C++ Básico

Autor:
Luis Antonio

C++
Básico

ALÔ MUNDO!

Uma tradição seguida em muitos cursos de programação é iniciar o aprendizado de uma nova linguagem com o programa "Alo, Mundo!". Trata-se de um programa elementar, que simplesmente exibe esta mensagem na tela.

Mesmo sendo um programa muito simples, AloMundo.cpp serve para ilustrar diversos aspectos importantes da linguagem.

AloMundo.cpp ilustra também a estrutura básica de um programa C++.

Por enquanto, não se preocupe em entender o significado de todas as linhas do programa AloMundo.cpp. Simplesmente digite o programa exatamente como está mostrado na listagem abaixo.

Para digitar um programa C++, você deve utilizar um editor de texto que possa salvar arquivos em formato texto puro. Porém, ao invés de salvar os programas com a extensão .TXT no nome, salve-os com a extensão .CPP. Geralmente, os editores de texto que acompanham ambientes de desenvolvimento C++ são os melhores para esse tipo de trabalho. Porém nada impede que você use um editor de texto de uso geral, como o Bloco de Notas do Windows.

Os exemplos deste curso devem ser compilados na linha de comando. Para isso, você precisa abrir um prompt, como a janela do DOS no Windows.

Digite então o comando de compilação de seu compilador. Todos os exemplos deste curso foram testados no Borland C++Builder 3.0. O compilador de linha de comando do C++Builder é chamado com o seguinte comando:

bcc32 AloMundo.cpp

Observe que em cada caso, é preciso substituir o nome do arquivo a ser compilado pelo nome correto. No caso acima, estaremos compilando o programa AloMundo.cpp.

Este comando faz com que seja gerado um arquivo executável, no caso AloMundo.exe.

Para executar o programa AloMundo.exe, basta digitar na linha de comando:

AloMundo

e pressionar a tecla <Enter>. O programa será executado.

Ao longo deste curso, você começará a entender as diferentes partes de um programa C++, e tudo passará a fazer sentido.

Exemplo

```
// AloMundo.cpp
// Um programa elementar.
#include <iostream.h>
int main()
{
        cout << "Alo, Mundo!\n";
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

Alo, Mundo!

Exercício

Modifique o programa AloMundo.cpp, de maneira que ele exiba na tela a frase

Alo, Brasil!

Exercício

Modifique o programa AloMundo.cpp, de maneira que ele exiba na tela as frases

Alo, Brasil!

Estamos aqui!

Cada frase deve aparecer em uma linha.

APRESENTANDO COUT

Em outro ponto deste curso, veremos com detalhes como usar cout para exibir dados na tela. Por enquanto, podemos usar cout, mesmo sem entender por completo seu funcionamento. Para exibir um valor na tela, escreva a palavra cout, seguida pelo operador de inserção <<, que é criado digitando-se duas vezes o caractere menor do que <. Observe que embora o operador << seja composto de dois caracteres, para a linguagem C++ ele representa um único operador.

Depois do operador de inserção << colocamos os dados que queremos exibir. O exemplo abaixo ilustra o uso do fluxo de saída cout.

Exemplo

```
// AprCout.cpp
// Apresenta o uso
// de cout.
#include <iostream.h>
int main()
{
        cout << "Alo, Mundo!\n";
        cout << "Eis um numero: " << 42 << "\n";
        cout << "Um numero grande: "
                << 280830583058 << "\n";
        cout << "Eis uma soma: "
                << "Soma de 245 + 432 = "
                << 245 + 432
                << "\n";
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

Alo, Mundo!

Eis um numero: 42

Um numero grande: 280830583058

Eis uma soma: Soma de 245 + 432 = 677

Exercício

Modifique o programa AprCout.cpp de maneira que ele exiba na tela o resultado de uma subtração, o produto de uma multiplicação e o nome do programador.

COMENTÁRIOS

Quando escrevemos um programa, é importante inserir comentários que esclareçam o que fazem as diversas partes do código. Isso é particularmente necessário em linguagem C++.

C++ tem dois estilos de comentários: o comentário de barra dupla // e o comentário barra asterisco /*

O comentário de barra dupla //, também conhecido como comentário no estilo C++, diz ao compilador para ignorar tudo que se segue ao comentário, até o final da linha.

O comentário /*, também conhecido como comentário em estilo C, diz ao compilador para ignorar tudo que se segue ao par de caracteres /*, até que seja encontrado um par de caracteres de fechamento */

Exemplo

```
// Coment.cpp
// Ilustra o uso
// de comentários.
#include <iostream.h>
int main()
{
        /* Este comentário
        se estende se estende
        por várias linhas, até
        que o par de caracteres
        de fechamento seja
        encontrado */
        cout << "Alo, Mundo!\n";
        // Comentários neste estilo
        // vão somente até o
        // final da linha.
        cout << "Alo, Brasil!\n";
} // Fim de main()
```

Saída gerada por este programa:

```
Alo, Mundo!

Alo, Brasil!
```

Exercício

Acrescente um comentário inicial ao programa Coment.cpp. O comentário inicial deve conter o nome do programa, o nome do arquivo fonte, os nomes das funções contidas no programa, uma descrição do que faz o programa, o nome do autor, dados sobre o ambiente de desenvolvimento e compilação, dados de versão e observações adicionais.

TIPOS DE DADOS

A linguagem C++ tem cinco tipos básicos de dados, que são especificados pelas palavras-chave:

char
int
float
double
bool

O tipo char (caractere) é usado para texto. O tipo int é usado para valores inteiros. Os tipos float e double expressam valores de ponto flutuante (fracionários). O tipo bool expressa os valores verdadeiro (true) e falso (false).

É importante ressaltar que embora C++ disponha do tipo bool, qualquer valor diferente de zero é interpretado como sendo verdadeiro (true). O valor zero é interpretado como sendo falso (false).

O exemplo abaixo cria variáveis dos tipos básicos e exibe seus valores.

Exemplo

```
// Tipos.cpp
// Ilustra os tipos
// básicos de C++.
#include <iostream.h>
int main()
{
        // Declara e
        // inicializa uma
        // variável char.
        char cVar = 't';
        // Declara e
        // inicializa uma
        // variável int.
        int iVar = 298;
        // Declara e
        // inicializa uma
        // variável float.
        float fVar = 49.95;
        // Declara e
        // inicializa uma
        // variável double.
        double dVar = 99.9999;
        // Declara e
        // inicializa uma
        // variável bool.
        bool bVar = (2 > 3); // False.
        // O mesmo que:
        // bool bVar = false;
        // Exibe valores.
        cout << "cVar = "
                << cVar << "\n";
        cout << "iVar = "
                << iVar << "\n";
        cout << "fVar = "
                << fVar << "\n";
        cout << "dVar = "
                << dVar << "\n";
```

```
        cout << "bVar = "
                << bVar << "\n";
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

cVar = t

iVar = 298

fVar = 49.95

dVar = 99.9999

bVar = 0

Exercício

No programa Tipos.cpp, modifique os valores de cada uma das variáveis, após a declaração. Faça com que os novos valores sejam exibidos na tela.

O TIPO CHAR

O tipo char (caractere) geralmente tem o tamanho de um byte, o que é suficiente para conter 256 valores.

Observe que um char pode ser interpretado de duas maneiras:

- Como um número pequeno (0 a 255)

- Como um elemento de um conjunto de caracteres, como ASCII.

É importante ter em mente que, na realidade, computadores não entendem letras nem caracteres de pontuação. Computadores somente entendem números. O que o conjunto ASCII faz na realidade é associar um número a cada caractere, para que o computador possa trabalhar com esses caracteres. Por exemplo, a letra 'a' é associada ao número 97; a letra 'b' é associada ao número 98, e assim por diante.

Exemplo

```
// TChar.cpp
// Ilustra o uso
// do tipo char.
#include <iostream.h>
int main()
{
        // Exibe o alfabeto
        // minúsculo.
        for(char ch = 97; ch <= 122; ch++)
        cout << ch << " ";
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

a b c d e f g h i j k l m n o p q r s t u v w x y z

Exercício

Modifique o programa TChar.cpp, de maneira que o alfabeto seja exibido na tela em MAIÚSCULAS e minúsculas, da seguinte forma:

MAIUSCULAS

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

minusculas

a b c d e f g h i j k l m n o p q r s t u v w x y z

SEQUÊNCIAS DE ESCAPE

Alguns caracteres podem ser representados por combinações especiais de outros caracteres. Essas combinações são conhecidas como seqüências de escape, porque "escapam" do significado normal do caractere. Por exemplo, o caractere 't' representa, obviamente a letra t minúscula. Já a combinação '\t' representa o caractere de tabulação (a tecla tab). Já usamos em vários exemplos a combinação '\n', que representa um caractere de nova linha.

A tabela abaixo representa algumas seqüências de escape mais comuns:

Seqüência de escape	O que representa
\n	caractere de nova linha
\t	caractere de tabulação (tab)
\b	caractere backspace
\"	aspa dupla
\'	aspa simples
\?	ponto de interrogação
\\	barra invertida

Exemplo

```
// Escape.cpp
// Ilustra o uso
// de seqüências
// de escape.
#include <iostream.h>
int main()
{
        // Exibe frases
        // usando seqüências de
```

```
        // escape.
        cout << "\"Frase entre aspas\"\n";
        cout << "Alguma duvida\?\n";
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

"Frase entre aspas"

Alguma duvida?

Exercício

Escreva um programa que exiba na tela as letras do alfabeto maiúsculo, separadas por tabulações.

VARIÁVEIS

Podemos pensar na memória do computador como sendo uma coleção enorme de pequenas gavetas. Cada uma dessas gavetinhas é numerada seqüencialmente e representa um byte.

Esse número seqüencial é conhecido como endereço de memória. Uma variável reserva uma ou mais gavetinhas para armazenar um determinado valor.

O nome da variável é um rótulo que se refere a uma das gavetinhas. Isso facilita a localização e o uso dos endereços de memória. Uma variável pode começar em um determinado endereço e estender-se por várias gavetinhas, ou vários bytes, subseqüentes.

Quando definimos uma variável em C++, precisamos informar ao compilador o tipo da variável: um número inteiro, um número de ponto flutuante, um caractere, e assim por diante. Essa informação diz ao compilador quanto espaço deve ser reservado para a variável, e o tipo de valor que será armazenado nela.

Dissemos que cada gavetinha corresponde a um byte. Se a variável for de um tipo que ocupa dois bytes, precisaremos de dois bytes de memória, ou duas gavetinhas. Portanto, é o tipo da variável (por exemplo, int) que informa ao compilador quanta memória deve ser reservada para ela.

Em um determinado tipo de computador/sistema operacional, cada tipo de variável ocupa um número de bytes definido e invariável. Ou seja, uma variável int pode ocupar dois bytes em um tipo de máquina (por exemplo, no MS-DOS), quatro bytes em outro tipo de máquina (por exemplo, no Windows 95), e assim por diante.

C++ oferece um operador, chamado sizeof, que nos permite determinar o tamanho em bytes de um tipo de dados ou de uma variável.

Exemplo

```
// TamVar.cpp
// Ilustra o tamanho
// das variáveis.
#include <iostream.h>
int main()
{
        cout << "*** Tamanhos das variaveis ***\n";
        cout << "Tamanho de int = "
                << sizeof(int)
```

```
                << " bytes.\n";
        cout << "Tamanho de short int = "
                << sizeof(short)
                << " bytes.\n";
        cout << "Tamanho de bool = "
                << sizeof(bool)
                << " bytes.\n";
        cout << "Tamanho de char = "
                << sizeof(char)
                << " bytes.\n";
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

*** Tamanhos das variaveis ***

Tamanho de int = 4 bytes.

Tamanho de short int = 2 bytes.

Tamanho de bool = 1 bytes.

Tamanho de char = 1 bytes.

Exercício

Modifique o programa TamVar.cpp, de maneira que ele exiba na tela o tamanho em bytes das seguintes variáveis: long, float e double.

ATRIBUINDO VALORES ÀS VARIÁVEIS

Para criar uma variável, precisamos declarar o seu tipo, seguido pelo nome da variável e por um caractere de ponto e vírgula ;

int larg;

Para atribuir um valor a uma variável, usamos o operador de atribuição =

larg = 7;

Opcionalmente, podemos combinar esses dois passos, declarando e inicializando a variável em uma só linha:

int larg = 7;

Mais tarde, quando tratarmos das constantes, veremos que alguns valores devem obrigatoriamente ser inicializados no momento da declaração, porque não podemos atribuir-lhes valores posteriormente.

Podemos também definir mais de uma variável em uma só linha. Podemos ainda misturar declarações simples com inicializações.

// Declara duas variáveis,

// inicializa uma.

int larg = 7, compr;

Exemplo

```
// AtriVal.cpp
// Ilustra a atribuição
// de valores a variáveis.
#include <iostream.h>
int main()
{
        // Declara duas variáveis,
        // inicializa uma.
        int larg = 7, compr;
        // Atribui valor.
        compr = 8;
        // Declara e inicializa
        // mais uma variável.
        int area = larg * compr;
        // Exibe valores.
        cout << "*** Valores finais ***\n";
        cout << "Largura = "
                << larg << "\n";
        cout << "Comprimento = "
                << compr << "\n";
        cout << "Area = "
                << area << "\n";
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

*** Valores finais ***

Largura = 7

Comprimento = 8

Area = 56

Exercício

Modifique o exemplo AtriVal.cpp, de maneira que ele calcule o volume de uma caixa retangular. Para isso, acrescente uma variável para representar a profundidade.

VARIÁVEIS UNSIGNED

Em C++, os tipos inteiros existem em duas variedades: signed (com sinal) e unsigned (sem sinal). A idéia é que, às vezes é necessário poder trabalhar com valores negativos e positivos; outras vezes, os valores são somente positivos. Os tipos inteiros (short, int e long), quando não são precedidos pela palavra unsigned sempre podem assumir valores negativos ou positivos. Os valores unsigned são sempre positivos ou iguais a zero.

Como o mesmo número de bytes é utilizado para os inteiros signed e unsigned, o maior número que pode ser armazenado em um inteiro unsigned é o dobro do maior número positivo que pode ser armazenado em um inteiro signed.

A tabela abaixo ilustra os valores de uma implementação típica de C++:

Tipo	Tamanho (em bytes)	Valores
unsigned short int	2	0 a 65.535
short int	2	-32.768 a 32.767
unsigned long int	4	0 a 4.294.967.295
long int	4	-2.147.483.648 a 2.147.483.647
int (16 bits)	2	-32.768 a 32.767
int (32 bits)	4	-2.147.483.648 a 2.147.483.647
unsigned int (16 bits)	2	0 a 65.535
unsigned int (32 bits)	4	0 a 4.294.967.295
char	1	256 valores de caracteres
float	4	1,2e-38 a 3,4e38
double	8	2,2e-308 a 1,8e308

Exemplo

```
// TamUns.cpp
// Ilustra o tamanho
// das variáveis unsigned.
#include <iostream.h>
int main()
{
        cout << "*** Tamanhos das variaveis ***\n";
        cout << "Tamanho de unsigned int = "
                << sizeof(unsigned int)
                << " bytes.\n";
        cout << "Tamanho de unsigned short int = "
                << sizeof(unsigned short)
                << " bytes.\n";
        cout << "Tamanho de unsigned char = "
                << sizeof(unsigned char)
                << " bytes.\n";
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

```
*** Tamanhos das variaveis ***
```

Tamanho de unsigned int = 4 bytes.

Tamanho de unsigned short int = 2 bytes.

Tamanho de unsigned char = 1 bytes.

Exercício

Modifique o programa TamUns.cpp, de maneira que ele exiba o tamanho de uma variável unsigned long.

ESTOURANDO UMA VARIÁVEL UNSIGNED

O que acontece quando tentamos armazenar em uma variável um valor fora da faixa de valores que essa variável pode conter? Isso depende da variável ser signed ou unsigned.

Quando uma variável unsigned int chega a seu valor máximo, ela volta para zero, de forma similar ao que acontece com o marcador de quilometragem de um automóvel quando todos os dígitos indicam 9.

Exemplo

```
// EstShrt.cpp
// Ilustra "estouro"
// de uma variável
// unsigned short.
#include <iostream.h>
int main()
{
        unsigned short int usVar;
        usVar = 65535;
        cout << "Valor inicial = "
                << usVar << "\n";
        // Soma 1.
        usVar = usVar + 1;
        cout << "Somando 1 = "
                << usVar << "\n";
        // Soma mais 1.
        usVar = usVar + 1;
        cout << "Somando mais 1 = "
                << usVar << "\n";
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

Valor inicial = 65535

Somando 1 = 0

Somando mais 1 = 1

Exercício

Modifique o exemplo EstShrt.cpp de maneira que o valor estourado seja do tipo unsigned int.

ESTOURANDO UM AVRIÁVEL SIGNED

Uma variável signed é diferente de uma variável unsigned, porque metade de seus valores são reservados para representar valores negativos. Assim, quando chegamos ao maior valor positivo, o "marcador de quilometragem" da variável signed não pula para zero, e sim para o maior valor negativo.

Exemplo

```
// EstSShrt.cpp
// Ilustra "estouro"
// de uma variável
// signed short.
#include <iostream.h>
int main()
{
        short int sVar;
        sVar = 32767;
        cout << "Valor inicial = "
                << sVar << "\n";
        // Soma 1.
        sVar = sVar + 1;
        cout << "Somando 1 = "
                << sVar << "\n";
        // Soma mais 1.
        sVar = sVar + 1;
        cout << "Somando mais 1 = "
                << sVar << "\n";
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

Valor inicial = 32767

Somando 1 = -32768

Somando mais 1 = -32767

Exercício

Reescreva o exemplo EstSShrt.cpp, de maneira que a variável estourada seja do tipo signed int.

O TIPO STRING

Uma das atividades mais comuns em qualquer tipo de programa é a manipulação de strings de texto. Uma string de texto pode ser uma palavra, uma frase ou um texto mais longo, como uma série de frases.

Por isso, a biblioteca padrão C++ oferece um tipo, chamado string, que permite realizar diversas operações úteis com strings de texto.

O exemplo abaixo é uma reescrita do programa elementar AloMundo.cpp, usando o tipo string.

Exemplo

```
// AloStr.cpp
// Ilustra o uso
// do tipo string.
#include <iostream.h>
int main()
{
        // Declara e inicializa
        // uma variável do
        // tipo string.
        string aloTar = "Alo, Tarcisio!";
        // Exibe a string.
        cout << aloTar;
} // Fim de main()
```

Saída gerada por este programa:

Alo, Tarcisio!

Exercício

Modifique o programa AloStr.cpp, de maneira que a saída mostrada na tela deixe uma linha vazia antes e outra linha vazia depois da frase Alo, Tarcisio!.

CONCATENANDO STRINGS

O tipo string permite o uso do operador + para concatenar (somar) strings. O exemplo abaixo mostra como isso é feito.

Exemplo

```
// SomaStr.cpp
// Ilustra o uso
// do operador +
// com o tipo string.
#include <iostream.h>
int main()
{
        // Declara e inicializa
        // algumas variáveis do
        // tipo string.
        string s1 = "Agua mole ";
        string s2 = "em pedra dura ";
        string s3 = "tanto bate ";
        string s4 = "ate que fura";
        // Exibe usando
        // o operador +
        cout << s1 + s2 +
                s3 + s4 + "!!!\n\n";
        // Acrescenta exclamações
        // e salta duas linhas
        // no final.
} // Fim de main()
```

Saída gerada por este programa:

Agua mole em pedra dura tanto bate ate que fura!!!

Exercício

C++ permite o uso de operadores combinados, como +=, *=, -= e \=, para simplificar a escrita de operações como:

a = a + b;

// Pode ser escrito como:

a += b;

Escreva um programa que utilize o operador += com strings. Faça com que o resultado exibido na tela evidencie esse uso.

FUNÇÕES

As funções representam um dos blocos construtivos da linguagem C++. Outro bloco construtivo básico de C++ são as classes de objetos, que veremos no futuro.

Todo programa C++ tem obrigatoriamente pelo menos uma função, chamada main(). Todo comando executável em C++ aparece dentro de alguma função. Dito de forma simples, uma função é um grupo de comandos que executa uma tarefa específica, e muitas vezes retorna (envia) um valor para o comando que a chamou.

As funções em C++ são o equivalente às procedures e functions do Pascal, ou aos procedimentos SUB e FUNCTION do Basic. São as funções que possibilitam a escrita de programas bem organizados.

Em um programa bem escrito, cada função desempenha uma tarefa bem definida.

O exemplo abaixo, AloFunc.cpp contém duas funções: main() e digaAlo(). A seção principal de execução de todo programa C++ é representada pela função main(), que marca onde começa e onde termina a execução. Ou seja, todo programa C++ tem uma e somente uma função main().

A execução de AloFunc.cpp (e de qualquer outro programa C++) começa no começo da função main() e termina quando a função main() é encerrada.

Exemplo

```
// AloFunc.cpp
// Ilustra uma
// função elementar.
#include <iostream.h>
// Definição da função
// digaAlo()
void digaAlo()
{
        cout << "\nAlo, Mundo!";
} // Fim de digaAlo()
int main()
{
        // Chama a função
```

```
        // digaAlo()
        digaAlo();
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

Alo, Mundo!

Exercício

Acrescente ao exemplo AloFunc.cpp uma segunda função, chamada digaTchau(). A função digaTchau() deve exibir na tela a mensagem Tchau!. Faça com que a função digaAlo() chame a função digaTchau(), de que maneira que o programa produza na tela a seguinte saída:

Alo, Mundo!

Tchau!

CHAMANDO UMA FUNÇÃO

Embora main() seja uma função, ela é diferente das outras. A função main() é sempre chamada para iniciar a execução de um programa. As outras funções são chamadas ao longo da execução do programa.

Começando no início de main(), o programa é executado linha por linha, na ordem em que elas aparecem no código. Porém quando a execução chega a uma chamada a função, algo diferente acontece. O programa pula para o código correspondente àquela função. Terminada a execução da função, o programa é retomado na linha que se segue imediatamente à chamada à função.

É como se você estivesse lendo um livro, e encontrasse uma palavra desconhecida. Você suspenderia a leitura e consultaria um dicionário. Após descobrir o significado da nova palavra, você retomaria então a leitura do livro, no ponto em que havia parado.

Exemplo

```
// ChamFun.cpp
// Ilustra chamada
// a uma função.
#include <iostream.h>
// Definição da função.
void UmaFuncao()
{
        cout << "...agora, estamos em UmaFuncao()...\n";
} // Fim de UmaFuncao()
int main()
{
        cout << "Estamos em main()...\n";
        // Chama UmaFuncao();
        UmaFuncao();
        cout << "...e voltamos a main().\n";
} // Fim de main()
```

Saída gerada por este programa:

Estamos em main()...

...agora, estamos em UmaFuncao()...

...e voltamos a main().

Exercício

Modifique o programa ChamFun.cpp, definindo uma segunda função, chamada OutraFuncao(). Faça com que a primeira função, UmaFuncao(), chame a segunda função, OutraFuncao(). A saída mostrada na tela deve evidenciar essas chamadas.

UMA FUNÇÃO COM PARÂMETROS

A definição de uma função consiste de um cabeçalho e de um corpo. O cabeçalho contém o tipo retornado, o nome da função e os parâmetros que ela recebe. Os parâmetros de uma função permitem que passemos valores para a função. Assim, se uma função deve somar dois números, esses números seriam os parâmetros da função. Eis um exemplo de cabeçalho de função:

int Soma(int i, int j)

Um parâmetro é uma declaração de qual o tipo de valor que será passado para a função. O valor passado de fato é chamado de argumento.

O corpo da função consiste de uma chave de abertura {, seguida pelos comandos que executam a tarefa da função, e finalmente, pelo chave de fechamento }.

Exemplo

```
// FunSimp.cpp
// Ilustra o uso
// de uma função simples.
#include <iostream.h>
int Soma(int i, int j)
{
        cout << "Estamos na funcao Soma().\n";
        cout << "Valores recebidos: \n";
        cout << "i = "
                << i
                << ", j = "
                << j
                << "\n";
        return (i + j);
} // Fim de Soma(int, int)
int main()
{
        cout << "Estamos em main()\n";
        int x, y, z;
        cout << "\nDigite o primeiro num. + <Enter>";
        cin >> x;
        cout << "\nDigite o segundo num. + <Enter>";
        cin >> y;
```

```
        cout << "Chamando funcao Soma()...\n";
        z = Soma(x, y);
        cout << "Voltamos a main()\n";
        cout << "Novo valor de z = "
                << z
                << "\n";
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

Estamos em main()

Digite o primeiro num. + <Enter>48

Digite o segundo num. + <Enter>94

Chamando funcao Soma()...

Estamos na funcao Soma().

Valores recebidos:

i = 48, j = 94

Voltamos a main()

Novo valor de z = 142

Exercício

Modifique o programa FunSimp.cpp, criando uma função chamada Multiplic(), no lugar da função Soma(). A função Multiplic() deve multiplicar dois números inteiros, e retornar um valor inteiro. Os números a serem multiplicados devem ser solicitados do usuário, e o resultado da multiplicação deve ser exibido na tela.

UMA FUNÇÃO MEMBRO DE STRING

Vimos que a biblioteca padrão de C++ contém o tipo string, usado na manipulação de strings de texto. Na verdade, esse tipo é implementado como uma classe de objetos, um conceito fundamental em C++. Embora ainda não tenhamos estudado os objetos em C++, podemos usá-los de forma mais ou menos intuitiva, para ter uma idéia do poder e da praticidade que representam.

Por exemplo, os fluxos de entrada e saída cin e cout, que já usamos, são objetos de C++. O tipo string também é um objeto.

Objetos contêm operações que facilitam sua manipulação. Essas operações são similares a funções que ficam contidas no objeto, por isso são chamadas de funções membro. Para chamar uma função membro de um objeto, usamos o operador ponto . Por exemplo, a linha abaixo chama a função membro substr() de um objeto da classe string, para acessar uma substring contida nesta string.

sobreNome = nome.substr(9, 5);

O exemplo abaixo mostra como isso é feito.

Exemplo

```
// MaiStr.cpp
// Ilustra outras
// funções de strings.
#include <iostream.h>
int main()
{
        // Declara e inicializa
        // uma variável do
        // tipo string.
        string nome = "Tarcisio Lopes";
        // Exibe.
        cout << "Meu nome = "
                << nome
                << "\n";
        // Declara outra string.
        string sobreNome;
        // Acessa substring que
        // começa na posição 9
        // e tem comprimento 5.
        sobreNome = nome.substr(9, 5);
        // Exibe sobrenome.
        cout << "Meu sobrenome = "
                << sobreNome
                << "\n";
} // Fim de main()
```

Saída gerada por este programa:

Meu nome = Tarcisio Lopes

Meu sobrenome = Lopes

Exercício

Reescreva o exemplo MaiStr.cpp, utilizando a função membro substr() para acessar seu próprio nome e sobrenome.

OUTRA FUNÇÃO MEMBRO DE STRING

Dissemos que os objetos têm funções membros que facilitam sua manipulação.

Outra operação comum com strings é substituir parte de uma string. Como se trata de uma operação com strings, nada mais lógico que esta operação esteja contida nos objetos da classe string. Esta operação é feita com uma função membro de string chamada replace(). O exemplo abaixo mostra como ela pode ser utilizada.

Exemplo

```
// ReplStr.cpp
// Ilustra outras
// funções de strings.
#include <iostream.h>
int main()
{
        // Declara e inicializa
        // uma variável do
        // tipo string.
        string nome = "Tarcisio Lopes";
        // Exibe.
        cout << "Meu nome = "
                << nome
                << "\n";
        // Utiliza a função membro
        // replace() para
        // substituir parte
        // da string.
        // A parte substituída
        // começa em 0 e
        // tem o comprimento 8
        nome.replace(0, 8, "Mateus");
        // Exibe nova string.
        cout << "Nome do meu filho = "
                << nome
                << "\n";
} // Fim de main()
```

Saída gerada por este programa:

Meu nome = Tarcisio Lopes

Nome do meu filho = Mateus Lopes

Exercício

Reescreva o exemplo ReplStr.cpp utilizando seu próprio nome e o nome de alguém de sua família.

USANDO TYPEDEF

Às vezes, o processo de declaração de variáveis pode se tornar tedioso, repetitivo e sujeito a erros. Isso acontece, por exemplo, se usamos muitas variáveis do tipo unsigned short int em um programa. C++ permite criar um novo nome para esse tipo, com o uso da palavra-chave typedef.

Na verdade, com typedef estamos criando um sinônimo para um tipo já existente. Não estamos criando um novo tipo. Isso será visto em outro ponto deste curso.

Eis a forma de uso de typedef:

typedef unsigned short int USHORT;

A partir daí, podemos usar USHORT, ao invés de unsigned short int.

Exemplo

```
// typedef.cpp
// Ilustra o uso
// de typedef.
#include <iostream.h>
// Cria um sinônimo usando typedef.
typedef unsigned short int USHORT;
int main()
{
        // Declara duas variáveis,
        // inicializa uma.
        USHORT larg = 7, compr;
        // Atribui valor.
        compr = 8;
        // Declara e inicializa
        // mais uma variável.
        USHORT area = larg * compr;
        // Exibe valores.
        cout << "*** Valores finais ***\n";
        cout << "Largura = "
                << larg << "\n";
        cout << "Comprimento = "
                << compr << "\n";
        cout << "Area = "
                << area << "\n";
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

*** Valores finais ***

Largura = 7

Comprimento = 8

Area = 56

Exercício

Modifique o exemplo typedef.cpp, de maneira a criar um sinônimo para o tipo unsigned long.

CONSTANTES COM #DEFINE

Muitas vezes, é conveniente criar um nome para um valor constante. Este nome é chamado de constante simbólica.

A forma mais tradicional de definir constantes simbólicas é usando a diretiva de preprocessador #define:

#define PI 3.1416

Observe que neste caso, PI não é declarado como sendo de nenhum tipo em particular (float, double ou qualquer outro). A diretiva #define faz simplesmente uma substituição de texto. Todas as vezes que o preprocessador encontra a palavra PI, ele a substitui pelo texto 3.1416.

Como o preprocessador roda antes do compilador, o compilador nunca chega a encontrar a constante PI; o que ele encontra é o valor 3.1416.

Exemplo

```
// DefTst.cpp
// Ilustra o uso
// de #define.
#include <iostream.h>
// Para mudar a precisão,
// basta alterar #define.
#define PI 3.1416
//#define PI 3.141593
int main()
{
        cout << "Area do circulo "
                << "de raio 5 = "
                << PI * 5 * 5
                << "\n";
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

Area do circulo de raio 5 = 78.54

Exercício

A distância percorrida pela luz em um ano, conhecida como ano-luz, pode ser calculada pela seguinte fórmula:

anoLuz = KM_POR_SEGUNDO * SEGUNDOS_POR_MINUTO * MINUTOS_POR_HORA * HORAS_POR_DIA * DIAS_POR_ANO;

Utilize #define de maneira que a fórmula acima possa ser usada diretamente em um programa C++. Dica: velocidade da luz = 300.000 Km/s.

CONSTANTES COM CONST

Embora a diretiva #define funcione, C++ oferece uma forma melhor de definir constantes simbólicas: usando a palavra-chave const.

const float PI = 3.1416;

Este exemplo também declara uma constante simbólica chamada PI, mas desta vez o tipo de PI é declarado como sendo float. Este método tem diversas vantagens. Além de tornar o código mais fácil de ler e manter, ele dificulta a introdução de bugs.

A principal diferença é que esta constante tem um tipo, de modo que o compilador pode checar se a constante está sendo usada de acordo com seu tipo.

Exemplo

```
// CstTst.cpp
// Ilustra o uso
// de const.
#include <iostream.h>
// Com const, a constante
// tem um tipo definido
// (neste caso, float)
const float PI = 3.1416;
//const float PI = 3.141593;
int main()
{
        cout << "Area do circulo "
                << "de raio 5 = "
                << PI * 5 * 5
                << "\n";
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

Area do circulo de raio 5 = 78.54

Exercício

Utilize a palavra-chave const para calcular a distância percorrida pela luz em um ano, conhecida como ano-luz, com a seguinte fórmula:

anoLuz = KM_POR_SEGUNDO * SEGUNDOS_POR_MINUTO * MINUTOS_POR_HORA * HORAS_POR_DIA * DIAS_POR_ANO;

Dica: velocidade da luz = 300.000 Km/s.

CONSTANTES ENUMERADAS

Vimos que uma constante simbólica é um nome que usamos para representar um valor. Vimos também que em C++, podemos definir uma constante simbólica de duas maneiras:

Usando #define

#define PI 3.1416

Usando a palavra chave const

const float PI = 3.1416;

Esta segunda forma é a mais recomendada na maioria dos casos.

Podemos também definir coleções de constantes, chamadas constantes enumeradas, usando a palavra-chave enum. As constantes enumeradas permitem criar novos tipos e depois definir variáveis desses tipos. Os valores

assumidos ficam restritos a uma determinada coleção de valores. Por exemplo, podemos declarar uma enumeração para representar os dias da semana:

```
enum DiasDaSemana

{

Segunda,

Terca,

Quarta,

Quinta,

Sexta,

Sabado,

Domingo

}; // Fim de enum DiasDaSemana.
```

Depois disso, podemos definir variáveis do tipo DiasDaSemana, que somente podem assumir os valores Segunda = 0, Terca = 1, Quarta = 2, e assim por diante.

Assim, cada constante enumerada tem um valor inteiro. Se não especificarmos esse valor, a primeira constante assumirá o valor 0, a segunda constante assumirá o valor 1, e assim por diante. Se necessário, podemos atribuir um valor determinado a uma dada constante. Se somente uma constante for inicializada, as constantes subseqüentes assumirão valores com incremento de 1, a partir daquela que foi inicializada. Por exemplo:

```
enum DiasDaSemana

{

Segunda = 100,

Terca,

Quarta,

Quinta,

Sexta = 200,

Sabado,

Domingo

}; // Fim de enum DiasDaSemana.
```

Na declaração acima, as constantes não inicializadas assumirão os seguintes valores:

Terca = 101, Quarta = 102, Quinta = 103

Sabado = 201, Domingo = 202

As constantes enumeradas são representadas internamente como sendo do tipo int.

Exemplo

```
// Enum.cpp
// Ilustra o uso
// de enumerações.
#include <iostream.h>
int main()
{
        // Define uma enumeração.
        enum DiasDaSemana
        {
                Segunda,
                Terca,
                Quarta,
                Quinta,
                Sexta,
                Sabado,
                Domingo
        }; // Fim de enum DiasDaSemana.
        // O mesmo que:
        // const int Segunda = 0;
        // const int Terca = 1;
        // Etc...
        // const int Domingo = 6;
        // Declara uma variável do tipo
        // enum DiasDaSemana.
        DiasDaSemana dias;
        // Uma variável int.
        int i;
        cout << "Digite um num. (0 a 6) + <Enter>:\n";
        cin >> i;
        dias = DiasDaSemana(i);
        if((dias == Sabado) || (dias == Domingo))
                cout << "Voce escolheu o fim de semana.\n";
        else
                cout << "Voce escolheu um dia util.\n";
return 0;
} // Fim de main()
```

Saída gerada por este programa:

Digite um num. (0 a 6) + <Enter>:

5

Voce escolheu o fim de semana.

Exercício

Escreva um programa que declare e utilize uma enumeração chamada Horas, de maneira que a constante UmaHora tenha o valor 1, a constante DuasHoras tenha o valor 2, e assim por diante, até que a constante DozeHoras tenha o valor 12.

EXPRESSÕES

Em C++, uma expressão é qualquer comando (statement) que após ser efetuado gera um valor. Outra forma de dizer isso é: uma expressão sempre retorna um valor.

Uma expressão pode ser simples:

3.14 // Retorna o valor 3.14

Ou mais complicada:

x = a + b * c / 10;

Observe que a expressão acima retorna o valor que está sendo atribuído a x. Por isso, a expressão inteira pode ser atribuída a outra variável. Por exemplo:

y = x = a + b * c / 10;

Exemplo

```
// Expres.cpp
// Ilustra o uso
// de expressões.
#include <iostream.h>
int main()
{
        int a = 0, b = 0, c = 0, d = 20;
        cout << "*** Valores iniciais ***\n";
        cout << "a = " << a
                << ", b = " << b
                << ", c = " << c
                << ", d = " << d
                << "\n";
        // Atribui novos valores.
        a = 12;
        b = 15;
        // Avalia expressão.
        c = d = a + b;
        // Exibe novos valores.
        cout << "*** Novos valores ***\n";
        cout << "a = " << a
                << ", b = " << b
                << ", c = " << c
                << ", d = " << d
                << "\n";
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

*** Valores iniciais ***

a = 0, b = 0, c = 0, d = 20

*** Novos valores ***

a = 12, b = 15, c = 27, d = 27

Exercício

Modifique o exemplo Expres.cpp de maneira que os novos valores das variáveis a e b sejam solicitados do usuário.

OPERADORES MATEMÁTICOS

Existem cinco operadores matemáticos em C++:

+ adição

- subtração

* multiplicação

/ divisão

% módulo

Os quatro primeiros operadores, funcionam da forma que seria de se esperar, com base na matemática elementar. O operador módulo % fornece como resultado o resto de uma divisão inteira. Por exemplo, quando fazemos a divisão inteira 31 por 5, o resultado é 6, e o resto é 1. (Lembre-se, inteiros não podem ter parte fracionária.) Para achar o resto da divisão inteira, usamos o operador módulo %. Assim, 31 % 5 é igual a 1.

Exemplo

```
// Resto.cpp
// Ilustra o uso
// do operador
// módulo.
#include <iostream.h>
int main()
{
        cout << "*** Resto da divisao inteira ***\n";
        cout << "40 % 4 = "
                << 40 % 4
                << "\n";
        cout << "41 % 4 = "
                << 41 % 4
                << "\n";
        cout << "42 % 4 = "
                << 42 % 4
                << "\n";
```

```
        cout << "43 % 4 = "
                << 43 % 4
                << "\n";
        cout << "44 % 4 = "
                << 44 % 4
                << "\n";
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

*** Resto da divisao inteira ***

40 % 4 = 0

41 % 4 = 1

42 % 4 = 2

43 % 4 = 3

44 % 4 = 0

Exercício

Modifique o programa Resto.cpp, de maneira que sejam exibidos os restos da divisão inteira por 5 de cada um dos números entre 40 e 45, inclusive.

SUBTRAÇÃO COM UNSIGNED

Vimos que uma variável unsigned somente pode assumir valores não-negativos. O que acontece quando tentamos armazenar um valor negativo em uma variável unsigned? Isso pode acontecer como resultado de uma subtração, conforme ilustrado abaixo.

Exemplo

```
// EstDif.cpp
// Ilustra estouro
// de uma variável unsigned
// como resultado de uma
// operação matemática.
#include <iostream.h>
int main()
{
        unsigned int diferenca;
        unsigned int numMaior = 1000;
        unsigned int numMenor = 300;
        cout << "\nnumMaior = "
                << numMaior
                << ", numMenor = "
                << numMenor
                << "\n";
```

```
        diferenca = numMaior - numMenor;
        cout << "\nnumMaior - numMenor = "
                << diferenca
                << "\n";
        diferenca = numMenor - numMaior;
        cout << "\nnumMenor - numMaior = "
                << diferenca
                << "\n";
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

numMaior = 1000, numMenor = 300

numMaior - numMenor = 700

numMenor - numMaior = 4294966596

Exercício

Modifique o exemplo EstDif.cpp, de maneira que a operação de subtração não cause o estouro da variável.

OPERADORES EM PREFIXO E SUFIXO

Dois operadores muito importantes e úteis em C++ são o operador de incremento ++ e o operador de decremento --

O operador de incremento aumenta em 1 o valor da variável à qual é aplicado; o operador de decremento diminui 1.

A posição dos operadores ++ e -- em relação a variável (prefixo ou sufixo) é muito importante. Na posição de prefixo, o operador é aplicado primeiro, depois o valor da variável é acessado. Na posição de sufixo, o valor da variável é acessado primeiro, depois o operador é aplicado.

Exemplo

```
// PreSuf.cpp
// Ilustra o uso de
// operadores em prefixo
// e sufixo.
#include <iostream.h>
int main()
{
        int i = 10, j = 10;
        cout << "\n*** Valores iniciais ***\n";
        cout << "i = " << i
                << ", j = " << j;
        // Aplica operadores.
        i++;
```

```
        ++j;
        cout << "\n*** Apos operadores ***\n";
        cout << "i = " << i
               << ", j = " << j;
        cout << "\n*** Exibindo usando operadores ***\n";
        cout << "i = " << i++
               << ", j = " << ++j;
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

*** Valores iniciais ***

i = 10, j = 10

*** Apos operadores ***

i = 11, j = 11

*** Exibindo usando operadores ***

i = 11, j = 12

Exercício

Modifique o exemplo PreSuf.cpp. Faça com que operadores em sufixo e prefixo sejam aplicados às duas variáveis, i e j. Depois inverta a ordem da aplicação dos operadores.

O COMANDO IF

O fluxo de execução de um programa faz com que as linhas sejam executadas na ordem em que aparecem no código. Entretanto, é muito comum que um programa precise dar saltos em sua execução, em resposta a determinadas condições. O comando if permite testar uma condição (por exemplo, se duas variáveis são iguais) e seguir para uma parte diferente do código, dependendo do resultado desse teste.

A forma mais simples do comando if é:

if(expressão)

comando;

A expressão entre parênteses pode ser de qualquer tipo. O mais comum é que seja uma expressão relacional. Se o valor da expressão for zero, ela é considerada falsa, e o comando não é executado. Se o valor da expressão for diferente de zero, ela é considerada verdadeira, e o comando é executado.

No exemplo:

if(a > b)

a = b;

somente se a for maior que b, a segunda linha será executada.

Um bloco de comandos contidos entre chaves { } tem efeito similar ao de um único comando. Portanto, o comando if pode ser também utilizado da seguinte forma:

if(expressao)

{

comando1;

comando2;

// etc.

}

Exemplo

```
// DemoIf.cpp
// Ilustra o uso
// do comando if.
#include <iostream.h>
int main()
{
        int golsBrasil, golsHolanda;
        cout << "\n*** Placar Brasil X Holanda ***\n";
        cout << "Digite gols do Brasil: ";
        cin >> golsBrasil;
        cout << "\nDigite gols da Holanda: ";
        cin >> golsHolanda;
        if(golsBrasil > golsHolanda)
                cout << "A festa e' verde e amarela!!!\n";
        if(golsHolanda > golsBrasil)
                cout << "A festa e' holandesa!!!\n";
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

*** Placar Brasil X Holanda ***

Digite gols do Brasil: 5

Digite gols da Holanda: 3

A festa e' verde e amarela!!!

Exercício

Modifique o exemplo DemoIf.cpp para levar em consideração a possibilidade de empate.

O COMANDO ELSE

Muitas vezes, um programa precisa seguir um caminho de execução se uma dada condição for verdadeira, e outro caminho de execução se a mesma condição for falsa. Para isto, C++ oferece a combinação if... else. Eis a forma genérica:

if(expressao)

comando1;

else

comando2;

Exemplo

```
// DemElse.cpp
// Ilustra o uso
// de else.
#include <iostream.h>
int main()
{
        int numMaior, numMenor;
        cout << "Digite numMaior + <Enter>: ";
        cin >> numMaior;
        cout << "Digite numMenor + <Enter>: ";
        cin >> numMenor;
        if(numMaior > numMenor)
                cout << "\nOk. numMaior e' maior "
                        "que numMenor.\n";
        else
                cout << "Erro!!! numMaior e' menor "
                        "ou igual a numMenor!\n";
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

Digite numMaior + <Enter>: 10

Digite numMenor + <Enter>: 18

Erro!!! numMaior e' menor ou igual a numMenor!

Exercício

Modifique o exemplo DemElse.cpp de maneira que o programa cheque inicialmente se os dois números são diferentes, e exiba uma mensagem se eles forem iguais.

MAIS SOBRE IF / ELSE

Qualquer comando pode aparecer dentro da cláusula if... else. Isso inclui até mesmo outra cláusula if... else.

O exemplo abaixo ilustra esse fato.

Exemplo

```
// IfElse.cpp
// Outro exemplo
// de if/else.
#include <iostream.h>
int main()
{
        int numMaior, numMenor;
        cout << "Digite numMaior + <Enter>: ";
        cin >> numMaior;
        cout << "Digite numMenor + <Enter>: ";
                cin >> numMenor;
        if(numMaior >= numMenor)
        {
                if((numMaior % numMenor) == 0)
                {
                        if(numMaior == numMenor)
                                cout << "numMaior e' igual a numMenor.\n";
                        else
                                cout << "numMaior e' multiplo "
                                        "de numMenor\n";
                } // Fim de if((numMaior % numMenor) == 0)
                else
                        cout << "A divisao nao e' exata.\n";
        } // Fim de if(numMaior >= numMenor)
        else
                cout << "Erro!!! numMenor e' maior "
                        "que numMaior!\n";
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

Digite numMaior + <Enter>: 44

Digite numMenor + <Enter>: 39

A divisao nao e' exata.

Exercício

Reescreva o programa IfElse.cpp, de maneira que o programa cheque primeiro se os dois números são iguais e utilize a construção else if para checar se numMaior é maior que numMenor.

IDENTAÇÃO

As cláusulas if...else podem ser aninhadas indefinidamente. Isso quer dizer que uma cláusula if...else pode conter outra cláusula if...else, que pode conter uma terceira cláusula if...else, a qual pode conter uma quarta cláusula if...else, e assim por diante.

Esse tipo de código pode se tornar difícil de ler, e induzir a erros. Por isso é importante usar adequadamente o recurso da indentação do código e as chaves { }.

A indentação consiste em indicar níveis de aninhamento, afastando gradualmente os blocos de código da margem esquerda da página. Isso geralmente é feito com o uso da tecla <Tab>.

Exemplo

```
// Indent.cpp
// ATENÇÃO: ESTE PROGRAMA
// CONTÉM ERROS PROPOSITAIS!!!
// Ilustra a importância da
// indentação e do uso
// de chaves.
#include <iostream.h>
int main()
{
cout << "\nDigite um num. menor que 5 "
"ou maior que 10: ";
int num;
cin >> num;
if(num >= 5)
if(num > 10)
cout << "\nVoce digitou maior que 10.\n";
else
cout << "\nVoce digitou menor que 5.\n";
// Erro no casamento if/else.
return 0;
} // Fim de main()
```

Saída gerada por este programa:

Digite um num. menor que 5 ou maior que 10: 7

Voce digitou menor que 5.

Exercício

Modifique o programa Indent.cpp, de maneira que ele apresente o comportamento correto. Utiliza a indentação para facilitar a leitura do código fonte.

OPERADORES LÓGICOS

Muitas vezes, pode surgir a necessidade de fazer mais de uma pergunta relacional de uma só vez. Por exemplo,

"x é maior que y e, ao mesmo tempo, y é maior que z?"

Pode ser necessário determinar se essas duas condições são verdadeiras ao mesmo tempo, dentro de um determinado programa.

Os operadores lógicos mostrados abaixo são usados nesse tipo de situação.

Operador	**Símbolo**	***Exemplo***
AND	&&	expressao1 && expressao2
OR	||	expressao1 || expressao2
NOT	!	!expressao

O operador lógico AND avalia duas expressões. Se as duas forem verdadeiras, o resultado da operação lógica AND será verdadeiro. Ou seja, é preciso que ambos os lados da operação seja verdadeira, para que a expressão completa seja verdadeira.

O operador lógico OR avalia duas expressões. Se qualquer uma delas for verdadeira, o resultado da operação lógica OR será verdadeiro. Ou seja, basta que um dos lados da operação seja verdadeiro, para que a expressão completa seja verdadeira.

O operador lógico NOT avalia uma só expressão. O resultado é verdadeiro se a expressão avaliada for falsa, e vice versa.

Exemplo

```
// AndTst.cpp
// Ilustra o uso
// do operador lógico
// AND.
#include <iostream.h>
int main()
{
        int a;
        cout << "\nDigite um num. positivo "
                "e menor que 10: ";
        cin >> a;
        if((a > 0) && (a < 10))
                cout << "\nVoce digitou corretamente...";
        cout << "\nDigite um num. negativo "
                "ou maior que 1000: ";
        cin >> a;
        if((a < 0) || (a > 1000))
                cout << "\nVoce acertou de novo...";
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

Digite um num. positivo e menor que 10: 12

Digite um num. negativo ou maior que 1000: 1001

Voce acertou de novo...

Exercício

Modifique o exemplo AndTst.cpp, usando o operador NOT ! para exibir mensagens caso o usuário digite valores errados.

O OPERADOR CONDICIONAL TERNÁRIO

O operador condicional ? : é o único operador ternário de C++. Ou seja, ele recebe três termos.

O operador condicional recebe três expressões e retorna um valor.

(expressao1) ? (expressao2) : (expressao3);

Esta operação pode ser interpretada da seguinte forma: se expressao1 for verdadeira, retorne o valor de expressao2; caso contrario, retorne o valor de expressao3.

Exemplo

```
// OpTern.cpp
// Ilustra o uso do
// operador condicional
// ternário.
#include <iostream.h>
int main()
{
        int a, b, c;
        cout << "Digite um num. + <Enter>: ";
        cin >> a;
        cout << "\nDigite outro num. + <Enter>: ";
        cin >> b;
        if(a == b)
                cout << "Os numeros sao iguais. "
                        "Tente novamente.\n";
        else
        {
                // Atribui o valor
                // mais alto à
                // variável c.
                c = (a > b) ? a : b;
                // Exibe os valores.
                cout << "\n*** Valores finais ***\n";
                cout << "a = " << a << "\n";
                cout << "b = " << b << "\n";
                cout << "c = " << c << "\n";
        } // Fim de else.
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

Digite um num. + <Enter>: 99

Digite outro num. + <Enter>: 98

*** Valores finais ***

a = 99

b = 98

c = 99

Exercício

Reescreva a atribuição de OpTern.cpp usando if...else.

PROTÓTIPOS DE FUNÇÕES

O protótipo de uma função é uma declaração que indica o tipo que a função retorna, o nome da função e os parâmetros que recebe. Eis um exemplo de protótipo de função:

int CalcArea(int compr, int larg);

O protótipo e a definição da função devem conter exatamente o mesmo tipo retornado, o mesmo nome e a mesma lista de parâmetros. Se houver alguma discordância, isso gerará um erro de compilação. Porém o protótipo da função não precisa conter os nomes dos parâmetros, somente seus tipos. Por exemplo, o protótipo acima poderia ser reescrito da seguinte forma:

int CalcArea(int, int);

Este protótipo declara uma função chamada CalcArea(), que retorna um valor int e recebe dois parâmetros, também int.

Todas as funções retornam um tipo, ou void. Se o tipo retornado não for especificado explicitamente, fica entendido que o tipo é int.

Muitas das funções que usamos em nossos programas já existem como parte da biblioteca padrão que acompanha o compilador C++. Para usar uma dessas funções, é necessário incluir no programa o arquivo que contém o protótipo da função desejada, usando a diretiva #include. Para as funções que nós mesmos escrevemos, precisamos escrever o protótipo.

Exemplo

```
// IntrFun.cpp
// Introduz o uso
// de funções.
#include <iostream.h>
// Protótipo.
int CalcArea(int compr, int larg);
int main()
{
        int comp, lrg, area;
        cout << "*** Calculo da area"
                " de um retangulo ***\n";
        cout << "Digite o comprimento "
                "(metros) + <Enter>: ";
        cin >> comp;
        cout << "\nDigite a largura "
```

```
                "(metros) + <Enter>: ";
        cin >> lrg;
        // Calcula area usando
        // a funcao CalcArea()
        area = CalcArea(comp, lrg);
        cout << "\nArea = "
                << area
                << " metros quadrados.\n";
        return 0;
} // Fim de main()
// Definição da função.
int CalcArea(int compr, int larg)
{
        return compr * larg;
} // Fim de CalcArea()
```

Saída gerada por este programa:

*** Calculo da area de um retangulo ***

Digite o comprimento (metros) + <Enter>: 12

Digite a largura (metros) + <Enter>: 15

Area = 180 metros quadrados.

Exercício

Reescreva o programa IntrFun.cpp. No lugar da função CalcArea(), crie uma função CalcVolume(), que calcula o volume de uma caixa retangular.

FUNÇÕES: VARIÁVEIS LOCAIS

Além de podermos passar variáveis para uma função, na forma de argumentos, podemos também declarar variáveis dentro do corpo da função. Essas variáveis são chamadas locais, porque somente existem localmente, dentro da função. Quando a função retorna, a variável deixa de existir.

As variáveis locais são definidas da mesma forma que as outras variáveis. Os parâmetros da função são também considerados variáveis locais, e podem ser usados exatamente como se tivessem sido definidos dentro do corpo da função.

Exemplo

```
// Local.cpp
// Ilustra o uso de
// variáveis locais.
#include <iostream.h>
// Protótipo.
// Converte temperatura em graus
// Fahrenheit para graus centígrados.
double FahrParaCent(double);
int main()
{
        double tempFahr, tempCent;
        cout << "\n*** Conversao de graus Fahrenheit "
```

```
                "para graus Centigrados ***\n";
        cout << "Digite a temperatura em Fahrenheit: ";
        cin >> tempFahr;
        tempCent = FahrParaCent(tempFahr);
        cout << "\n"
                << tempFahr
                << " graus Fahrenheit = "
                << tempCent
                << " graus Centigrados.\n";
        return 0;
} // Fim de main()
// Definição da função.
double FahrParaCent(double fahr)
{
        // Variável local.
        double cent;
        cent = ((fahr - 32) * 5) / 9;
        return cent;
} // Fim de FahrParaCent(double fahr)
```

Saída gerada por este programa:

*** Conversao de graus Fahrenheit para graus Centigrados ***

Digite a temperatura em Fahrenheit: 65

65 graus Fahrenheit = 18.3333 graus Centigrados.

Exercício

Reescreva o programa Local.cpp, de maneira que a função faça a conversão de graus centígrados para graus Fahrenheit.

FUNÇÕES: VARIÁVEIS GLOBAIS

Além de podermos passar variáveis para uma função, na forma de argumentos, podemos também declarar variáveis dentro do corpo da função. Essas variáveis são chamadas locais, porque somente existem localmente, dentro da função. Quando a função retorna, a variável deixa de existir.

As variáveis locais são definidas da mesma forma que as outras variáveis. Os parâmetros da função são também considerados variáveis locais, e podem ser usados exatamente como se tivessem sido definidos dentro do corpo da função.

Exemplo

```
// Local.cpp
// Ilustra o uso de
// variáveis locais.
#include <iostream.h>
// Protótipo.
// Converte temperatura em graus
// Fahrenheit para graus centígrados.
double FahrParaCent(double);
```

```
int main()
{
        double tempFahr, tempCent;
        cout << "\n*** Conversao de graus Fahrenheit "
                "para graus Centigrados ***\n";
        cout << "Digite a temperatura em Fahrenheit: ";
        cin >> tempFahr;
        tempCent = FahrParaCent(tempFahr);
        cout << "\n"
                << tempFahr
                << " graus Fahrenheit = "
                << tempCent
                << " graus Centigrados.\n";
        return 0;
} // Fim de main()
// Definição da função.
double FahrParaCent(double fahr)
{
        // Variável local.
        double cent;
        cent = ((fahr - 32) * 5) / 9;
        return cent;
} // Fim de FahrParaCent(double fahr)
```

Saída gerada por este programa:

*** Conversao de graus Fahrenheit para graus Centigrados ***

Digite a temperatura em Fahrenheit: 65

65 graus Fahrenheit = 18.3333 graus Centigrados.

Exercício

Reescreva o programa Local.cpp, de maneira que a função faça a conversão de graus centígrados para graus Fahrenheit.

VARIÁVEL DENTRO DE UM BLOCO

Dissemos que uma variável local declarada dentro de uma função tem escopo local. Isso quer dizer que essa variável é visível e pode ser usada somente dentro da função na qual foi declarada.

C++ permite também definir variáveis dentro de qualquer bloco de código, delimitado por chaves { e }. Uma variável declarada dentro de um bloco de código fica disponível somente dentro desse bloco.

Exemplo

```
// VarBloc.cpp
// Ilustra variável
// local dentro de
// um bloco.
#include <iostream.h>
// Protótipo.
void UmaFuncao();
```

```
int main()
{
        // Variável local em main().
        int var1 = 10;
        cout << "\nEstamos em main()\n";
        cout << "var1 = " << var1 << "\n";
        // Chama função.
        UmaFuncao();
        cout << "\nDe volta a main()\n";
        cout << "var1 = " << var1 << "\n";
        return 0;
} // Fim de main()
// Definição da função.
void UmaFuncao()
{
        // Variável local na função.
        int var1 = 20;
        cout << "\nEstamos em UmaFuncao()\n";
        cout << "var1 = " << var1 << "\n";
        // Define um bloco de código.
        {
                // Variável local no bloco
                // de código.
                int var1 = 30;
                cout << "\nEstamos dentro do "
                "bloco de codigo.\n";
                cout << "var1 = " << var1 << "\n";
        } // Fim do bloco de código.
        cout << "\nEstamos em UmaFuncao(), "
                "fora do bloco de codigo.\n";
        cout << "var1 = " << var1 << "\n";
} // Fim de UmaFuncao()
```

Saída gerada por este programa:

Estamos em main()

var1 = 10

Estamos em UmaFuncao()

var1 = 20

Estamos dentro do bloco de codigo.

var1 = 30

Estamos em UmaFuncao(), fora do bloco de codigo.

var1 = 20

De volta a main()

var1 = 10

Exercício

Reescreva o programa VarBloc.cpp declarando uma variável global. Faça com que o valor da variável global seja alterado dentro de main(), dentro da função UmaFuncao() e dentro do bloco de código.

FUNÇÕES: PARÂMETROS COMO VARIÁVEIS LOCAIS

Como dissemos, os parâmetros de uma função são variáveis locais à função. Isso significa que alterações feitas nos argumentos recebidos não afetam os valores originais desses argumentos. Isso é conhecido como passagem por valor. O que acontece é que a função cria cópias locais dos argumentos recebidos, e opera sobre essas cópias. Tais cópias locais são tratadas exatamente como as outras variáveis locais.

Qualquer expressão válida em C++ pode ser usada como argumento, inclusive constantes, expressões matemáticas e lógicas e outras funções que retornem um valor do tipo correto.

Lembre-se também que os parâmetros de uma função não precisam ser todos do mesmo tipo. Por exemplo, podemos escrever uma função que receba como argumentos um int, um float e um double.

Exemplo

```
// Param.cpp
// Ilustra os parâmetros
// de uma função como
// variáveis locais.
#include <iostream.h>
// Protótipo.
void troca(int, int);
int main()
{
        // Declara variáveis
        // locais em main()
        int var1 = 10, var2 = 20;
        cout << "Estamos em main(), antes de troca()\n";
        cout << "var1 = " << var1 << "\n";
        cout << "var2 = " << var2 << "\n";
        // Chama a função.
        troca(var1, var2);
        cout << "Estamos em main(), depois de troca()\n";
        cout << "var1 = " << var1 << "\n";
        cout << "var2 = " << var2 << "\n";
        return 0;
} // Fim de main()
// Definição da função.
void troca(int var1, int var2)
{
        // Exibe os valores.
        cout << "Estamos em troca(), antes da troca\n";
        cout << "var1 = " << var1 << "\n";
        cout << "var2 = " << var2 << "\n";
        // Efetua a troca.
        int temp;
        temp = var1;
        var1 = var2;
        var2 = temp;
```

```
        // Exibe os valores.
        cout << "Estamos em troca(), depois da troca\n";
        cout << "var1 = " << var1 << "\n";
        cout << "var2 = " << var2 << "\n";
} // Fim de troca(int, int)
```

Saída gerada por este programa:

Estamos em main(), antes de troca()

var1 = 10

var2 = 20

Estamos em troca(), antes da troca

var1 = 10

var2 = 20

Estamos em troca(), depois da troca

var1 = 20

var2 = 10

Estamos em main(), depois de troca()

var1 = 10

var2 = 20

Exercício

Reescreva o programa Param.cpp, substituindo a função troca() por uma função chamada vezes10(), que multiplica por 10 os argumentos recebidos.

RETORNANDO VALORES

Em C++, todas as funções, ou retornam um valor, ou retornam void. A palavra-chave void é uma indicação de que nenhum valor será retornado.

Para retornar um valor de uma função, escrevemos a palavra-chave return, seguida do valor a ser retornado. O valor pode ser também uma expressão que retorne um valor. Eis alguns exemplos válidos de uso de return:

return 10;

return (a > b);

return(funcaoArea(larg, comp));

É claro que, no caso de uso de uma expressão, o valor retornado pela expressão deve ser compatível com o valor a ser retornado por return. O mesmo vale para uma função.

Quando a palavra-chave return é encontrada, a expressão que se segue a return é retornada como sendo o valor da função. A execução do programa volta imediatamente para a função chamadora, e quaisquer comandos que venham depois da palavra-chave return não serão executados.

Podemos ter diversos comandos return dentro de uma função. Porém, cada vez que a função for chamada, somente um dos return será executado.

Exemplo

```
// RetVal.cpp
// Ilustra o uso de
// return para retornar um
// valor de uma função.
#include <iostream.h>
// Protótipo.
int Dobra(int x);
// Esta função retorna
// o dobro do valor que
// recebe como parâmetro.
int main()
{
        int resultado = 0;
        int vlrInicial;
        cout << "\nDigite um num. entre 0 e 5000 + <Enter>: ";
        cin >> vlrInicial;
        cout << "Estamos em main(), "
                "antes da chamada a Dobra()\n";
        cout << "Valor inicial = "
                << vlrInicial << "\n";
        cout << "Resultado = "
                << resultado << "\n";
        // Chama a função Dobra().
        resultado = Dobra(vlrInicial);
        cout << "Estamos em main(), "
                "depois da chamada a Dobra()\n";
        cout << "Valor inicial = "
                << vlrInicial << "\n";
        cout << "Resultado = "
                << resultado << "\n";
        return 0;
} // Fim de main()
// Definição da função.
int Dobra(int x)
// Esta função retorna
// o dobro do valor que
// recebe como parâmetro.
{
        if(x <= 5000)
                return x * 2;
        else
        {
                cout << "Valor invalido.\n";
                return -1; // Indica erro.
        } // Fim de else.
        // Em muitos compiladores,
        // este código causará um
        // warning.
```

```
        cout << "Este codigo nao sera' executado.\n";
} // Fim de Dobra(int)
```

Saída gerada por este programa:

Digite um num. entre 0 e 5000 + <Enter>: 4999

Estamos em main(), antes da chamada a Dobra()

Valor inicial = 4999

Resultado = 0

Estamos em main(), depois da chamada a Dobra()

Valor inicial = 4999

Resultado = 9998

Exercício

Reescreva o programa RetVal.cpp, substituindo a função Dobra() pela função int ValorAbs(int x). Esta função deve retornar o valor absoluto, sempre positivo, do valor recebido como argumento.

VALORES DEFAULT

Para cada parâmetro que declaramos no protótipo e na definição de uma função, precisamos fornecer um argumento correspondente na chamada a essa função. Os valores passados para a função devem ser dos mesmos tipos que aparecem no protótipo. Ou seja, se tivermos uma função cujo protótipo seja:

int funcaoX(long);

cada vez que chamarmos funcaoX() devemos fornecer um argumento long. Se houver qualquer incoerência nos tipos, o compilador emitirá uma mensagem de erro.

Porém há uma exceção a essa regra: O protótipo da função pode declarar um valor default para um ou mais parâmetros. Neste caso, se na chamada à função não for fornecido um valor, o valor default será utilizado. Eis o protótipo acima, reescrito com um valor default:

int funcaoX(long lVal = 100000);

O que este protótipo diz é o seguinte: "a função funcaoX() retorna um int e recebe um parâmetro long. Se não for fornecido um argumento, utilize o valor 100000". Como não é obrigatório o uso de nomes de parâmetros nos protótipos de funções, o protótipo acima poderia também ser escrito assim:

int funcaoX(long = 100000);

Mesmo com o uso de um valor default, a definição da função permanece inalterada. Ou seja, na definição de funcaoX(), o cabeçalho seria escrito assim:

int funcaoX(long lVal)

Assim, se a chamada à função funcaoX() não contiver um argumento, o compilador usará automaticamente o valor 100000. O nome do parâmetro default que aparece no protótipo nem sequer precisa ser o mesmo nome usado no cabeçalho da função; o importante aqui é a posição do valor default, não o nome.

Uma função pode ter um, alguns ou todos os parâmetros com valores default. A única restrição é a seguinte: se um parâmetro não tiver valor default, nenhum parâmetro antes dele pode ter. Ou seja, parâmetros com valores default devem ser os últimos na lista de parâmetros de uma função.

Exemplo

```
// VlrDef.cpp
// Ilustra o uso de
// valores default.
#include <iostream.h>
// Protótipo.
int Volume(int compr, int larg = 10, int profund = 12);
// Calcula o volume de um paralelepípedo
// com as dimensões dadas.
// Oferece valores default para
// largura e profundidade.
int main()
{
        int comp, lg, pr;
        int vol;
        cout << "\nDigite comprimento: ";
        cin >> comp;
        // Utiliza valores default
        // para largura e profundidade.
        vol = Volume(comp);
        cout << "Volume = "
                << vol
                << "\n";

        cout << "\nDigite comprimento: ";
        cin >> comp;
        cout << "\nDigite largura: ";
        cin >> lg;
        // Utiliza valor default
        // somente para profundidade.
        vol = Volume(comp, lg);
        cout << "Volume = "
                << vol
                << "\n";
        cout << "\nDigite comprimento: ";
        cin >> comp;
        cout << "\nDigite largura: ";
        cin >> lg;
        cout << "\nDigite profundidade: ";
        cin >> pr;
        // Não utiliza nenhum
        // valor default.
        vol = Volume(comp, lg, pr);
        cout << "Volume = "
                << vol
                << "\n";
        return 0;
```

```
} // Fim de main()
// Definição da função.
int Volume(int compr, int larg, int profund)
// Calcula o volume de um paralelepípedo
// com as dimensões dadas.
{
        return compr * larg * profund;
} // Fim de Volume()
```

Saída gerada por este programa:

Digite comprimento: 40

Volume = 4800

Digite comprimento: 40

Digite largura: 60

Volume = 28800

Digite comprimento: 40

Digite largura: 60

Digite profundidade: 80

Volume = 192000

Exercício

Reescreva o programa VlrDef.cpp, definindo uma função chamada VolArea() da seguinte maneira: se VolArea() receber três argumentos, ela calcula o volume de uma caixa retangular; se VolArea() receber dois argumentos, ela calcula a área de um retângulo.

SOBRECARGA DE FUNÇÕES

Em C++, podemos ter mais de uma função com o mesmo nome. Esse processo chama-se sobrecarga de funções. As funções de mesmo nome devem diferir na lista de parâmetros, seja por diferenças nos tipos ou no número de parâmetros, ou em ambos. Exemplo:

int funcaoX(int, int);

int funcaoX(long, long);

int funcaoX(long);

Observe que a lista de parâmetros das três versões de funcaoX() são diferentes. A primeira e a segunda versões diferem nos tipos dos parâmetros. A terceira versão difere no número de parâmetros.

O tipo retornado pode ou não ser o mesmo nas várias versões sobrecarregadas. Importante: se tentarmos sobrecarregar uma função modificando somente o tipo retornado, isso gerará um erro de compilação.

A versão correta da função será chamada pelo compilador, com base nos argumentos recebidos. Por exemplo, podemos criar uma função chamada media(), que calcula a média entre dois números int, long, float ou double. A versão correta será chamada, conforme os argumentos recebidos sejam do tipo int, long, float ou double.

Exemplo

```
// Sobrec.cpp
// Ilustra sobrecarga
// de funções
#include <iostream.h>
// Protótipos.
int Dobra(int);
float Dobra(float);
int main()
{
        int intValor, intResult;
        float floatValor, floatResult;
        cout << "\nDigite um valor inteiro + <Enter>: ";
        cin >> intValor;
        cout << "\nChamando int Dobra(int)...";
        intResult = Dobra(intValor);
        cout << "\nO dobro de "
                << intValor
                << " = "
                << intResult
                << "\n";
        cout << "\nDigite um valor fracionario + <Enter>: ";
        cin >> floatValor;
        cout << "\nChamando float Dobra(float)...";
        floatResult = Dobra(floatValor);
        cout << "\nO dobro de "
                << floatValor
                << " = "
                << floatResult
                << "\n";
        return 0;
} // Fim de main()
// Definições.
int Dobra(int iVal)
{
        return 2 * iVal;
} // Fim de Dobra(int iVal)
float Dobra(float fVal)
{
        return 2 * fVal;
} // Fim de Dobra(float)
```

Saída gerada por este programa:

Digite um valor inteiro + <Enter>: 30

Chamando int Dobra(int)...

O dobro de 30 = 60

Digite um valor fracionario + <Enter>: 30.99

Chamando float Dobra(float)...

O dobro de 30.99 = 61.98

Exercício

Escreva um programa contendo uma função sobrecarregada chamada Divide(). Se esta função for chamada com dois argumentos float, ela deve retornar o resultado da divisão de um argumento pelo outro. Se a função Divide() for chamada com um único argumento float, deve retornar o resultado da divisão deste argumento por 10.

FUNÇÕES INLINE

Quando criamos uma função, normalmente o compilador cria um conjunto de instruções na memória. Quando chamamos a função, a execução do programa pula para aquele conjunto de instruções, e quando a função retorna, a execução volta para a linha seguinte àquela em que foi feita a chamada à função. Se a função for chamada 10 vezes, o programa salta para o conjunto de instruções correspondente 10 vezes. Ou seja, existe apenas uma cópia da função.

O processo de pular para uma função e voltar envolve um certo trabalho adicional para o processador. Quando a função é muito pequena, contendo apenas uma ou duas linhas de código, podemos aumentar a eficiência do programa evitando os saltos para executar apenas uma ou duas instruções. Aqui, eficiência quer dizer velocidade: o programa torna-se mais rápido se houver menos chamadas a funções.

Se uma função é declarada com a palavra-chave inline, o compilador não cria de fato uma função. Ele copia o código da função inline diretamente para o ponto em que ela é chamada. Então, não acontece um salto. É como se os comandos que formam a função tivessem sido escritos diretamente no código.

O ganho em velocidade não vem de graça. Ele tem um custo em termos de tamanho do programa. Se a função inline for chamada 10 vezes, seu código será inserido 10 vezes no executável do programa, aumentando o tamanho deste. Conclusão: somente devemos tornar inline funções muito curtas.

Exemplo

```
// Inline.cpp
// Demonstra o uso
// de uma função inline.
#include <iostream.h>
// Protótipo.
inline int Vezes10(int);
// Multiplica o argumento
// por 10.
int main()
{
        int num;
        cout << "\nDigite um numero + <Enter>: ";
        cin >> num;
        // Chama função.
        num = Vezes10(num);
        cout << "\nNovo valor = "
                << num
                << "\n";
```

```
        return 0;
} // Fim de main()
// Definição.
int Vezes10(int i)
// Multiplica o argumento
// por 10.
{
        return i * 10;
} // Fim de Vezes10(int i)
```

Saída gerada por este programa:

Digite um numero + <Enter>: 99

Novo valor = 990

Exercício

Modifique o programa Inline.cpp, introduzindo uma segunda função inline chamada MostraValor(). Faça com que a função Vezes10() chame a função MostraValor() para exibir seu resultado na tela.

RECURSÃO

Uma função pode chamar a si mesma. Esse processo é chamado recursão. A recursão pode ser direta ou indireta. Ela é direta quando a função chama a si mesma; na recursão indireta, uma função chama outra função, que por sua vez chama a primeira função.

Alguns problemas são solucionados com mais facilidade com o uso de recursão. Geralmente são problemas nos quais fazemos um cálculo com os dados, e depois fazemos novamente o mesmo cálculo com o resultado. A recursão pode ter um final feliz, quando a cadeia de recursão chega a um fim e temos um resultado. Ou pode ter um final infeliz, quando a cadeia recursiva não tem fim e acaba travando o programa.

É importante entender que quando uma função chama a si mesma, uma nova cópia da função passa a ser executada. As variáveis locais da segunda cópia são independentes das variáveis locais da primeira cópia, e não podem afetar umas às outras diretamente

Um exemplo clássico de uso de recursão é a seqüência matemática chamada série de Fibonacci. Na série de Fibonacci, cada número, a partir do terceiro, é igual à soma dos dois números anteriores. Eis a série de Fibonacci:

1, 1, 2, 3, 5, 8, 13, 21, 34...

Geralmente, o que se deseja é determinar qual o n-ésimo número da série. Para solucionar o problema, precisamos examinar com cuidado a série de Fibonacci. Os primeiros dois elementos são iguais a 1. Depois disso, cada elemento subseqüente é igual à soma dos dois anteriores. Por exemplo, o sétimo número é igual à soma do sexto com o quinto. Ou, dito de um modo genérico, o n-ésimo número é igual à soma do elemento n - 1 com o elemento n - 2, desde que n > 2.

Para evitar desastres, uma função recursiva precisa ter uma condição de parada. Alguma coisa precisa acontecer para fazer com que o programa encerre a cadeia recursiva, senão ela se tornará infinita. Na série de Fibonacci, essa condição é n < 3.

Portanto, o algoritmo usado será:

a) Solicitar do usuário a posição desejada na série.

b) Chamar a função Fibonacci, fibo(), usando como argumento essa posição, passando o valor digitado pelo usuário.

c) A função fibo() examina o argumento n. Se n < 3, a função retorna 1; caso contrário, a função fibo() chama a si mesma recursivamente, passando como argumento n - 2, e depois chama a si mesma novamente, passando como argumento n - 1, e retorna a soma.

Assim, se chamarmos fibo(1), ela retornará 1. Se chamarmos fibo(2), ela retornará 1. Se chamarmos fibo(3), ela retornará a soma das chamadas fibo(2) + fibo(1). Como fibo(2) retorna 1 e fibo(1) retorna 1, fibo(3) retornará 2.

Se chamarmos fibo(4), ela retornará a soma das chamadas fibo(3) + fibo(2). Já mostramos que fibo(3) retorna 2, chamando fibo(2) + fibo(1). Mostramos também que fibo(2) retorna 1, de modo que fibo(4) somará esses dois números e retornará 3, que é o quarto elemento da série. O exemplo abaixo ilustra o uso desse algoritmo.

Exemplo

```
// Recurs.cpp
// Utiliza a série de
// Fibonacci para demonstrar
// o uso de uma função recursiva.
#include <iostream.h>
// Protótipo.
int fibo(int i);
// Calcula o valor do
// i-ésimo elemento da
// série de Fibonacci.
int main()
{
        int n, resp;
        cout << "Digite um numero: + <Enter>: ";
        cin >> n;
        resp = fibo(n);
        cout << "\nElemento "
                << n
                << " na serie Fibonacci = "
                << resp;
        return 0;
} // Fim de main()
// Definição.
int fibo(int i)
// Calcula o valor do
// i-ésimo elemento da
// série de Fibonacci.
{
        cout << "\nProcessando fibo("
                << i
                << ")...";
        if(i < 3)
        {
                cout << "Retornando 1...\n";
                return 1;
        } // Fim de if
        else
        {
```

```
                cout << "Chamando fibo("
                       << i - 2
                       << ") e fibo("
                       << i - 1
                       << ").\n";
                return(fibo(i - 2) + fibo(i - 1));
        } // Fim de else.
} // Fim de fibo(int)
```

Saída gerada por este programa:

Digite um numero: + <Enter>: 5

Processando fibo(5)...Chamando fibo(3) e fibo(4).

Processando fibo(3)...Chamando fibo(1) e fibo(2).

Processando fibo(1)...Retornando 1...

Processando fibo(2)...Retornando 1...

Processando fibo(4)...Chamando fibo(2) e fibo(3).

Processando fibo(2)...Retornando 1...

Processando fibo(3)...Chamando fibo(1) e fibo(2).

Processando fibo(1)...Retornando 1...

Processando fibo(2)...Retornando 1...

Elemento 5 na serie Fibonacci = 5

Exercício

Escreva um programa que utilize recursividade para calcular o fatorial de um número.

A PALAVRA-CHAVE GOTO

Nos tempos primitivos da programação, os loops consistiam de um comando goto e um label indicando para onde a execução devia pular. Em C++, um label é simplesmente um nome, seguido do caractere de dois pontos:

A palavra-chave goto tem péssima fama, e isso tem um motivo. O comando goto pode fazer com que a execução do programa pule para qualquer ponto do código fonte, para frente ou para trás. O uso indiscriminado do comando goto pode criar código confuso, difícil de ler e de manter, conhecido como "código espaguete". Por isso, todo professor de programação insiste com seus alunos para que evitem o uso de goto.

Para evitar o uso de goto, existem formas de controlar a execução de maneira muito mais sofisticada e racional. Em C++ temos os comandos for, while, do...while. O uso dessas palavras-chave torna os programas mais fáceis de entender.

De qualquer modo, o comitê ANSI que padronizou a linguagem C++ decidiu manter a palavra-chave goto por dois motivos: (a) um bom programador sempre pode encontrar usos legítimos para goto e (b) para manter a compatibilidade com código já existente que utiliza goto.

Exemplo

```
// GotoTst.cpp
// Ilustra o uso de
// goto em C++.
#include <iostream.h>
int main()
{
        int contador = 0;
        label1: // Label para goto.
        contador++;
        // Exibe valor.
        cout << "\nContador = "
                << contador;
        if(contador < 10)
                goto label1;
        cout << "\n\nValor final: Contador = "
                << contador;
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

Contador = 1

Contador = 2

Contador = 3

Contador = 4

Contador = 5

Contador = 6

Contador = 7

Contador = 8

Contador = 9

Contador = 10

Valor final: Contador = 10

Exercício

Reescreva o exemplo GotoTst.cpp, de maneira que o loop goto seja interrompido ao atingir determinado valor. Para isso, utilize um segundo goto e um segundo label.

O LOOP WHILE

O loop while faz com que o programa repita uma seqüência de comandos enquanto uma determinada condição for verdadeira. Eis a forma genérica do loop while:

while(condicao)

comando;

O exemplo abaixo ilustra o uso de while.

Exemplo

```
// WhileTst.cpp
// Ilustra o uso
// do loop while.
#include <iostream.h>
int main()
{
        int contador = 0;
        while(contador < 10)
        {
                contador++;
                // Exibe valor.
                cout << "\nContador = "
                        << contador;
        } // Fim de while.
        cout << "\n\nValor final: Contador = "
                << contador;
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

Contador = 1

Contador = 2

Contador = 3

Contador = 4

Contador = 5

Contador = 6

Contador = 7

Contador = 8

Contador = 9

Contador = 10

Valor final: Contador = 10

Exercício

Reescreva o exemplo WhileTst.cpp, de maneira que, ao atingir um determinado valor, o loop while seja interrompido (Dica: usar break).

MAIS LOOP WHILE

A condição de teste do loop while pode ser complexa, desde que seja uma expressão C++ válida. Isso inclui expressões produzidas com o uso dos operadores lógicos && (AND), || (OR) e ! (NOT).

Exemplo

```
// MaWhile.cpp
// Outro exemplo de
// uso de while.
#include <iostream.h>
int main()
{
        int pequeno, grande;
        cout << "\nDigite um numero "
                "pequeno: ";
        cin >> pequeno;
        cout << "\nDigite um numero "
                "grande: ";
        cin >> grande;
        while((pequeno < grande) &&
                (grande > 0))
        {
                if((pequeno % 1000) == 0)
                        cout << "\npequeno = "
                                << pequeno;
                pequeno += 5;
                grande -= 20;
        } // Fim de while.
        cout << "\n*** Valores finais ***";
        cout << "\npequeno = "
                << pequeno
                << ", grande = "
                << grande
                << "\n";
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

Digite um numero pequeno: 19

Digite um numero grande: 2000

*** Valores finais ***

pequeno = 419, grande = 400

Exercício

Reescreva o exemplo MaWhile.cpp, definindo o número 100000 como o valor máximo a ser alcançado pela variável pequeno.

BREAK E CONTINUE

Às vezes pode ser necessário voltar para o topo do loop while antes que todos os comandos do loop tenham sido executados. O comando continue faz com que a execução volte imediatamente para o topo do loop.

Outras vezes, pode ser necessário sair do loop antes que a condição de término do loop seja satisfeita. O comando break causa a saída imediata do loop while. Neste caso, a execução do programa é retomada após a chave de fechamento do loop }

Exemplo

```
// BrkCont.cpp
// Ilustra o uso
// de break e continue.
#include <iostream.h>
int main()
{
        int num, limite,
        num_achar, num_pular;
        cout << "\nDigite o num. inicial: ";
        cin >> num;
        cout << "\nDigite o num. limite: ";
        cin >> limite;
        cout << "\nDigite um num. "
                "a ser encontrado: ";
        cin >> num_achar;
        cout << "\nDigite um num. cujos multiplos "
                "\nserao pulados na procura: ";
        cin >> num_pular;
        while(num < limite)
        {
                num++;
                if(num % num_pular == 0)
                {
                        cout << "\nPulando "
                                << num
                                << "...";
                        continue;
                } // Fim de if.
                cout << "\nProcurando... num = "
                        << num
                        << "...";
                if(num == num_achar)
                {
                        cout << "\nAchei! num = "
                                << num;
                        break;
                } // Fim de if.
        } // Fim de while.
        if(num != num_achar)
                cout << "\n\nO num. "
                        << num_achar
                        << " foi pulado.\n";
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

Procurando... num = 586...

Procurando... num = 587...

Procurando... num = 588...

Procurando... num = 589...

Procurando... num = 590...

Procurando... num = 591...

Procurando... num = 592...

Procurando... num = 593...

Procurando... num = 594...

Procurando... num = 595...

Procurando... num = 596...

Procurando... num = 597...

Procurando... num = 598...

Procurando... num = 599...

Procurando... num = 600...

Achei! num = 600

Exercício

Reescreva o exemplo BrkCont1.cpp, usando um loop while para checar a validade dos valores digitados pelo usuário.

O LOOP WHILE INFINITO

A condição testada em um loop while pode ser qualquer expressão C++ válida. Enquanto essa condição permanecer verdadeira, o loop while continuará a ser executado. Podemos criar um loop sem fim, usando true ou 1 como condição de teste. É claro que em algum ponto do loop deve haver um comando break, para evitar que o programa fique travado.

Exemplo

```
// WhileTr.cpp
// Ilustra o uso de
// um loop while infinito.
#include <iostream.h>
int main()
{
```

```
        int contador = 0;
        while(true)
        {
                cout << "\nContador = "
                        << contador++;
                // Esta condição determina
                // o fim do loop.
                if(contador > 20)
                        break;
        } // Fim de while.
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

Contador = 0

Contador = 1

Contador = 2

Contador = 3

Contador = 4

Contador = 5

Contador = 6

Contador = 7

Contador = 8

Contador = 9

Contador = 10

Contador = 11

Contador = 12

Contador = 13

Contador = 14

Contador = 15

Contador = 16

Contador = 17

Contador = 18

Contador = 19

Contador = 20

Exercício

Reescreva o exemplo WhileTr.cpp usando um comando if, combinado com as palavras-chave break e continue para definir o final do loop.

O LOOP DO...WHILE

Pode acontecer do corpo de um loop while nunca ser executado, se a condição de teste nunca for verdadeira. Muitas vezes, é desejável que o corpo do loop seja executado no mínimo uma vez. Para isto, usamos o loop do...while. Eis a forma genérica do loop do...while:

do

comando;

while(condicao);

O loop do...while garante que os comandos que formam o corpo do loop serão executados pelo menos uma vez.

Exemplo

```
// DoWhile.cpp
// Ilustra o uso do
// loop do...while
#include <iostream.h>
int main()
{
        int num;
        cout << "\nDigite um num. entre 1 e 10: ";
        cin >> num;
        while(num > 6 && num < 10)
        {
                cout << "\nEsta mensagem pode "
                        "ou nao ser exibida...";
                num++;
        } // Fim de while.
        do
        {
                cout << "\nEsta mensagem sera' exibida "
                        "pelo menos uma vez...";
                num++;
        } while(num > 6 && num < 10);
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

Digite um num. entre 1 e 10: 9

Esta mensagem pode ou nao ser exibida...

Esta mensagem sera' exibida pelo menos uma vez...

Exercício

Modifique o exemplo DoWhile.cpp, acrescentando um loop do...while para checar a validade dos valores digitados pelo usuário.

O LOOP FOR

Quando usamos o loop while, muitas vezes precisamos definir uma condição inicial, testar essa condição para ver se continua sendo verdadeira e incrementar ou alterar uma variável a cada passagem pelo loop.

Quando sabemos de antemão o número de vezes que o corpo do loop deverá ser executado, podemos usar o loop for, ao invés de while.

Eis a forma genérica do loop for:

for(inicializacao; condicao; atualizacao)

comando;

O comando de inicializacao é usado para inicializar o estado de um contador, ou para preparar o loop de alguma outra forma. A condicao é qualquer expressão C++, que é avaliada a cada passagem pelo loop. Se condicao for verdadeira, o corpo do loop é executado e depois a atualizacao é executada (geralmente, o contador é incrementado).

Exemplo

```
// LoopFor.cpp
// Ilustra o uso do
// loop for.
#include <iostream.h>
int main()
{
        int contador = 0;
        cout << "\n*** Usando while ***";
        while(contador < 10)
        {
                contador++;
                cout << "\nContador = "
                        << contador;
        } // Fim de while.
        cout << "\n\n*** Usando for ***";
        for(contador = 0; contador <= 10; contador++)
                cout << "\nContador = "
                        << contador;
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

*** Usando while ***

Contador = 1

Contador = 2

Contador = 3

Contador = 4

Contador = 5

Contador = 6

Contador = 7

Contador = 8

Contador = 9

Contador = 10

*** Usando for ***

Contador = 0

Contador = 1

Contador = 2

Contador = 3

Contador = 4

Contador = 5

Contador = 6

Contador = 7

Contador = 8

Contador = 9

Contador = 10

Exercício

Reescreva o exemplo LoopFor.cpp, de maneira que no loop for os valores da variável contador variem de 10 a 1.

LOOP FOR COM MÚLTIPLOS COMANDOS

O loop for é extremamente flexível, permitindo inúmeras variações. Ele pode, por exemplo, inicializar mais de uma variável, testar uma condição composta e executar múltiplos comandos. A inicializacao e a atualizacao podem ser substituídas por múltiplos comandos C++, separados por vírgulas.

Exemplo

```
// ForDupl.cpp
// Ilustra o uso do
```

```
// loop for com
// múltiplos comandos.
#include <iostream.h>
int main()
{
        cout << "\n\n*** Usando for duplo ***";
        for(int paraCima = 1, paraBaixo = 10;
                paraCima <= 10/*, paraBaixo >= 1*/;
                paraCima++, paraBaixo--)
        cout << "\nparaCima = "
                << paraCima
                << "\tparaBaixo = "
                << paraBaixo;
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

*** Usando for duplo ***

paraCima = 1 paraBaixo = 10

paraCima = 2 paraBaixo = 9

paraCima = 3 paraBaixo = 8

paraCima = 4 paraBaixo = 7

paraCima = 5 paraBaixo = 6

paraCima = 6 paraBaixo = 5

paraCima = 7 paraBaixo = 4

paraCima = 8 paraBaixo = 3

paraCima = 9 paraBaixo = 2

paraCima = 10 paraBaixo = 1

Exercício

Modifique o exemplo ForDupl.cpp, acrescentando ao loop for uma variável int negPos, cujo valor inicial seja -5, e que seja incrementada a cada passagem pelo loop. Faça com que o valor dessa variável seja também exibido na tela a cada passagem pelo loop.

LOOP FOR COM COMANDOS NULOS

Qualquer um, ou todos os comandos do cabeçalho do loop for podem ser nulos. Para isso, basta utilizar um caractere de ponto e vírgula ; para marcar a posição do comando nulo. Com isso, podemos, por exemplo, criar um loop for que funciona da mesma forma que um loop while.

Exemplo

```
// ForNull.cpp
// Ilustra o uso do
// loop for com
// comandos nulos.
#include <iostream.h>
int main()
{
        // Inicialização fora do loop.
        int contador = 1;
        for( ; contador <= 10; )
        {
                cout << "\nContador = "
                        << contador;
                contador++;
        } // Fim de for.
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

Contador = 1

Contador = 2

Contador = 3

Contador = 4

Contador = 5

Contador = 6

Contador = 7

Contador = 8

Contador = 9

Contador = 10

Exercício

Reescreva o exemplo ForNull.cpp, declarando duas variáveis, int paraCima, e int paraBaixo. Faça com que o loop exiba os valores dessas variáveis variando de 1 a 10 e de 10 a 1, respectivamente. Utilize um loop for com o comando de inicialização e o comando de atualização nulos.

FOR COM TODOS OS COMANDOS NULOS

Um dos motivos do poder de C++ é sua flexibilidade. Com C++, sempre temos diversas maneiras de fazer uma

mesma coisa. Embora o exemplo abaixo não seja exatamente um primor de estilo de programação, ele serve para ilustrar a flexibilidade do loop for.

O exemplo ilustra também o uso de break com o loop for.

Exemplo

```
// ForNada.cpp
// Ilustra o uso
// de um loop for
// com todos os comandos
// de controle nulos.
#include <iostream.h>
int main()
{
        int contador = 1;
        cout << "\nContando de "
                << contador
                << " a 10...\n\n";
        for( ; ; )
        {
                if(contador <= 10)
                {
                        cout << contador << " ";
                        contador++;
                } // Fim de if
                else
                {
                        cout << "\n";
                        break;
                } // Fim de else
        } // Fim de for
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

Contando de 1 a 10...

1 2 3 4 5 6 7 8 9 10

Exercício

Modifique o exemplo ForNada.cpp, de maneira que sejam solicitados um valor inicial e um valor final para a contagem. Utilize um loop do...while para checar a validade dos valores digitados.

O LOOP FOR VAZIO

O cabeçalho do loop for pode fazer tantas coisas que às vezes nem sequer é preciso usar o corpo do loop for.

Neste caso, é obrigatório colocar um comando nulo no corpo do loop. O comando nulo é representado simplesmente pelo caractere de ponto e vírgula ;

Exemplo

```
// ForVaz.cpp
// Ilustra o uso do
// loop for vazio.
#include <iostream.h>
int main()
{
        for(int contador = 1;
        contador <= 10;
        cout << "\nContador = "
                << contador++)
                ; // Comando nulo.
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

Contador = 1

Contador = 2

Contador = 3

Contador = 4

Contador = 5

Contador = 6

Contador = 7

Contador = 8

Contador = 9

Contador = 10

Exercício

Reescreva o programa ForVaz.cpp, de maneira que o loop conte simultaneamente de 1 a 10 e de 10 a 1.

LOOPS FOR ANINHADOS

Podemos ter loops for aninhados, ou seja, o corpo de um loop pode conter outro loop. O loop interno será executado por completo a cada passagem pelo loop externo.

Exemplo

```
// ForAnin.cpp
// Ilustra o uso de
// loops for aninhados.
#include <iostream.h>
int main()
{
        int linhas = 10, colunas = 12;
        for(int i = 0; i < linhas; i++)
        {
                for(int j = 0; j < colunas; j++)
                        cout << "* ";
                cout << "\n";
        } // Fim de for(int i = 0...
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

```
* * * * * * * * * * * *

* * * * * * * * * * * *

* * * * * * * * * * * *

* * * * * * * * * * * *

* * * * * * * * * * * *

* * * * * * * * * * * *

* * * * * * * * * * * *

* * * * * * * * * * * *

* * * * * * * * * * * *

* * * * * * * * * * * *
```

Exercício

Modifique o exemplo ForAnin.cpp, de maneira que o programa exiba em cada posição o número da linha e da coluna correspondentes.

SÉRIE FIBONACCI COM LOOP FOR

Vimos que a série de Fibonacci pode ser solucionada com o uso de recursão. Aqui, usaremos um algoritmo com o loop for para resolver o mesmo problema.

Apenas para relembrar, a série de Fibonacci começa com

1, 1,

e todos os números subseqüentes são iguais à soma dos dois números anteriores:

1, 1, 2, 3, 5, 8, 13, 21, 34...

Assim, o n-ésimo número da série de Fibonacci é igual à soma dos membros n-1 mais n-2.

Exemplo

```
// FibLoop.cpp
// Implementa a série
// de Fibonacci com
// o loop for.
#include <iostream.h>
// Protótipo.
long forFibo(long posicao);
int main()
{
        long resp, pos;
        cout << "\nDigite a posicao na serie: ";
        cin >> pos;
        resp = forFibo(pos);
        cout << "\nPosicao = "
                << pos
                << ", Num. Fibonacci = "
                << resp;
        return 0;
} // Fim de main()
// Definição.
long forFibo(long posicao)
{
        long menosDois,
                menosUm = 1,
                resposta = 2;
        if(posicao < 3)
                return 1;
        for(posicao -= 3; posicao; posicao--)
        {
                menosDois = menosUm;
                menosUm = resposta;
                resposta = menosUm + menosDois;
        } // Fim de for.
        return resposta;
} // Fim de forFibo()
```

Saída gerada por este programa:

Digite a posicao na serie: 20

Posicao = 20, Num. Fibonacci = 6765

Exercício

Modifique o exemplo FibLoop.cpp, de maneira que a cada posição avaliada na série de Fibonacci, o valor calculado seja exibido na tela.

O COMANDO SWITCH

Já vimos como usar o comando if...else. O uso de if...else pode se tornar um tanto complicado quando existem muitas alternativas. Para essas situações, C++ oferece o comando switch. Eis sua forma genérica:

switch(expressao)

{

case valorUm:

comandos;

break;

case valorDois:

comandos;

break;

...

case valorN:

comandos;

break;

default:

comandos;

}

A expressao pode ser qualquer expressão C++ válida, e os comandos, podem ser quaisquer comandos C++, ou blocos de comandos. O switch avalia a expressão e compara o resultado dos valores de cada caso. Observe que a avaliação é somente quanto a igualdade; operadores relacionais não podem ser usados, nem operações booleanas.

Se o valor de um dos casos for igual ao da expressão, a execução salta para os comandos correspondentes àquele caso, e continua até o final do bloco switch, a menos que seja encontrado um comando break. Se nenhum valor for igual ao da expressão, o caso default (opcional) é executado. Se nenhum valor for igual ao da expressão e não houver um caso default, a execução atravessa o switch sem que nada aconteça, e o switch é encerrado.

Exemplo

```
// SwitDem.cpp
// Ilustra o uso do comando
// switch.
#include <iostream.h>
int main()
{
        cout << "\n1 - Verde";
        cout << "\n2 - Azul";
        cout << "\n3 - Amarelo";
        cout << "\n4 - Vermelho";
        cout << "\n5 - Laranja";
```

```
        cout << "\n\nEscolha uma cor: ";
        int numCor;
        cin >> numCor;
        switch(numCor)
        {
                case 1:
                        cout << "\nVoce escolheu Green.";
                        break;
                case 2:
                        cout << "\nVoce escolheu Blue.";
                        break;
                case 3:
                        cout << "\nVoce escolheu Yellow.";
                        break;
                case 4:
                        cout << "\nVoce escolheu Red.";
                        break;
                case 5:
                        cout << "\nVoce escolheu Orange.";
                        break;
                default:
                        cout << "\nVoce escolheu uma cor "
                                "desconhecida";
                        break;
        } // Fim de switch.
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

1 - Verde

2 - Azul

3 - Amarelo

4 - Vermelho

5 - Laranja

Escolha uma cor: 5

Voce escolheu Orange.

Exercício

Modifique o exemplo SwitDem.cpp, acrescentando três opções de cores exóticas. Faça com que, quando qualquer dessas cores for escolhida, o programa exibe uma única mensagem: "Você escolheu uma cor exótica".

MENU COM SWITCH

Um dos usos mais comuns do comando switch é para processar a escolha de um menu de opções. Um loop for, while ou do...while é usado para exibir o menu, e o switch é usado para executar uma determinada ação, dependendo da escolha do usuário.

Exemplo

```
// MenuSwi.cpp
// Ilustra o uso do comando
// switch para implementar
// um menu de opções.
#include <iostream.h>
#include <conio.h>
int main()
{
        int numOpcao;
        do
        {
                cout << "\n0 - Sair";
                cout << "\n1 - Verde";
                cout << "\n2 - Azul";
                cout << "\n3 - Amarelo";
                cout << "\n4 - Vermelho";
                cout << "\n5 - Laranja";
                cout << "\n6 - Bege";
                cout << "\n7 - Roxo";
                cout << "\n8 - Grena'";
                cout << "\n\nEscolha uma cor "
                        "ou 0 para sair: ";
                cin >> numOpcao;
                switch(numOpcao)
                {
                        case 1:
                                clrscr();
                                cout << "\nVoce escolheu Green.\n";
                                break;
                        case 2:
                                clrscr();
                                cout << "\nVoce escolheu Blue.\n";
                                break;
                        case 3:
                                clrscr();
                                cout << "\nVoce escolheu Yellow.\n";
                                break;
                        case 4:
                                clrscr();
                                cout << "\nVoce escolheu Red.\n";
                                break;
                        case 5:
                                clrscr();
                                cout << "\nVoce escolheu Orange.\n";
                                break;
                        case 6:
                        case 7:
                        case 8:
                                clrscr();
                                cout << "\nVoce escolheu uma cor "
                                        "exotica!\n";
                                break;
                        case 0:
                                clrscr();
                                cout << "\nVoce escolheu sair. "
                                        "Tchau!\n";
                                break;
```

```
                        default:
                                clrscr();
                                cout << "\nVoce escolheu uma cor "
                                        "desconhecida.\n";
                                break;
                } // Fim de switch.
        } while(numOpcao != 0);
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

Voce escolheu uma cor exotica!

0 - Sair

1 - Verde

2 - Azul

3 - Amarelo

4 - Vermelho

5 - Laranja

6 - Bege

7 - Roxo

8 - Grena'

Escolha uma cor ou 0 para sair:

Exercício

Modifique o exemplo MenuSwi.cpp, usando um loop for no lugar de do...while.

INTRODUÇÃO A ARRAYS

Um array é uma coleção de elementos. Todos os elementos do array são obrigatoriamente do mesmo tipo de dados.

Para declarar um array, escrevemos um tipo, seguido pelo nome do array e pelo índice. O índice indica o número de elementos do array, e fica contido entre colchetes []

Por exemplo:

long arrayLong[10];

Esta linha declara um array de 10 longs. Aqui, o nome do array é arrayLong. Ao encontrar essa declaração, o compilador reserva espaço suficiente para conter os 10 elementos. Como cada long ocupa quatro bytes, essa declaração faz com que sejam reservados 40 bytes.

Para acessar um elemento do array, indicamos sua posição entre os colchetes []

Os elementos do array são numerados a partir de zero. Portanto, no exemplo acima, o primeiro elemento do array é arrayLong[0]. O segundo elemento é arrayLong[1], e assim por diante. O último elemento do array é arrayLong[9].

Exemplo

```
// InArray.cpp
// Introduz o uso
// de arrays.
#include <iostream.h>
int main()
{
        // Declara um array
        // de 7 ints.
        int intArray[7];
        // Inicializa o array.
        for(int i = 0; i < 7; i++)
                intArray[i] = i * 3;
        // Exibe valores.
        for(int i = 0; i < 7; i++)
                cout << "\nValor de intArray["
                        << i
                        << "] = "
                        << intArray[i];
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

Valor de intArray[0] = 0

Valor de intArray[1] = 3

Valor de intArray[2] = 6

Valor de intArray[3] = 9

Valor de intArray[4] = 12

Valor de intArray[5] = 15

Valor de intArray[6] = 18

Exercício

Modifique o exemplo InArray.cpp de maneira que os valores para inicialização do array sejam solicitados do usuário.

ESCREVENDO ALÉM DO FINAL DE UM ARRAY

Quando escrevemos um valor em um elemento de um array, o compilador calcula onde armazenar o valor com base no tamanho de cada elemento do array e no valor do índice. Suponhamos que queremos escrever um valor em arrayLong[5], ou seja, no sexto elemento do array. O compilador multiplica o índice pelo tamanho de cada elemento, que neste caso é quatro bytes. O valor obtido é 5 * 4 = 20. O compilador vai até a posição que fica a 20 bytes do início do array e escreve o novo valor nesse local.

Se pedirmos para escrever arrayLong[50], o compilador ignora o fato de que esse elemento não existe. Ele calcula a posição da mesma maneira, e escreve o valor desejado nesse local, sem levar em conta o fato de que essa posição de memória pode estar sendo usada para outra finalidade. Virtualmente qualquer coisa pode estar ocupando essa posição. Escrever nela pode ter resultados totalmente imprevisíveis. Se o programador tiver sorte, o programa travará imediatamente. Se o programador não tiver tanta sorte, ele observará um comportamento estranho no programa muito mais tarde, e vai precisar trabalhar duro para descobrir o que está causando esse comportamento.

Exemplo

```
// FimArr.cpp
// Ilustra o risco
// de escrever além do
// final de um array.
// ATENÇÃO: ESTE PROGRAMA
// CONTÉM UM ERRO PROPOSITAL
// E PODE DERRUBAR O SISTEMA.
#include <iostream.h>
int main()
{
        // Um array de 10 longs.
        long longArray[10];
        // Inicializa o array.
        for(int i = 0; i < 10; i++)
        {
                longArray[i] = 5 * i;
        } // Fim de for(int i = 0...
        // Tenta escrever mais dois
        // elementos, além do final do
        // array.
        // ATENÇÃO: ERRO!!!
        longArray[10] = 5 * 10;
        longArray[11] = 5 * 11;
        // Tenta exibir o array,
        // inclusive os dois elementos
        // "extras".
        for(int i = 0; i < 12; i++)
        {
                cout << "\nlongArray["
                        << i
                        << "] = "
                        << longArray[i];
        } // Fim de for.
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

longArray[0] = 0

longArray[1] = 5

longArray[2] = 10

longArray[3] = 15

longArray[4] = 20

longArray[5] = 25

longArray[6] = 30

longArray[7] = 35

longArray[8] = 40

longArray[9] = 45

longArray[10] = 50

longArray[11] = 55

Exercício

Escreva um programa que declare três arrays de long na seguinte ordem:

long lArrayAnterior[3];

long lArrayDoMeio[10];

long lArrayPosterior[3];

- Inicialize lArrayAnterior e lArrayPosterior com todos os valores iguais a zero. - - Inicialize lArrayDoMeio com um valor qualquer diferente de zero em todos os elementos. Por "engano", inicialize 12 elementos em lArrayDoMeio, e não apenas os 10 elementos alocados.

- Exiba os valores dos elementos dos três arrays e interprete os resultados.

ARRAYS MULTIDIMENSIONAIS

Podemos ter arrays com mais de uma dimensão. Cada dimensão é representada como um índice para o array. Portanto, um array bidimensional tem dois índices; um array tridimensional tem três índices, e assim por diante. Os arrays podem ter qualquer número de dimensões. Na prática, porém, a maioria dos arrays têm uma ou duas dimensões.

Exemplo

```
// ArrMult.cpp
// Ilustra o uso de
```

```
// arrays multidimensionais.
#include <iostream.h>
int main()
{
        // Declara e inicializa um
        // array bidimensional.
        int array2D[4][3] = { {2, 4 ,6},
                              {8, 10, 12},
                              {14, 16, 18},
                              {20, 22, 24}
                              };
        // Exibe conteúdo do array.
        for(int i = 0; i < 4; i++)
        {
                for(int j = 0; j < 3; j++)
                        cout << array2D[i][j]
                                << "\t";
                cout << "\n";
        } // Fim de for(int i...
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

2 4 6

8 10 12

14 16 18

20 22 24

Exercício

Modifique o exemplo ArrMult.cpp, criando e exibindo um array tridimensional.

ARRAY DE CARACTERES

Vimos como utilizar objetos da classe string para trabalhar com strings de texto.

Porém no nível mais fundamental, uma string é uma seqüência de caracteres. Um exemplo de string seria a frase usada no programa elementar "Alo, Mundo!".

cout << "Alo, Mundo!\n";

Nesse nível mais básico de C++, uma string é um array de caracteres, terminado com o caractere nulo '\0'. Podemos declarar e inicializar uma string, da mesma forma que fazemos com qualquer outro array:

char frase[] = {'A', 'l', 'o', ',', ' ', 'M', 'u', 'n', 'd', 'o', '!', '\n', '\0'};

O caractere final, '\0', é o caractere nulo. Muitas funções que C++ herdou da linguagem C utilizam esse caractere como um indicador do final da string.

Embora o enfoque caractere por caractere mostrado acima funcione, ele é difícil de digitar e muito sujeito a erros. C++ permite usar uma forma mais racional da linha acima:

char frase[] = "Alo, Mundo!\n";

Devemos observar dois aspectos nessa sintaxe:

(a) Ao invés de caracteres isolados, separados por vírgulas e envolvidos por chaves { }, temos uma string entre aspas duplas " " sem vírgulas nem chaves.

(b) Não precisamos digitar o caractere nulo '\0'. O compilador o insere automaticamente.

Exemplo

```
// CharArr.cpp
// Ilustra o uso de
// arrays de caracteres.
#include <iostream.h>
int main()
{
        char strArray[] = "O rato roeu a roupa do rei.";
        cout << "\nA string e': \n"
                << strArray;
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

A string e':

O rato roeu a roupa do rei.

Exercício

Modifique o exemplo CharArr.cpp, de maneira que a string a ser exibida seja solicitada do usuário.

USANDO CIN COM ARRAY DE CARACTERES

O fluxo de entrada cin pode ser usado para receber uma string digitada pelo usuário, com a mesma sintaxe usada para outros tipos de dados.

cin >> string;

Ao usar cin, para receber uma string, precisamos ter cuidado com dois aspectos:

(a) Se o usuário digitar mais caracteres do que a string que está recebendo a entrada, haverá um estouro da string, com possível travamento do programa (ou algo pior)

(b) Se o usuário digitar um espaço no meio da string, cin interpretará esse espaço como o final da string. O que vier depois do espaço será ignorado.

Exemplo

```
// CinArr.cpp
// Ilustra o uso de cin com
// arrays de caracteres.
// Ilustra o risco de
// estouro de uma string.
// ATENÇÃO: ESTE PROGRAMA PODE
// DERRUBAR O SISTEMA.
#include <iostream.h>
int main()
{
        // Uma string muito pequena.
        char string[16];
        cout << "\nDigite uma frase + <Enter>: \n";
        cin >> string;
        cout << "\nVoce digitou: \n"
                << string;
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

Digite uma frase + <Enter>:

Tarcisio Lopes

Voce digitou:

Tarcisio

Exercício

Modifique o exemplo CinArr.cpp, de maneira que (a) o usuário possa digitar espaços na string (b) não haja risco de estouro da string. Dica: cin.get(char*, int)

ESTRUTURAS: A CONSTRUÇÃO STRUCT

Uma estrutura (struct) é um grupo de dados correlacionados, agrupados sob um único nome. Enquanto os elementos de um array são todos do mesmo tipo, os elementos de uma estruturas, conhecidos como membros, podem ser de diferentes tipos.

As estruturas são equivalentes aos records do Pascal, ou aos tipos user-defined do Basic. Em todas essas linguagens, a possibilidade de agrupar diferentes tipos na mesma construção representa um recurso muito flexível de poderoso para manuseio de dados.

Um exemplo de uso de uma estrutura é em um registro de banco de dados. Suponhamos que queremos escrever um programa de folha de pagamento que registre os seguintes fatos sobre cada funcionário:

- Nome

- Número de meses no emprego

- Salário por hora

Cada um desses itens requer um tipo diferente de dado. O nome pode ser armazenado em uma string (array de char), enquanto um inteiro poderá conter o número de meses no emprego. O salário por hora pode conter uma fração; por isso, será armazenado em uma variável de ponto flutuante.

Embora cada uma dessas variáveis seja de um tipo diferente, podemos agrupá-las todas em uma estrutura única, formando um registro. O programa StrctTst.cpp, abaixo, ilustra como isso é feito.

Exemplo

```
// StrctTst.cpp
// Ilustra o uso de
// uma struct.
#include <iostream.h>
#include <string.h>
// Declara uma struct.
struct funcionario
{
        char nome[32];
        int numMeses;
        float salarioHora;
}; // Fim de struct funcionario.
// Protótipo.
void exibe(struct funcionario func);
int main()
{
        // Declara uma variável
        // do tipo struct funcionario.
        struct funcionario jose;
        // Inicializa a struct.
        strcpy(jose.nome, "Jose da Silva");
        jose.numMeses = 36;
        jose.salarioHora = 25.59;
        // Exibe os dados do
        // funcionario jose.
        exibe(jose);
        return 0;
} // Fim de main()
// Definição da função.
void exibe(struct funcionario func)
{
        cout << "Nome: "
                << func.nome
                << "\n";
        cout << "Meses no emprego: "
                << func.numMeses
                << "\n";
        cout << "Salario por hora: "
                << func.salarioHora
                << "\n";
} // Fim de exibe()
```

Saída gerada por este programa:

Nome: Jose da Silva

Meses no emprego: 36

Salario por hora: 25.59

Exercício

Acrescente à struct funcionario do exemplo StrctTst.cpp um outro membro do tipo int, chamado numSerie. Inicialize e exiba esse membro, junto com os outros.

ENDEREÇOS NA MEMÓRIA

Uma dos recursos mais poderosos que C++ oferece ao programador é a possibilidade de usar ponteiros. Os ponteiros são variáveis que apontam para endereços na memória. Portanto para entender os ponteiros, precisamos entender como funciona a memória do computador.

A memória é dividida em locais de memória, que são numeradas seqüencialmente. Cada variável fica em um local único da memória, conhecido como endereço de memória.

Diferentes tipos de computadores e sistemas operacionais utilizam a memória de formas diversas. Mas isso não precisa ser motivo de preocupação. Em geral, o programador não precisa saber do endereço absoluto de uma dada variável, porque o compilador cuida de todos os detalhes. O verdadeiro poder dos ponteiros está em outra forma de utilização, que veremos adiante. O exemplo abaixo é apenas para ilustrar a possibilidade de acessar um endereço absoluto de memória. Para isso, usamos o operador de endereço &

Exemplo

```
// Ender.cpp
// Ilustra o acesso
// a endereços na
// memória.
#include <iostream.h>
int main()
{
        unsigned short usVar = 200;
        unsigned long ulVar = 300;
        long lVar = 400;
        cout << "\n*** Valores e enderecos ***\n";
        cout << "\nusVar: Valor = "
                << usVar
                << ", Endereco = "
                << &usVar;
        cout << "\nulVar: Valor = "
                << ulVar
                << ", Endereco = "
                << &ulVar;
        cout << "\nlVar: Valor = "
                << lVar
                << ", Endereco = "
                << &lVar
                << "\n";
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

*** Valores e enderecos ***

usVar: Valor = 200, Endereco = 0066FE02

ulVar: Valor = 300, Endereco = 0066FDFC

lVar: Valor = 400, Endereco = 0066FDF8

Exercício

Reescreva o exemplo Ender.cpp de maneira a criar 10 variáveis int. Atribua valores às variáveis. Depois exiba na tela seus valores e endereços.

ENDEREÇOS DE UM ARRAY

Os elementos de um array são organizados em endereços sucessivos da memória. Ou seja, os elementos ficam enfileirados lado a lado na memória, sem vazios ou descontinuidades entre eles. Normalmente, não precisamos nos preocupar com endereços absolutos de memória quando trabalhamos com programas do mundo real. Porém, mais adiante, veremos que a garantia de que os elementos de um array ficam organizados sem descontinuidade na memória tem importantes implicações práticas.

O exemplo abaixo ilustra a organização dos elementos de um array na memória.

Exemplo

```
// ArrTst.cpp
// Ilustra endereços
// dos elementos de um array.
#include <iostream.h>
int main()
{
        int i;
        // Declara um array com
        // 5 ints.
        int intArray[5];
        // Coloca valores no array.
        intArray[0] = 205;
        intArray[1] = 32;
        intArray[2] = 99;
        intArray[3] = 10000;
        intArray[4] = 1234;
        // Exibe valores e endereços
        // dos elementos na memória.
        cout << "*** Valores *** \t *** Enderecos ***\n\n";
        for(i = 0; i < 5; i = i + 1)
        {
                cout << "intArray[" << i << "] = "
                        << intArray[i];
                cout << "\t&intArray[" << i << "] = "
                        << &intArray[i] << "\n";
        } // Fim de for.
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

*** Valores *** *** Enderecos ***

intArray[0] = 205 &intArray[0] = 0066FDF0

intArray[1] = 32 &intArray[1] = 0066FDF4

intArray[2] = 99 &intArray[2] = 0066FDF8

intArray[3] = 10000 &intArray[3] = 0066FDFC

intArray[4] = 1234 &intArray[4] = 0066FE00

Exercício

Modifique o exemplo ArrTst.cpp, criando um array de cinco valores double, ao invés de cinco valores int. Observe os endereços exibidos e interprete os resultados.

PONTEIROS

Vimos que toda variável tem um endereço. Mesmo sem saber o endereço específico de uma dada variável, podemos armazenar esse endereço em um ponteiro.

Por exemplo, suponhamos que temos uma variável idade, do tipo int. Para declarar um ponteiro para conter o endereço dessa variável, podemos escrever:

int* pIdade = 0;

Esta linha declara pIdade como sendo um ponteiro para int. Ou seja, pIdade pode conter o endereço de uma variável int.

Observe que pIdade é uma variável como qualquer outra. Quando declaramos uma variável inteira (do tipo int), ela deve pode conter um valor inteiro. Quando declaramos uma variável ponteiro, como pIdade, ela pode conter um endereço. Assim, pIdade é apenas mais um tipo de variável.

Neste caso, pIdade foi inicializada com o valor zero. Um ponteiro cujo valor é igual a zero é chamado de ponteiro nulo. Todos os ponteiros, quando são criados, devem ser inicializados com algum valor. Se não houver nenhum valor para atribuir a um ponteiro, ele deve ser inicializado com o valor zero. Um ponteiro não-inicializado representa sempre um grande risco de erro.

Eis como fazer com que um ponteiro aponte para algo de útil.

int idade = 18;

// Uma variável inteira.

int* pIdade = 0;

// Um ponteiro para int.

pIdade = &idade;

// O ponteiro agora contém

// o endereço de idade.

A primeira linha cria uma variável, idade, do tipo int, e a inicializa com o valor 18. A segunda linha declara pIdade como sendo um ponteiro para int, e inicializa-o com o valor zero. O que indica que pIdade é um ponteiro é o asterisco * que aparece entre o tipo da variável e o nome da variável. A terceira linha atribui o endereço de idade ao ponteiro pIdade. A forma usada para acessar o endereço de uma variável é o operador endereço de, representado pelo caractere &.

Exemplo

```
// Ponteir.cpp
// Ilustra o uso
// de ponteiros.
#include <iostream.h>
int main()
{
        // Variáveis.
        unsigned short usVar = 200;
        unsigned long ulVar = 300;
        long lVar = 400;
        // Ponteiros.
        unsigned short* usPont = &usVar;
        unsigned long* ulPont = &ulVar;
        long* lPont = &lVar;
        cout << "\n*** Valores iniciais ***\n";
        cout << "\nusVar: Valor = "
                << usVar
                << ", Endereco = "
                << usPont;
        cout << "\nulVar: Valor = "
                << ulVar
                << ", Endereco = "
                << ulPont;
        cout << "\nlVar: Valor = "
                << lVar
                << ", Endereco = "
                << lPont
                << "\n";
        // Modifica valores das variáveis
        // usando os ponteiros.
        *usPont = 210;
        *ulPont = 310;
        *lPont = 410;
        // Exibe os novos valores.
        cout << "\n*** Novos valores ***\n";
        cout << "\nusVar: Valor = "
                << usVar
                << ", Endereco = "
                << usPont;
        cout << "\nulVar: Valor = "
                << ulVar
                << ", Endereco = "
                << ulPont;
        cout << "\nlVar: Valor = "
                << lVar
                << ", Endereco = "
```

```
                << lPont
                << "\n";
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

*** Valores iniciais ***

usVar: Valor = 200, Endereco = 0066FE02

ulVar: Valor = 300, Endereco = 0066FDFC

lVar: Valor = 400, Endereco = 0066FDF8

*** Novos valores ***

usVar: Valor = 210, Endereco = 0066FE02

ulVar: Valor = 310, Endereco = 0066FDFC

lVar: Valor = 410, Endereco = 0066FDF8

Exercício

Modifique o programa Ponteir.cpp, de maneira que os novos valores atribuídos às variáveis sejam exibidos (a) usando as próprias variáveis (b) usando ponteiros para as variáveis.

REUTILIZANDO UM PONTEIRO

Um ponteiro pode ser reutilizado quantas vezes quisermos. Para isso, basta atribuir um novo endereço ao ponteiro. Observe que neste caso, o endereço contido anteriormente no ponteiro é perdido.

Exemplo

```
// PontReu.cpp
// Ilustra a reutilização
// de um ponteiro.
#include <iostream.h>
int main()
{
        // Duas variáveis int.
        int iVar1 = 1000, iVar2 = 2000;
        // Um ponteiro int.
        int* iPont;
        // Inicializa o ponteiro.
        iPont = &iVar1;
        // Exibe valor e endereço
        // de iVar1.
        cout << "\niVar1: Valor = "
                << iVar1
                << ", Endereco = "
```

```
                << iPont;
        // Reatribui o ponteiro.
        iPont = &iVar2;
        // Exibe valor e endereço
        // de iVar2.
        cout << "\niVar2: Valor = "
                << iVar2
                << ", Endereco = "
                << iPont;
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

iVar1: Valor = 1000, Endereco = 0066FE00

iVar2: Valor = 2000, Endereco = 0066FDFC

Exercício

Reescreva o exemplo PontReu.cpp, trabalhando com variáveis double. Faça com que o ponteiro seja reatribuído duas vezes, exibindo o valor armazenado em cada caso.

PONTEIROS PARA PONTEIROS

Um ponteiro pode apontar para qualquer tipo de variável. Como o próprio ponteiro é uma variável, podemos fazer com que um ponteiro aponte para outro ponteiro, criando um ponteiro para ponteiro. Veremos que este conceito é em si bastante útil. Mas é também importante para a compreensão da equivalência entre a notação de array e a notação de ponteiro, que explicaremos mais adiante.

O programa PtrPtr.cpp, abaixo, demonstra um ponteiro para ponteiro da forma mais simples possível.

Exemplo

```
// PtrPtr.cpp
// Ilustra o uso de
// ponteiro para ponteiro.
#include <iostream.h>
int main()
{
        // Uma variável int.
        int iVar = 1000;
        // Um ponteiro para int,
        // inicializado com o
        // endereço de iVar.
        int *iPtr = &iVar;
        // Um ponteiro para ponteiro
        // para int,
        // inicializado com o
        // endereço de iPtr.
        int **iPtrPtr = &iPtr;
```

```
        // Exibe valor da variável,
        // acessando-o via iPtr.
        cout << "Acessando valor via ponteiro\n";
        cout << "iVar = "
                << *iPtr
                << "\n";
        // Exibe valor da variável,
        // acessando-o via iPtrPtr.
        cout << "Acessando valor via ponteiro para ponteiro\n";
        cout << "iVar = "
                << **iPtrPtr
                << "\n";
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

Acessando valor via ponteiro

iVar = 1000

Acessando valor via ponteiro para ponteiro

iVar = 1000

Exercício

Modifique o exemplo PtrPtr.cpp, de maneira que o programa exiba na tela o endereço e o valor de cada uma das variáveis: iVar, iPtr e iPtrPtr.

PONTEIROS PARA ARRAYS

Em C++, ponteiros e arrays estão estreitamente relacionados. Vimos anteriormente que os elementos de um array ficam enfileirados lado a lado na memória. Esta lição explica uma das formas mais simples de usar ponteiros com arrays, tirando partido desse fato.

Uma ponteiro para um array combina dois poderosos recursos da linguagem: a capacidade do ponteiro de proporcionar acesso indireto e a conveniência de acessar elementos de um array com o uso de índices numéricos.

Um ponteiro para um array não é muito diferente de um ponteiro para uma variável simples. Em ambos os casos, o ponteiro somente consegue apontar para um único objeto de cada vez. Um ponteiro para array, porém, pode referenciar qualquer elemento individual dentro de um array. Mas apenas um de cada vez.

O programa PArrTst.cpp mostra como acessar os elementos de um array de int através de um ponteiro.

Exemplo

```
// PArrTst.cpp
// Ilustra o uso de um
// ponteiro para array.
#include <iostream.h>
```

```
// Declara um array de int.
int intArray[] = {10, 15, 296, 3, 18};
int main()
{
        // Declara um ponteiro
        // para int.
        int *arrPont;
        // Uma variável int.
        int i;
        // Atribui ao ponteiro
        // o endereço do primeiro
        // elemento do array.
        arrPont = &intArray[0];
        // Exibe os elementos do array,
        // acessando-os via ponteiro.
        for(i = 0; i < 5; i++)
        {
                cout << "intArray["
                        << i
                        << "] = "
                        << *arrPont
                        << "\n";
                // Incrementa o ponteiro.
                arrPont++;
        } // Fim de for.
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

intArray[0] = 10

intArray[1] = 15

intArray[2] = 296

intArray[3] = 3

intArray[4] = 18

Exercício

Modifique o programa PArrTst.cpp, de maneira que os elementos do array sejam acessados (a) via notação de array (b) via notação de ponteiro. Faça com que os elementos sejam exibidos lado a lado.

PONTEIROS PARA STRUCTS

Um ponteiro para uma estrutura é conceitualmente similar a um ponteiro para array. Da mesma forma que um ponteiro para array pode apontar para qualquer elemento do array, um ponteiro para estrutura pode apontar para qualquer membro da estrutura. A principal diferença é de notação.

Caso você ainda não esteja totalmente familiarizado com a notação de estruturas, vamos revisá-la brevemente. Primeiro, relembraremos que todos os elementos de um array são do mesmo tipo, de modo que podemos nos referir aos elementos individuais do array usando índices:

iArray[10]

Como os membros de uma estrutura podem ser de diferentes tipos, não podemos usar subscritos numéricos para acessá-los com base em sua ordem. Ao invés disso, cada membro de uma estrutura tem um nome simbólico. Podemos acessar cada membro usando o nome da estrutura e o nome do membro, separando os dois nomes com o operador membro-de, indicado por um ponto .

umaStruct.nome

A notação de ponteiros para estruturas segue esse mesmo procedimento, apenas com duas diferenças:

1 - O nome da estrutura deve ser substituído pelo nome do ponteiro

2 - O operador membro-de . deve ser substituído pelo operador aponta-para ->

O operador aponta-para é formado com um hífen - mais o sinal maior do que >

O exemplo abaixo, PtStr.cpp, ilustra o uso operador aponta-para.

Exemplo

```
// PtStr.cpp
// Ilustra o uso de
// ponteiros para
// estruturas.
#include <iostream.h>
#include <string.h>
// Declara uma struct.
struct funcionario
{
        char nome[32];
        int numMeses;
        float salarioHora;
}; // Fim de struct funcionario.
// Protótipo.
void exibe(struct funcionario *pFunc);
int main()
{
        // Declara e inicializa
        // uma variável
        // do tipo struct funcionario.
        struct funcionario jose =
        {
                "Jose da Silva",
                36,
                25.59
        };
        // Exibe os dados do
        // funcionario jose.
        exibe(&jose);
        return 0;
} // Fim de main()
// Definição da função.
void exibe(struct funcionario *pFunc)
```

```
{
        cout << "Nome: "
                << pFunc->nome
                << "\n";
        cout << "Meses no emprego: "
                << pFunc->numMeses
                << "\n";
        cout << "Salario por hora: "
                << pFunc->salarioHora
                << "\n";
} // Fim de exibe()
```

Saída gerada por este programa:

Nome: Jose da Silva

Meses no emprego: 36

Salario por hora: 25.59

Exercício

Acrescente à struct funcionario do exemplo PtStr.cpp um outro membro do tipo int, chamado numSerie. Inicialize e exiba esse membro, junto com os outros.

O NOME DO ARRAY

Há um fato muito importante que precisamos saber sobre o nome de um array em C++: o nome do array é na verdade um ponteiro const para o primeiro elemento do array.

Por exemplo, se temos um array de int declarado da seguinte forma:

int intArray[] = {10, 15, 29, 36, 18};

Podemos inicializar um ponteiro para o primeiro elemento do array assim:

int *arrPont = intArray;

// O mesmo que:

// int *arrPont = &intArray[0];

O exemplo abaixo ilustra como tirar partido desse fato para acessar os elementos de um array de forma mais compacta.

Exemplo

```
// NomArr.cpp
// Ilustra o uso de um
// ponteiro para array, tirando
// partido do fato do
// nome do array ser
// um ponteiro.
```

```
#include <iostream.h>
// Declara um array de int.
int intArray[] = {10, 15, 29, 36, 18};
int main()
{
        // Declara e inicializa
        // um ponteiro
        // para int.
        int *arrPont = intArray;
        // O mesmo que:
        // int *arrPont = &intArray[0];
        // Uma variável int.
        int i;
        // Exibe os elementos do array,
        // acessando-os via ponteiro.
        for(i = 0; i < 5; i++)
                cout << "intArray["
                        << i
                        << "] = "
                        << *arrPont++
                        << "\n";
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

intArray[0] = 10

intArray[1] = 15

intArray[2] = 29

intArray[3] = 36

intArray[4] = 18

Exercício

Reescreva o exemplo NomArr.cpp declarando uma string (array de char) no lugar do array de int intArray.

A FUNÇÃO STRCPY()

C++ herdou de C uma biblioteca de funções para o manuseio de strings. Entre as muitas funções disponíveis nessa biblioteca estão duas que permitem copiar uma string para outra: strcpy() e strncpy(). A primeira, strcpy(), copia todo o conteúdo de uma string para a string de destino.

Exemplo

```
// FStrcpy.cpp
// Ilustra o uso da
// função strcpy()
#include <iostream.h>
#include <string.h>
```

```
int main()
{
        // Declara duas strings.
        char string1[] = "O rato roeu a roupa do rei";
        char string2[128] = "Alo, Mundo!";
        // Exibe as strings.
        cout << "\nString1 = "
                << string1;
        cout << "\nString2 = "
                << string2;
        // Copia string1 para
        // string2.
        strcpy(string2, string1);
        // Exibe novamente.
        cout << "\nApos a copia: ";
        cout << "\nString1 = "
                << string1;
        cout << "\nString2 = "
                << string2;
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

String1 = O rato roeu a roupa do rei

String2 = Alo, Mundo!

Apos a copia:

String1 = O rato roeu a roupa do rei

String2 = O rato roeu a roupa do rei

Exercício

Reescreva o exemplo FStrcpy.cpp de maneira que o conteúdo das duas strings de texto seja trocado entre elas.

A FUNÇÃO STRNCPY()

Quando usamos a função strcpy(), a string de origem é copiada inteira para a string de destino. Se a string de origem for maior que a string de destino, pode haver um estouro do espaço desta última.

Para evitar esse problema, a biblioteca padrão inclui uma outra função de cópia de strings, chamada strncpy(). Essa função recebe, como um de seus argumentos, o número máximo de caracteres a serem copiados. Assim, strncpy() copia os caracteres da string de origem, até encontrar um caractere nulo, sinalizando o final da string, ou até que o número de caracteres copiados atinja o valor especificado no parâmetro recebido para indicar o tamanho.

Exemplo

```
// FStrncp.cpp
```

```
// Ilustra o uso da
// função strncp()
#include <iostream.h>
#include <string.h>
int main()
{
        // Declara duas strings.
        char string1[] = "O rato roeu a roupa do rei";
        char string2[12] = "Alo, Mundo!";
        // Exibe as strings.
        cout << "\nString1 = "
                << string1;
        cout << "\nString2 = "
                << string2;
        // Copia string1 para
        // string2. Usa strncpy()
        // para evitar estouro.
        strncpy(string2, string1, 11);
        // Exibe novamente.
        cout << "\nApos a copia: ";
        cout << "\nString1 = "
                << string1;
        cout << "\nString2 = "
                << string2;
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

String1 = O rato roeu a roupa do rei

String2 = Alo, Mundo!

Apos a copia:

String1 = O rato roeu a roupa do rei

String2 = O rato roeu

Exercício

Reescreva o exemplo FStrncp.cpp de maneira que o programa troque o conteúdo das strings string1 e string2. Utilize a função strncpy() para evitar o risco de estouro.

NEW E DELETE

Para alocar memória do free store em C++, utilizamos o operador new. O operador new deve ser seguido pelo tipo do objeto que queremos alocar, para que o compilador saiba quanta memória é necessária. Por exemplo, a linha:

new int;

aloca exatamente o número de bytes necessários para conter um valor int.

O valor retornado por new é um endereço de memória. Ele deve atribuído a um ponteiro. Por exemplo, para criar um unsigned short no free store, poderíamos escrever:

unsigned short int* shortPont;

shortPont = new unsigned short int;

Naturalmente, poderíamos criar o ponteiro e alocar a memória em uma só linha:

unsigned short int* shortPont =

new unsigned short int;

Em qualquer dos casos, shortPont passa a apontar para uma variável unsigned short int no free store. Por exemplo, poderíamos armazenar um valor na nova variável da seguinte maneira:

*shortPont = 100;

O que esta linha diz é: armazene o valor 100 no endereço contido em shortPont.

Se o operador new não conseguir alocar a memória necessária, ele retorna um ponteiro nulo. É importante checar sempre o valor retornado por new, para saber se a alocação foi bem sucedida.

Depois de terminar o trabalho com a memória alocada com new, devemos chamar o operador delete. Este operador devolve a memória para o free store. Lembre-se que o ponteiro em si - e não a memória para a qual ele aponta - é a variável local. Quando a função na qual o ponteiro foi declarado é encerrada, o ponteiro sai de escopo e deixa de existir. Mas a memória alocada com new não é automaticamente liberada. Nesta situação, a memória torna-se indisponível - não há como acessá-la, nem como liberá-la. Essa situação é chamada vazamento de memória.

Exemplo

```
// NewDel.cpp
// Ilustra o uso
// de new e delete.
#include <iostream.h>
int main()
{
        // Uma variável local.
        int varLocal = 200;
        // Um ponteiro para int.
        int* pVarLocal = &varLocal;
        // Uma variável alocada
        // dinamicamente usando new.
        int* pHeap;
        pHeap = new int;
        if(!pHeap)
        {
                cout << "\nErro alocando memoria!!! "
                        "Encerrando...\n";
                return 0;
        } // Fim de if.
        // Atribui valor à
        // variável dinâmica.
```

```
        *pHeap = 300;
        // Faz uma operação.
        *pHeap *= 2;
        // Exibe valores.
        cout << "\n*** Valores ***\n";
        cout << "\nvarLocal:\tValor = "
               << varLocal
               << "\tEndereco = "
               << pVarLocal;
        cout << "\nVar. dinamica:\tValor = "
               << *pHeap
               << "\tEndereco = "
               << pHeap;
        // Libera memória.
        delete pHeap;
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

*** Valores ***

varLocal: Valor = 200 Endereco = 0066FE00

Var. dinamica: Valor = 600 Endereco = 00683D10

Exercício

Modifique o exemplo NewDel.cpp, de maneira que o ponteiro seja reutilizado, após a liberação da memória com delete, para conter o endereço de uma nova variável alocada dinamicamente.

PONTEIROS SOLTOS

Um dos tipos de bugs mais difíceis de localizar e resolver é aquele causado por ponteiros soltos. Um ponteiro solto é criado quando chamamos delete para um ponteiro - liberando assim a memória para a qual ele aponta - e depois tentamos reutilizar o ponteiro, sem antes atribuir-lhe um novo endereço.

Observe que o ponteiro continua apontando para a antiga área de memória. Porém como essa memória foi deletada, o sistema fica livre para colocar o que quiser ali; o uso desse ponteiro pode então travar o programa ou derrubar todo o sistema. Ou pior, o programa pode continuar funcionando normalmente, e só travar momentos mais tarde, o que torna esse tipo de bug muito difícil de solucionar.

Exemplo

```
// StrayPt.cpp
// Ilustra o perigo
// dos ponteiros soltos.
// ATENÇÃO: ESTE PROGRAMA
// CONTÉM ERROS PROPOSITAIS.
#include <iostream.h>
int main()
{
        // Declara um
```

```
        // ponteiro para int.
        int *pInt;
        // Inicializa.
        pInt = new int;
        if(pInt != 0)
                *pInt = 200;
        // Exibe valor e
        // endereço.
        cout << "\n*pInt = "
                << *pInt
                << "\tEndereco = "
                << pInt
                << "\n";
        // ATENÇÃO: ESTA LINHA
        // CAUSARÁ O ERRO.
        delete pInt;
        // Agora a coisa fica feia...
        *pInt = 300;
        // Exibe valor e
        // endereço.
        cout << "\n*pInt = "
                << *pInt
                << "\tEndereco = "
                << pInt
                << "\n";
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

*pInt = 200 Endereco = 00683D10

*pInt = 300 Endereco = 00683D10

Exercício

Reescreva o exemplo StrayPt.cpp, declarando dois ponteiros para int, pInt1 e pInt2. Inicialize pInt1 usando new, e atribua o valor contido em pInt1 a pInt2. Exiba os valores. Em seguida, aplique delete a pInt1, e volte a atribuir seu valor a pInt2. Observe o resultado.

PONTEIROS CONST

Podemos usar a palavra-chave const em várias posições em relação a um ponteiro. Por exemplo, podemos declarar um ponteiro const para um valor não-const. Isso significa que o ponteiro não pode ser modificado. O valor apontado, porém, pode ser modificado.

Eis como é feita essa declaração:

// Duas variáveis int.

int iVar = 10, iVar2;

// Um ponteiro const

// apontando para iVar.

int* const pConst = &iVar;

Aqui, pConst é constante. Ou seja, a variável int pode ser alterada, mas o ponteiro não pode ser alterado.

Exemplo

```
// ConstPt.cpp
// Ilustra o uso de
// ponteiros const.
#include <iostream.h>
int main()
{
        // Duas variáveis int.
        int iVar = 10, iVar2;
        // Um ponteiro const
        // apontando para iVar.
        int* const pConst = &iVar;
        // Exibe valores iniciais.
        cout << "\n*** Valores iniciais ***\n";
        cout << "\nEndereco = "
                << pConst;
        cout << "\niVar = "
                << *pConst;
        // Faz alterações e
        // exibe valores.
        (*pConst) += 5;
        // OK! iVar
        // não é const.
        // Exibe novos valores
        // de pConst.
        cout << "\n*** Novos valores ***\n";
        cout << "\nEndereco = "
                << pConst;
        cout << "\niVar = "
                << *pConst;
        // Tenta alterar pConst.
        // ERRO!!! pConst é
        // constante.
        //pConst = &iVar2;
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

*** Valores iniciais ***

Endereco = 0066FE00

iVar = 10

*** Novos valores ***

Endereco = 0066FE00

iVar = 15

Exercício

No exemplo ConstPt.cpp, declare um segundo ponteiro, não-const, e faça com que este ponteiro aponte para a variável iVar2. Tente trocar as variáveis apontadas pelos dois ponteiros e observe o resultado.

PONTEIROS PARA VALOR CONST

Outra possibilidade de uso da palavra-chave const com ponteiros é declarar um ponteiro não-const para um valor const. Neste caso, o ponteiro pode ser modificado, passando a apontar para outro endereço. Porém o valor apontando por este ponteiro não pode ser modificado.

// Uma variável int.

int iVar = 10;

// Um ponteiro

// apontando para iVar.

// O objeto apontado é const.

const int* pConst = &iVar;

A linha acima declara um ponteiro para um valor constante int. Ou seja, o valor para o qual pConst aponta não pode ser alterado. Entretanto, o ponteiro pode ser modificado para, por exemplo, apontar para outra variável.

Um truque para facilitar o entendimento é olhar para o que está à direita da palavra-chave const, para descobrir o que está sendo declarado constante. Se o nome do tipo apontado estiver à direita de const, então é o valor que é constante. Se o nome ponteiro estiver à direita de const, o ponteiro é constante.

No exemplo acima, a palavra int está à direita de const. Isso significa que o int (ou seja, o valor apontado) é const.

Exemplo

```
// PtPConst.cpp
// Ilustra o uso de
// ponteiro para const.
#include <iostream.h>
int main()
{
        // Uma variável int.
        int iVar = 10;
        // Um ponteiro
        // apontando para iVar.
        // O objeto apontado é const.
        const int* pConst = &iVar;
        // Exibe valor inicial.
        cout << "\n*** Valor inicial ***\n";
        cout << "\niVar = "
```

```
                << *pConst;
        // Tenta fazer alterações.
        // ERRO! O objeto apontado
        // (iVar) é const.
        //(*pConst) += 5;
        // Faz alteração usando
        // a própria variável
        // iVar.
        iVar += 50;
        // Exibe novo valor.
        cout << "\n*** Novo valor ***\n";
        cout << "\niVar = "
                << *pConst;
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

*** Valor inicial ***

iVar = 10

*** Novo valor ***

iVar = 60

Exercício

No exemplo PtPConst.cpp, declare uma segunda variável int. Faça com que o ponteiro pConst aponte para esta nova variável e exiba o valor da variável usando o ponteiro. O que podemos concluir?

PONTEIRO CONST PARA VALOR CONST

Podemos também ter um ponteiro const para um valor const. Neste caso nenhum dos dois pode ser alterado. O ponteiro não pode ser usado para apontar para outro endereço; e o valor contido na variável não pode ser alterado via ponteiro.

const int * const pontTres;

Aqui, pontTres é um ponteiro constante para um valor constante. O valor não pode ser alterado, e o ponteiro não pode apontar para outra variável.

Exemplo

```
// PCstCst.cpp
// Ilustra o uso de
// ponteiro const
// para valor const.
#include <iostream.h>
int main()
{
        // Duas variáveis int.
        int iVar = 10, iVar1;
        // Um ponteiro const
```

```
        // apontando para iVar.
        // O objeto apontado
        // também é const.
        const int* const pConst = &iVar;
        // Exibe valor inicial.
        cout << "\n*** Valor inicial ***\n";
        cout << "\niVar = "
                << *pConst;
        // Tenta fazer alterações.
        // ERRO! O objeto apontado
        // (iVar) é const.
        //(*pConst) += 5;
        // ERRO!!! O ponteiro
        // também é const.
        //pConst = &iVar1;
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

*** Valor inicial ***

iVar = 10

Exercício

Modifique o exemplo PCstCst.cpp, acrescentando uma função chamada exibe(), que exibe na tela o valor recebido. A função exibe() deve receber como parâmetro um ponteiro const para um valor const. Tente modificar o ponteiro e o valor recebido dentro da função exibe(). Analise o resultado.

INTRODUÇÃO A REFERÊNCIAS

Uma referência é um sinônimo; quando criamos uma referência, devemos inicializá-la com o nome de um objeto alvo. A partir desse momento, a referência funciona como um segundo nome para o alvo, de modo que qualquer coisa que fizermos com a referência estará na verdade sendo feita com o alvo.

Para criar uma referência, digitamos o tipo do objeto alvo, seguido pelo operador de referência &, mais o nome da referência. As referências podem ter qualquer nome que seja válido em C++. Por exemplo, se tivermos uma variável int chamada idade, podemos criar uma referência a ela da seguinte forma:

int &refIdade = idade;

Exemplo

```
// IntrRef.cpp
// Ilustra o uso
// de referências.
#include <iostream.h>
int main()
{
        // Uma variável int.
        int intVar;
```

```
        // Uma referência a int.
        int &intRef = intVar;
        // Inicializa.
        intVar = 433;
        // Exibe valores.
        cout << "\n*** Valores iniciais ***";
        cout << "\nintVar = "
               << intVar;
        cout << "\nintRef = "
               << intRef;
        // Altera valores.
        intRef = 379;
        // Exibe novos valores.
        cout << "\n*** Novos valores ***";
        cout << "\nintVar = "
               << intVar;
        cout << "\nintRef = "
               << intRef;
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

*** Valores iniciais ***

intVar = 433

intRef = 433

*** Novos valores ***

intVar = 379

intRef = 379

Exercício

Modifique o exemplo IntrRef.cpp, alterando os valores da variável intVar usando a referência e depois a própria variável. Exiba os novos valores em cada caso.

REATRIBUIÇÃO DE UMA REFERÊNCIA

Uma referência é inicializada no momento de sua criação. Depois disso, ela apontará sempre para o dado com o qual foi inicializado. A referência não pode ser reatribuída a um novo dado. O que acontece quando tentamos atribuir um novo alvo a uma referência? Lembre-se: uma referência é inicializada no momento de sua criação, e não pode ser reatribuída. Assim, o que aparentemente pode parecer ser uma atribuição de um novo valor à referência, na verdade é a atribuição de um novo valor ao alvo. Veja o exemplo abaixo.

Exemplo

```
// AtrRef.cpp
// Ilustra tentativa de
```

```
// reatribuir uma
// referência.
#include <iostream.h>
int main()
{
        // Duas variáveis int.
        int intVar1 = 555, intVar2 = 777;
        // Uma referência a int.
        int &intRef = intVar1;
        // Exibe valores.
        cout << "\n*** Valores iniciais ***";
        cout << "\nintVar1: Valor = "
                << intVar1
                << "\tEndereco = "
                << &intVar1;
        cout << "\nintRef: Valor = "
                << intRef
                << "\tEndereco = "
                << &intRef;
        // Reatribui referência.
        // (ou tenta)
        // Atenção aqui!...
        intRef = intVar2;
        // Exibe novos valores.
        cout << "\n*** Novos valores ***";
        cout << "\nintVar2: Valor = "
                << intVar2
                << "\tEndereco = "
                << &intVar2;
        cout << "\nintRef: Valor = "
                << intRef
                << "\tEndereco = "
                << &intRef;
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

*** Valores iniciais ***

intVar1: Valor = 555 Endereco = 0066FE00

intRef: Valor = 555 Endereco = 0066FE00

*** Novos valores ***

intVar2: Valor = 777 Endereco = 0066FDFC

intRef: Valor = 777 Endereco = 0066FE00

Exercício

Modifique o exemplo AtrRef.cpp, criando e inicializando duas variáveis double, dVar1 e dVar2. Declare também uma referência a double, e inicializa-a com dVar1. Exiba valores e endereços. Em seguida, atribua o valor de dVar2 à referência. Exiba valores e endereços. Multiplique por 2 o valor de dVar2. Exiba novamente valores e endereços. Observe o resultado e tire conclusões.

REFERÊNCIAS A STRUCTS

Podemos usar referências com qualquer tipo de dados. Isso inclui tipos definidos pelo programador, como estruturas (structs).

Observe que não podemos criar uma referência à struct em si, mas sim a uma variável do tipo struct.

Uma referência a uma variável do tipo struct é tratada da mesma forma que a própria variável à qual se refere. O exemplo abaixo ilustra como isso é feito.

Exemplo

```
// RefStr.cpp
// Ilustra o uso de
// referências a
// estruturas.
#include <iostream.h>
#include <string.h>
// Declara uma struct.
struct funcionario
{
        char nome[32];
        int numMeses;
        float salarioHora;
}; // Fim de struct funcionario.
// Protótipo.
// Recebe uma referência.
void exibe(funcionario& refFunc);
int main()
{
        // Declara e inicializa
        // uma variável
        // do tipo struct funcionario.
        funcionario jose =
        {
                "Jose da Silva",
                36,
                25.59
        };
        // Declara e inicializa
        // uma referência a
        // struct funcionario.
        funcionario& rFunc = jose;
        // Exibe os dados do
        // funcionario jose
        // via referência.
        exibe(rFunc);
        return 0;
} // Fim de main()
// Definição da função.
void exibe(funcionario& refFunc)
{
        cout << "Nome: "
                << refFunc.nome
                << "\n";
        cout << "Meses no emprego: "
                << refFunc.numMeses
```

```
                << "\n";
        cout << "Salario por hora: "
                << refFunc.salarioHora
                << "\n";
} // Fim de exibe()
```

Saída gerada por este programa:

Nome: Jose da Silva

Meses no emprego: 36

Salario por hora: 25.59

Exercício

Acrescente à struct funcionario do exemplo RefStr.cpp um outro membro do tipo int, chamado numSerie. Inicialize e exiba esse membro, junto com os outros.

PASSANDO ARGUMENTOS POR VALOR

Vimos que em C++, as funções têm duas limitações: os argumentos são passados por valor, e somente podem retornar um único valor. O exemplo abaixo relembra essas limitações.

Exemplo

```
// PassVal.cpp
// Ilustra passagem de
// argumentos por valor.
#include <iostream.h>
// Protótipo.
void troca(int, int);
// Troca os valores entre
// os dois parâmetros.
int main()
{
        int iVal1 = 10, iVal2 = 20;
        cout << "\n*** Valores antes de troca() ***\n";
        cout << "\niVal1 = "
                << iVal1
                << "\tiVal2 = "
                << iVal2;
        // Chama função troca.
        troca(iVal1, iVal2);
        // Exibe novos valores.
        cout << "\n*** Valores depois de troca() ***\n";
        cout << "\niVal1 = "
                << iVal1
                << "\tiVal2 = "
                << iVal2;
        return 0;
} // Fim de main()
// Definição da função.
void troca(int x, int y)
// Troca os valores entre
```

```
// os dois parâmetros.
{
        int temp;
        cout << "\nEstamos na funcao troca()\n";
        cout << "\nx = "
                << x
                << ", y = "
                << y;
        cout << "\nTrocando valores...\n";
        // Efetua troca de valores.
        temp = x;
        x = y;
        y = temp;
        cout << "\nApos troca...";
        cout << "\nx = "
                << x
                << ", y = "
                << y;
} // Fim de troca()
```

Saída gerada por este programa:

*** Valores antes de troca() ***

iVal1 = 10 iVal2 = 20

Estamos na funcao troca()

x = 10, y = 20

Trocando valores...

Apos troca...

x = 20, y = 10

*** Valores depois de troca() ***

iVal1 = 10 iVal2 = 20

Exercício

Modifique o exemplo PassVal.cpp, substituindo a função troca() pela função vezes10(int, int). Esta função recebe dois valores int e multiplica-os por 10. Exiba os valores antes e depois da chamada à função.

PASSANDO ARGUMENTOS POR PONTEIROS

Quando passamos um ponteiro para uma função, estamos passando o endereço do argumento. Assim, a função pode manipular o valor que está nesse endereço, e portanto, modificar esse valor.

Essa é uma das formas como C++ contorna as limitações da passagem por valor. Também é uma solução para o fato de que uma função somente pode retornar um valor. O exemplo abaixo ilustra isso.

Exemplo

```
// PassPtr.cpp
// Ilustra passagem de
// argumentos por ponteiros.
#include <iostream.h>
// Protótipo.
void troca(int*, int*);
// Troca os valores entre
// os dois parâmetros.
// Agora troca() recebe
// dois ponteiros para int.
int main()
{
        int iVal1 = 10, iVal2 = 20;
        cout << "\n*** Valores antes de troca() ***\n";
        cout << "\niVal1 = "
                << iVal1
                << "\tiVal2 = "
                << iVal2;
        // Chama função troca.
        troca(&iVal1, &iVal2);
        // Exibe novos valores.
        cout << "\n*** Valores depois de troca() ***\n";
        cout << "\niVal1 = "
                << iVal1
                << "\tiVal2 = "
                << iVal2;
        return 0;
} // Fim de main()
// Definição da função.
void troca(int* px, int* py)
// Troca os valores entre
// os dois parâmetros.
// Observe que troca() agora
// recebe dois ponteiros para int.
{
        int temp;
        cout << "\nEstamos na funcao troca()\n";
        cout << "\n*px = "
                << *px
                << ", *py = "
                << *py;
        cout << "\nTrocando valores...\n";
        // Efetua troca de valores.
        temp = *px;
        *px = *py;
        *py = temp;
        cout << "\nApos troca...";
        cout << "\n*px = "
                << *px
                << ", *py = "
                << *py;
} // Fim de troca()
```

Saída gerada por este programa:

*** Valores antes de troca() ***

iVal1 = 10 iVal2 = 20

Estamos na funcao troca()

*px = 10, *py = 20

Trocando valores...

Apos troca...

*px = 20, *py = 10

*** Valores depois de troca() ***

iVal1 = 20 iVal2 = 10

Exercício

Reescreva o exemplo PassPtr.cpp, definindo a função vezes10(), de maneira que ela possa modificar os valores recebidos como argumentos.

PASSANDO ARGUMENTOS POR REFERÊNCIA

Vimos que o uso de ponteiros pode fazer com que uma função altere os valores recebidos. Isso funciona, mas a sintaxe é muito desajeitada, difícil de ler e sujeita a erros de digitação. Além disso, a necessidade de passar o endereço das variáveis torna o funcionamento interno da função muito transparente, o que não é desejável.

A passagem de argumentos por referência soluciona todos esses problemas.

Exemplo

```
// PassRef.cpp
// Ilustra passagem de
// argumentos por referência.
#include <iostream.h>
// Protótipo.
void troca(int&, int&);
// Troca os valores entre
// os dois parâmetros.
// Agora troca() recebe
// duas referências a int.
int main()
{
        int iVal1 = 10, iVal2 = 20;
        cout << "\n*** Valores antes de troca() ***\n";
        cout << "\niVal1 = "
                << iVal1
                << "\tiVal2 = "
                << iVal2;
        // Chama função troca.
```

```
        troca(iVal1, iVal2);
        // Exibe novos valores.
        cout << "\n*** Valores depois de troca() ***\n";
        cout << "\niVal1 = "
                << iVal1
                << "\tiVal2 = "
                << iVal2;
        return 0;
} // Fim de main()
// Definição da função.
void troca(int& refx, int& refy)
// Troca os valores entre
// os dois parâmetros.
// Observe que troca() agora
// recebe duas referências a int.
{
        int temp;
        cout << "\nEstamos na funcao troca()\n";
        cout << "\nrefx = "
                << refx
                << ", refy = "
                << refy;
        cout << "\nTrocando valores...\n";
        // Efetua troca de valores.
        temp = refx;
        refx = refy;
        refy = temp;
        cout << "\nApos troca...";
        cout << "\nrefx = "
                << refx
                << ", refy = "
                << refy;
} // Fim de troca()
```

Saída gerada por este programa:

*** Valores antes de troca() ***

iVal1 = 10 iVal2 = 20

Estamos na funcao troca()

refx = 10, refy = 20

Trocando valores...

Apos troca...

refx = 20, refy = 10

*** Valores depois de troca() ***

iVal1 = 20 iVal2 = 10

Exercício

Reescreva o exemplo PassRef.cpp, definindo a função vezes10() com o uso de referências como argumentos.

RETORNANDO VALORES POR PONTEIROS

Vimos anteriormente que uma função somente pode retornar um valor. E se precisarmos que uma função nos forneça dois valores?

Uma forma de resolver esse problema é passar dois objetos para a função por referência. A função pode então colocar os valores corretos nos objetos. Como a passagem por referência permite que a função altere os objetos originais, isso efetivamente permite que a função retorne dois elementos de informação. Esse enfoque ignora o valor retornado pela função com a palavra-chave return. Esse valor pode então ser usado para reportar erros.

Observe que quando falamos de passagem por referência, estamos nos referindo tanto ao uso de ponteiros como de referências propriamente ditas. Inicialmente, usaremos ponteiros.

Exemplo

```
// RetPtr.cpp
// Ilustra o uso de
// ponteiros para retornar
// valores de uma função.
#include <iostream.h>
// Protótipo.
bool quadCubo(long, long*, long*);
// Calcula o quadrado e o cubo do
// número recebido no primeiro argumento.
// Armazena o quadrado no segundo argumento
// e o cubo no terceiro argumento.
// Retorna true para indicar sucesso,
// ou false para indicar erro.
int main()
{
        long num, quadrado, cubo;
        bool sucesso;
        cout << "\nDigite um num. (menor que 1000): ";
        cin >> num;
        sucesso = quadCubo(num, &quadrado, &cubo);
        if(sucesso)
        {
                cout << "\nNumero = "
                        << num;
                cout << "\nQuadrado = "
                        << quadrado;
                cout << "\nCubo = "
                        << cubo;
        } // Fim de if.
        else
                cout << "\nHouve um erro!\n";
        return 0;
} // Fim de main()
// Definição da função.
bool quadCubo(long valor, long* pQuad, long* pCub)
// Calcula o quadrado e o cubo do
```

```
// número recebido no primeiro argumento.
// Armazena o quadrado no segundo argumento
// e o cubo no terceiro argumento.
// Retorna true para indicar sucesso,
// ou false para indicar erro.
{
        bool suc = true;
        if(valor >= 1000)
                suc = false;
        else
        {
                *pQuad = valor * valor;
                *pCub = valor * valor * valor;
        } // Fim de else.
        return suc;
} // Fim de quadCubo()
```

Saída gerada por este programa:

Digite um num. (menor que 1000): 999

Numero = 999

Quadrado = 998001

Cubo = 997002999

Exercício

Modifique o exemplo RetPtr.cpp, acrescentando mais uma operação à função quadCubo(). Ela deve multiplicar por 1000 o valor recebido no primeiro argumento, e retornar o produto desta multiplicação em um quarto argumento.

RETORNANDO VALORES COMO REFERÊNCIAS

Vimos como usar ponteiros para retornar mais de um valor de uma função. Embora isso funcione, o programa torna-se mais legível e fácil de manter se usarmos referências. Eis um exemplo.

Exemplo

```
// RetRef.cpp
// Ilustra o uso de
// referências para retornar
// valores de uma função.
#include <iostream.h>
// Protótipo.
bool quadCubo(long, long&, long&);
// Calcula o quadrado e o cubo do
// número recebido no primeiro argumento.
// Armazena o quadrado no segundo argumento
// e o cubo no terceiro argumento.
// Retorna true para indicar sucesso,
// ou false para indicar erro.
int main()
{
        long num, quadrado, cubo;
        bool sucesso;
```

```
        cout << "\nDigite um num. (menor que 1000): ";
        cin >> num;
        sucesso = quadCubo(num, quadrado, cubo);
        if(sucesso)
        {
                cout << "\nNumero = "
                        << num;
                cout << "\nQuadrado = "
                        << quadrado;
                cout << "\nCubo = "
                        << cubo;
        } // Fim de if.
        else
                cout << "\nHouve um erro!\n";
        return 0;
} // Fim de main()
// Definição da função.
bool quadCubo(long valor,
                long& refQuad,
                long& refCub)
// Calcula o quadrado e o cubo do
// número recebido no primeiro argumento.
// Armazena o quadrado no segundo argumento
// e o cubo no terceiro argumento.
// Retorna true para indicar sucesso,
// ou false para indicar erro.
{
        bool suc = true;
        if(valor >= 1000)
                suc = false;
        else
        {
                refQuad = valor * valor;
                refCub = valor * valor * valor;
        } // Fim de else.
        return suc;
} // Fim de quadCubo()
```

Saída gerada por este programa:

Digite um num. (menor que 1000): 800

Numero = 800

Quadrado = 640000

Cubo = 512000000

Exercício

Modifique o exemplo RetRef.cpp, acrescentando mais uma operação à função quadCubo(). Ela deve multiplicar por 1000 o valor recebido no primeiro argumento, e retornar o produto desta multiplicação em um quarto argumento.

ALTERANDO MEMBROS DE UMA STRUCT

Já vimos que quando passamos um argumento por valor para uma função, a função cria uma cópia local desse argumento. As modificações feitas nessa cópia local não têm efeito sobre o dado original.

Essa limitação se aplica também a variáveis do tipo struct. O exemplo abaixo ilustra esse fato.

Exemplo

```
// AltStruc.cpp
// Ilustra tentativa
// de alterar struct
// por valor.
#include <iostream.h>
#include <string.h>
// Declara uma struct.
struct Cliente
{
        int numCliente;
        char nome[64];
        float saldo;
}; // Fim de struct Cliente.
// Protótipos.
// Funções recebem por valor.
void alteraSaldo(Cliente cl, float novoSaldo);
void exibe(Cliente cl);
int main()
{
        // Declara e inicializa
        // uma variável
        // do tipo Cliente.
        Cliente cl1 =
        {
                301,
                "Tarcisio Lopes",
                100.99
        };
        // Exibe os dados do
        // cliente.
        cout << "*** Antes de alteraSaldo() ***\n";
        exibe(cl1);
        // Altera saldo(?)
        alteraSaldo(cl1, 150.01);
        // Exibe novos dados (?)
        cout << "*** Depois de alteraSaldo() ***\n";
        exibe(cl1);
        return 0;
} // Fim de main()
// Definições das funções.
void exibe(Cliente cl)
{
        cout << "Num. cliente: "
                << cl.numCliente
                << "\n";
        cout << "Nome: "
                << cl.nome
                << "\n";
        cout << "Saldo: "
```

```
                << cl.saldo
                << "\n";
} // Fim de exibe()
void alteraSaldo(Cliente cl, float novoSaldo)
{
        cl.saldo = novoSaldo;
} // Fim de alteraSaldo()
```

Saída gerada por este programa:

*** Antes de alteraSaldo() ***

Num. cliente: 301

Nome: Tarcisio Lopes

Saldo: 100.99

*** Depois de alteraSaldo() ***

Num. cliente: 301

Nome: Tarcisio Lopes

Saldo: 100.99

Exercício

Modifique o exemplo AltStruc.cpp usando referências, de maneira que a função alteraSaldo() consiga modificar a struct original, recebida como argumento.

RETORNANDO REFERÊNCIA INVÁLIDA

As referências são elegantes e fáceis de usar. Por isso, os programadores podem às vezes se descuidar e abusar delas. Lembre-se que uma referência é sempre um sinônimo para algum outro objeto. Ao passar uma referência para dentro ou para fora de uma função, convém ter certeza sobre qual a variável que está de fato sendo referenciada.

A passagem de uma referência a uma variável que já não existe representa um grande risco. O exemplo abaixo ilustra esse fato.

Exemplo

```
// RefInv.cpp
// Ilustra tentativa
// de retornar
// referência inválida.
// AVISO: ESTE PROGRAMA
// CONTÉM ERROS PROPOSITAIS.
#include <iostream.h>
#include <string.h>
// Declara uma struct.
struct Cliente
```

```
{
        int numCliente;
        char nome[64];
        float saldo;
}; // Fim de struct Cliente.
// Protótipos.
// Tenta retornar referência.
Cliente& alteraTudo(int novoNum,
char* novoNome,
float novoSaldo);
void exibe(Cliente& refCl);
int main()
{
        // Declara e inicializa
        // uma variável
        // do tipo Cliente.
        Cliente cl1 =
        {
                301,
                "Tarcisio Lopes",
                100.99
        };
        // Exibe os dados do
        // cliente.
        cout << "*** Antes de alteraSaldo() ***\n";
        exibe(cl1);
        // Altera valores(?)
        cl1 = alteraTudo(1002,
                "Fulano de Tal",
                150.01);
        // Exibe novos dados (?)
        cout << "*** Depois de alteraSaldo() ***\n";
        exibe(cl1);
        return 0;
} // Fim de main()
// Definições das funções.
void exibe(Cliente& refCl)
{
        cout << "Num. cliente: "
                << refCl.numCliente
                << "\n";
        cout << "Nome: "
                << refCl.nome
                << "\n";
        cout << "Saldo: "
                << refCl.saldo
                << "\n";
} // Fim de exibe()
Cliente& alteraTudo(int novoNum,
        char* novoNome,
        float novoSaldo)
{
        Cliente novoCli;
        novoCli.numCliente = novoNum;
        strcpy(novoCli.nome, novoNome);
        novoCli.saldo = novoSaldo;
        return novoCli;
} // Fim de alteraTudo()
```

Saída gerada por este programa

(Compilação):

Error RefInv.cpp 78: Attempting to return a reference to local variable 'novoCli

' in function alteraTudo(int,char *,float)

*** 1 errors in Compile ***

Exercício

Modifique o exemplo RefInv.cpp, para que ele compile e rode corretamente retornando uma referência.

ARRAYS DE PONTEIROS

Até agora, discutimos os arrays como variáveis locais a uma função. Por isso, eles armazenam todos os seus membros na pilha.

Geralmente, o espaço de memória da pilha é muito limitado, enquanto o free store é muito maior. Assim, pode ser conveniente declarar um array de ponteiros e depois alocar todos os dados no free store. Isso reduz dramaticamente a quantidade de memória da pilha usada pelo array.

Exemplo

```
// ArrPont.cpp
// Ilustra o uso de
// arrays de ponteiros.
#include <iostream.h>
int main()
{
        // Um array de ponteiros
        // para double
        double* arrPontDouble[50];
        // Inicializa.
        for(int i = 0; i < 50; i++)
        {
                arrPontDouble[i] = new double;
                if(arrPontDouble[i])
                {
                        *(arrPontDouble[i]) = 1000 * i;
                } // Fim de if.
        } // Fim de for(int i...
        // Exibe.
        for(int i = 0; i < 50; i++)
        {
                cout << "\n*(arrPontDouble["
                        << i
                        << "]) = "
                        << *(arrPontDouble[i]);
        } // Fim de for(int i = 1...
        // Deleta array.
        cout << "\nDeletando array...";
        for(int i = 0; i < 50; i++)
        {
                if(arrPontDouble[i])
                {
```

```
                        delete arrPontDouble[i];
                        arrPontDouble[i] = 0;
                } // Fim de if(arrPontDouble[i])
        } // Fim de for(int i = 1...
        return 0;
} // Fim de main()
```

Saída gerada por este programa

(Parcial):

*(arrPontDouble[44]) = 44000

*(arrPontDouble[45]) = 45000

*(arrPontDouble[46]) = 46000

*(arrPontDouble[47]) = 47000

*(arrPontDouble[48]) = 48000

*(arrPontDouble[49]) = 49000

Deletando array...

Exercício

Modifique o exemplo ArrPont.cpp, definindo uma struct Cliente, da seguinte forma:

struct Cliente

{

int numCliente;

float saldo;

};

Faça com que o array criado seja de ponteiros para Cliente.

ARRAYS NO HEAP

Outra forma de trabalhar com arrays é colocar todo o array no free store. Para isso, precisamos chamar new, usando o operador de índice []

Por exemplo, a linha:

Cliente* arrClientes = new Cliente[50];

declara arrClientes como sendo um ponteiro para o primeiro elemento, em um array de 50 objetos do tipo Cliente. Em outras palavras, arrClientes aponta para arrClientes[0].

A vantagem de usar arrClientes dessa maneira é a possibilidade de usar aritmética de ponteiros para acessar cada membro de arrClientes. Por exemplo, podemos escrever:

Cliente* arrClientes = new Cliente[50];

Cliente* pCliente = arrClientes;

// pCliente aponta para arrClientes[0]

pCliente++;

// Agora, pCliente aponta para arrClientes[1]

Observe as declarações abaixo:

Cliente arrClientes1[50];

arrClientes1 é um array de 50 objetos Cliente.

Cliente* arrClientes2[50];

arrClientes2 é um array de 50 ponteiros para Cliente.

Cliente* arrClientes3 = new Cliente[50];

arrClientes3 é um ponteiro para um array de 50 Clientes.

As diferenças entre essas três linhas de código afetam muito a forma como esses arrays operam. Talvez o mais surpreendente seja o fato de que arrClientes3 é uma variação de arrClientes1, mas é muito diferente de arrClientes2.

Isso tem a ver com a delicada questão do relacionamento entre arrays e ponteiros. No terceiro caso, arrClientes3 é um ponteiro para um array. Ou seja, o endereço contido em arrClientes3 é o endereço do primeiro elemento desse array. É exatamente o que acontece com arrClientes1.

Conforme dissemos anteriormente, o nome do array é um ponteiro constante para o primeiro elemento do array. Assim, na declaração

Cliente arrClientes1[50];

arrClientes1 é um ponteiro para &arrClientes1[0], que é o endereço do primeiro elemento do array arrClientes1. É perfeitamente legal usar o nome de um array como um ponteiro constante, e vice versa. Por exemplo, arrClientes1 + 4 é uma forma legítima de acessar arrClientes1[4].

Exemplo

```
// ArrHeap.cpp
// Ilustra o uso de
// arrays no heap
// (free store).
#include <iostream.h>
int main()
{
        // Aloca um array de doubles
        // no heap.
        double* arrDouble = new double[50];
        // Checa alocação.
        if(!arrDouble)
```

```
                return 0;
        // Inicializa.
        for(int i = 0; i < 50; i++)
                arrDouble[i] = i * 1000;
        // O mesmo que:
        //*(arrDouble + i) = i * 100;
        // Exibe.
        for(int i = 0; i < 50; i++)
                cout << "\narrDouble["
                        << i
                        << "] = "
                        << arrDouble[i];
                /***************
                // O mesmo que:
                cout << "\n*(arrDouble + "
                        << i
                        << ") = "
                        << *(arrDouble + i);
                ***************/
        // Deleta array.
        cout << "\nDeletando array...";
        if(arrDouble)
                delete[] arrDouble;
                // O mesmo que:
                //delete arrDouble;
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

arrDouble[42] = 42000

arrDouble[43] = 43000

arrDouble[44] = 44000

arrDouble[45] = 45000

arrDouble[46] = 46000

arrDouble[47] = 47000

arrDouble[48] = 48000

arrDouble[49] = 49000

Deletando array...

Exercício

Modifique o exemplo ArrHeap.cpp, de maneira que o programa crie no heap um array de estruturas (structs).

C++
Avançado

INTRODUÇÃO A CLASSES

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

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

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

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

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

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

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

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

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 PONTEIROS

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

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

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

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

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

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

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

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

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 ++

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`.

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.

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 `SobreO4.cpp`, de maneira que o operador`++` exiba uma mensagem na tela ao ser executado.

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.

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.

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.

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 `SobreO12.cpp`, para que o operador `++` seja usado nas duas posições, prefixo e sufixo.

SOBRECARREGANDO O OPERADOR +

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.

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 =

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

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.

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

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

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

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

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

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

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

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()
        // Superpõe (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

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

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

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

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

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

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()
//-------------------------------------------------
```