derivada. Isso é feito com o chamado operador de resolução de escopo, representado por dois caracteres de dois pontos `::`

Assim, digamos que temos uma classe base chamada `ClasseBase`. `ClasseBase` tem um método `met_base()`, que foi superposto na classe derivada `ClasseDeriv`. Assim, `met_base()` da classe base fica inacessível (oculto) na classe derivada `ClasseDeriv`. Para acessá-lo, usamos a notação:

```
ClasseBase::met_base();
```

Por exemplo, se tivermos um objeto de `ClasseDeriv` chamado `objDeriv`, podemos usar a seguinte notação para acessar o método de `ClasseBase`:

```
objDeriv.ClasseBase::met_base(10);
```

Exemplo

```
//-----------------------------------------------------

// AcsOcul.cpp

// Ilustra acesso a
```

```
// métodos ocultos na
// classe base.
#include <iostream.h>
class ClasseBase
{
protected:
        int m_propr_base1;
        int m_propr_base2;
public:
        void met_base() const
        {
                cout << "\nClasseBase::met_base()...\n";
        } // Fim de met_base1()
        void met_base(int vlr) const
        {
                cout << "\nClasseBase::met_base(int)...\n";
                cout << "\nValor = "
                        << vlr
                        << "\n";
        } // Fim de met_base1(int)
}; // Fim de class ClasseBase
```

```
class ClasseDeriv : public ClasseBase

{

private:

        int m_propr_deriv;

public:

        // Superpôe (overrides)

        // método da classe base.

        void met_base() const;

}; // Fim de class ClasseDeriv.

// Implementações.

void ClasseDeriv::met_base() const

{

        cout << "\nClasseDeriv::met_base1()...\n";

} // Fim de ClasseDeriv::met_base1()

int main()

{

        // Cria um objeto

        // de ClasseDeriv.

        ClasseDeriv objDeriv;

        // Chama método superposto.

        objDeriv.met_base();
```

```
        // Tenta chamar método
        // da classe base.
        objDeriv.ClasseBase::met_base(10);
        return 0;
} // Fim de main()
//-----------------------------------------------
```

Métodos virtuais

Teoria

Até agora, temos enfatizado o fato de que uma hierarquia de herança cria um relacionamento do tipo é um. Por exemplo, um objeto da classe `Cachorro` é um `Mamifero`. Isso significa que o objeto da classe `Cachorro` herda os atributos (dados) e as capacidades (métodos) de sua classe base. Porém, em C++, esse tipo de relacionamento vai ainda mais longe.

Através do polimorfismo, C++ permite que ponteiros para a classe base sejam atribuídos a objetos da classe derivada. Portanto, é perfeitamente legal escrever:

```
Mamifero* pMamifero = new Cachorro;
```

Estamos criando um novo objeto da classe `Cachorro` no free store, e atribuindo o ponteiro retornado por `new` a um ponteiro para `Mamifero`. Não há nenhum problema aqui: lembre-se, um `Cachorro` é um `Mamifero`.

Podemos usar esse ponteiro para invocar métodos da classe `Mamifero`. Mas seria também desejável poder fazer com que os métodos superpostos em `Cachorro` chamassem a versão correta da função. Isso é possível com o uso de funções virtuais. Veja o exemplo.

Exemplo

```
//--------------------------------------------------------

// Virt.cpp
```

```
// Ilustra o uso de métodos
// virtuais.
#include <iostream.h>
class Mamifero
{
protected:
        int m_idade;
public:
        // Construtor.
        Mamifero(): m_idade(1)
        {
                cout << "Construtor Mamifero()...\n";
        } // Fim de Mamifero()
        ~Mamifero()
        {
                cout << "Destrutor ~Mamifero()...\n";
        } // Fim de ~Mamifero()
        void andar() const
        {
                cout << "Mamifero anda 1 passo.\n";
        } // Fim de andar()
```

```
        // Um método virtual.
        virtual void emiteSom() const
        {
                cout << "Som de mamifero.\n";
        } // Fim de emiteSom()
}; // Fim de class Mamifero.
class Cachorro : public Mamifero
{
public:
        // Construtor.
        Cachorro() {cout << "Construtor Cachorro()...\n";}
        // Destrutor.
        ~Cachorro() {cout << "Destrutor ~Cachorro()...\n";}
        void abanaCauda() { cout << "Abanando cauda...\n";}
        // Implementa o método virtual.
        void emiteSom() const {cout << "Au! Au! Au!\n";}
        // Implementa outro método.
        void andar() const {cout << "Cachorro anda 5 passos.\n";}
}; // Fim de class Cachorro.
int main()
```

```
{
        // Um ponteiro para Mamifero
        // aponta para um objeto Cachorro.
        Mamifero* pMam = new Cachorro;
        // Chama um método
        // superposto.
        pMam->andar();
        // Chama o método
        // virtual superposto.
        pMam->emiteSom();
        return 0;
} // Fim de main()
//-------------------------------------------------
```

Chamando múltiplas funções virtuais

Teoria

Como funcionam as funções virtuais? Quando um objeto derivado, como o objeto `Cachorro`, é criado, primeiro é chamado o construtor da classe base `Mamifero`; depois é chamado o construtor da própria classe derivada `Cachorro`.

Assim, o objeto `Cachorro` contém em si um objeto da classe base `Mamifero`. As duas partes do objeto `Cachorro` ficam armazenadas em porções contíguas da memória.

Quando uma função virtual é criada em um objeto, o objeto deve manter controle sob essa nova função. Muitos compiladores utilizam uma tabela de funções virtuais, chamada `v-table`. Uma `v-table` é mantida para cada tipo, e cada objeto desse tipo mantém um ponteiro para a `v-table`. Esse ponteiro é chamado `vptr`, ou `v-pointer`).

Assim, o `vptr` de cada objeto aponta para a `v-table` que, por sua vez, tem um ponteiro para cada uma das funções virtuais. Quando a parte `Mamifero` de um objeto `Cachorro` é criada, o `vptr` é inicializado para apontar para a parte certa da `v-table`. Quando o construtor de `Cachorro` é chamado e a parte `Cachorro` do objeto é acrescentada, o `vptr` é ajustado para apontar para as funções virtuais superpostas, se houver, no objeto `Cachorro`.

Exemplo

```
//-------------------------------------------------------

// MulVirt.cpp
```

```
// Ilustra chamada a
// múltiplas versões
// de um método virtual.
#include <iostream.h>
class Mamifero
{
protected:
        int idade;
public:
        // Construtor.
        Mamifero() : idade(1) { }
        // Destrutor.
        ~Mamifero() {}
        // Método virtual.
        virtual void emiteSom() const
        {
                cout << "Som de mamifero.\n";
        } // Fim de emiteSom()
}; // Fim de class Mamifero.
class Cachorro : public Mamifero
{
```

```
public:
        // Implementa método virtual.
        void emiteSom() const {cout << "Au! Au!\n";}
}; // Fim de class Cachorro.
class Gato : public Mamifero
{
public:
        // Implementa método virtual.
        void emiteSom() const {cout << "Miau!\n";}
}; // Fim de class Gato
class Cavalo : public Mamifero
{
public:
        // Implementa método virtual.
        void emiteSom() const {cout << "Relincho!\n";}
}; // Fim de class Cavalo.
class Porco : public Mamifero
{
public:
        // Implementa método virtual.
        void emiteSom() const {cout << "Oinc!\n";}
```

```
}; // Fim de class Porco.

int main()

{

        // Um ponteiro para

        // Mamifero.

        Mamifero* mamPtr;

        int opcao;

        bool flag = true;

        while(flag)

        {

                cout << "\n(1)Cachorro"

                        << "\n(2)Gato"

                        << "\n(3)Cavalo"

                        << "\n(4)Porco"

                        << "\n(5)Mamifero";

                cout << "\nDigite um num. ou "

                        << "zero para sair: ";

                cin >> opcao;

                switch(opcao)

                {

                        case 0:
```

```
            flag = false;
            break;
        case 1:
            mamPtr = new Cachorro;
            mamPtr->emiteSom();
            break;
        case 2:
            mamPtr = new Gato;
            mamPtr->emiteSom();
            break;
        case 3:
            mamPtr = new Cavalo;
            mamPtr->emiteSom();
            break;
        case 4:
            mamPtr = new Porco;
            mamPtr->emiteSom();
            break;
        case 5:
            mamPtr = new Mamifero;
            mamPtr->emiteSom();
```

```
                                        break;

                                default:

                                        cout << "\nOpcao invalida.";

                                        break;

                        } // Fim de switch

                } // Fim de while.

                return 0;

} // Fim de main()

//-------------------------------------------------------
```

Métodos virtuais e passagem por valor

Teoria

Observe que a mágica da função virtual somente opera com ponteiros ou referências. A passagem de um objeto por valor não permite que funções virtuais sejam invocadas. Veja o exemplo abaixo.

Exemplo

```
//--------------------------------------------------------------------------

// VirtVal.cpp

// Ilustra tentativa

// de usar métodos virtuais

// com argumento passado

// por valor.

#include <iostream.h>

class Mamifero

{

protected:

        int idade;

public:
```

```
        // Construtor.
        Mamifero() : idade(1) { }
        // Destrutor.
        ~Mamifero() {}
        // Método virtual.
        virtual void emiteSom() const
        {
                cout << "Som de mamifero.\n";
        } // Fim de emiteSom()
}; // Fim de class Mamifero.
class Cachorro : public Mamifero
{
public:
        // Implementa método virtual.
        void emiteSom() const
        {
                cout << "Au! Au!\n";
        } // Fim de emiteSom()
}; // Fim de class Cachorro.
class Gato : public Mamifero
{
public:
```

```
        // Implementa método virtual.
        void emiteSom() const
        {
                cout << "Miau!\n";
        } // Fim de emiteSom()
}; // Fim de class Gato.
// Protótipos.
void funcaoPorValor(Mamifero);
void funcaoPorPonteiro(Mamifero*);
void funcaoPorRef(Mamifero&);
int main()
{
        Mamifero* mamPtr;
        int opcao;
        cout << "\n(1)Cachorro"
                << "\n(2)Gato"
                << "\n(3)Mamifero"
                << "\n(0)Sair";
        cout << "\n\nEscolha uma opcao: ";
        cin >> opcao;
        cout << "\n";
        switch(opcao)
```

```
{
        case 0:
                mamPtr = 0;
                break;
        case 1:
                mamPtr = new Cachorro;
                break;
        case 2:
                mamPtr = new Gato;
                break;
        case 3:
                mamPtr = new Mamifero;
                break;
        default:
                cout << "\nOpcao invalida.\n";
                mamPtr = 0;
                break;
} // Fim de switch.
// Chama funções.
if(mamPtr)
{
        funcaoPorPonteiro(mamPtr);
```

```
                funcaoPorRef(*mamPtr);
                funcaoPorValor(*mamPtr);
        } // Fim de if(mamPtr)
        return 0;
} // Fim de main()
void funcaoPorValor(Mamifero mamValor)
{
        mamValor.emiteSom();
} // Fim de funcaoPorValor()
void funcaoPorPonteiro(Mamifero* pMam)
{
        pMam->emiteSom();
} // Fim de funcaoPorPonteiro()
void funcaoPorRef(Mamifero& refMam)
{
        refMam.emiteSom();
} // Fim de funcaoPorRef()
//-----------------------------------------------------
```

Construtor de cópia virtual

Teoria

Métodos construtores não podem ser virtuais. Contudo, há ocasiões em que surge uma necessidade de poder passar um ponteiro para um objeto base e fazer com que uma cópia do objeto derivado correto seja criado. Uma solução comum para esse problema é criar um método chamado `clone()` na classe base e torná-lo virtual. O método `clone()` cria uma cópia do novo objeto da classe atual, e retorna esse objeto.

Como cada classe derivada superpõe o método `clone()`, uma cópia do objeto correto é criada.

Exemplo

```
//---------------------------------------------------------

// VirtCop.cpp

// Ilustra uso do

// método clone()

// como substituto para

// um construtor de cópia

// virtual.

#include <iostream.h>
```

```
class Mamifero
{
public:
        Mamifero() : idade(1)
        {
                cout << "Construtor de Mamifero...\n";
        } // Fim de Mamifero()
        ~Mamifero()
        {
                cout << "Destrutor de Mamifero...\n";
        } // Fim de ~Mamifero()
        // Construtor de cópia.
        Mamifero(const Mamifero& refMam);
        // Métodos virtuais.
        virtual void emiteSom() const
        {
                cout << "Som de mamifero.\n";
        } // Fim de emiteSom()
        virtual Mamifero* clone()
        {
```

```
                return new Mamifero(*this);
        } // Fim de clone()
        int acessaIdade() const
        {
                return idade;
        } // Fim de acessaIdade()
protected:
        int idade;
}; // Fim de class Mamifero.
// Construtor de cópia.
Mamifero::Mamifero(const Mamifero& refMam) :
                idade(refMam.acessaIdade())
{
        cout << "Construtor Mamifero(Mamifero&)...\n";
} // Fim de Mamifero::Mamifero(const Mamifero&)
class Cachorro : public Mamifero
{
public:
        Cachorro()
        {
                cout << "Construtor Cachorro()...\n";
```

```
        } // Fim de Cachorro()
        ~Cachorro()
        {
                cout << "Destrutor ~Cachorro()...\n";
        } // Fim de ~Cachorro()
        // Construtor de cópia.
        Cachorro(const Cachorro& refCach);
        // Implementa métodos virtuais.
        void emiteSom() const
        {
                cout << "Au!Au!\n";
        } // Fim de emiteSom()
        virtual Mamifero* clone()
        {
                return new Cachorro(*this);
        } // Fim de clone()
}; // Fim de class Cachorro.
// Construtor de cópia.
Cachorro::Cachorro(const Cachorro& refCach) :
                Mamifero(refCach)
{
```

```
        cout << "Construtor Cachorro(Cachorro&)...\n";
} // Fim de Cachorro::Cachorro(Cachorro&)
class Gato : public Mamifero
{
public:
        Gato()
        {
                cout << "Construtor Gato()...\n";
        } // Fim de Gato()
        ~Gato()
        {
                cout << "Destrutor ~Gato()...\n";
        } // Fim de ~Gato()
        // Construtor de cópia.
        Gato(const Gato& refGato);
        // Implementa métodos virtuais.
        void emiteSom() const
        {
                cout << "Miau!\n";
        } // Fim de emiteSom()
        virtual Mamifero* clone()
```

```
        {
                return new Gato(*this);
        } // Fim de clone()
}; // Fim de class Gato.
// Construtor de cópia.
Gato::Gato(const Gato& refGato) :
                Mamifero(refGato)
{
        cout << "Construtor Gato(const Gato&)...\n";
} // Fim de Gato::Gato(const Gato&)
enum ANIMAIS {MAMIFERO, CACHORRO, GATO};
int main()
{
        // Um ponteiro para
        // Mamifero.
        Mamifero* mamPtr;
        int opcao;
        // Exibe menu.
        cout <<"\n(1)Cachorro"
                << "\n(2)Gato"
                << "\n(3)Mamifero\n";
```

```
        cout << "\nDigite a opcao: ";
cin >> opcao;
switch(opcao)
{
        case CACHORRO:
                mamPtr = new Cachorro;
                break;
        case GATO:
                mamPtr = new Gato;
                break;
        default:
                mamPtr = new Mamifero;
                break;
} // Fim de switch.
// Um outro ponteiro
// para Mamifero.
Mamifero* mamPtr2;
cout << "\n*** Som do original ***\n";
// Emite som.
mamPtr->emiteSom();
// Cria clone.
```

```
        mamPtr2 = mamPtr->clone();

        cout << "\n*** Som do clone ***\n";

        mamPtr2->emiteSom();

        return 0;

} // Fim de main()

//-------------------------------------------------------
```