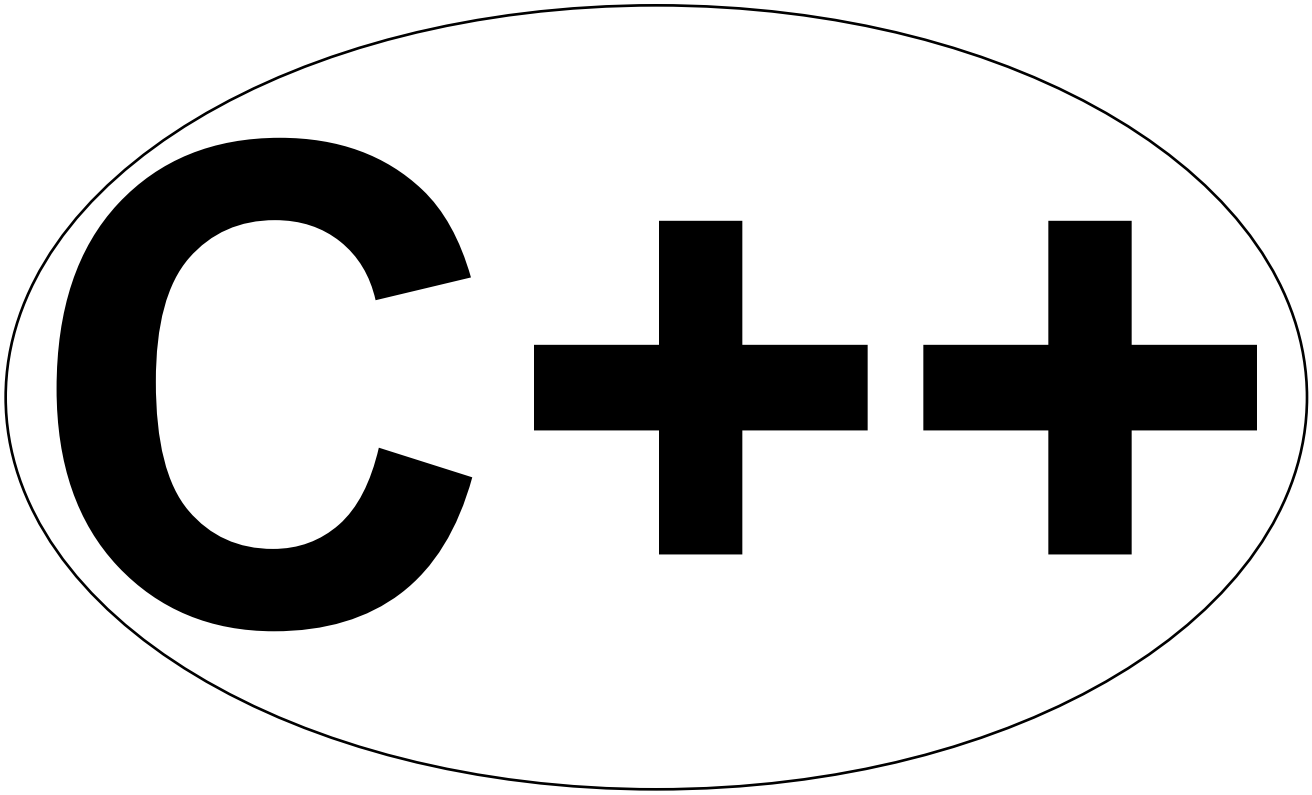

Curso Básico de C++

José Tarcisio Lopes Pereira

Este ebook é uma cópia do conteúdo respectivo no formato html que pode ser encontrado em www.tarcisiolopes.com e ou www.tarcisiolopes.com.br
Este e-book e grátis assim como a sua distribuição.

Curso C++ Básico
Introdução à sintaxe básica da linguagem C++. Como trata da linguagem C++. padrão ANSI/ISO, este curso se aplica a qualquer ambiente de desenvolvimento (DOS-Windows, Linux, Mac, outros).

Curso C++ Avançado
Apresenta a orientação a objetos e outros aspectos avançados de C++

Conversão e adaptação por:

...::MAROTO:::...

José Tarcisio Lopes Pereira

03 de Outubro de 2003

Sobre o Autor

José **Tarcisio Lopes** Pereira

Áreas de interesse: e-business, Internet, Java, Linux, Desenvolvimento de Software, Orientação a Objetos

Emprego atual: Technology Manager da divisão Solution Developer Marketing da IBM, para o Brasil e América Latina

Atividades na IBM:

- Janeiro, 2000 - Assumi uma nova posição dentro da IBM: Technology Manager da divisão Solution Developer Marketing, para o Brasil e América Latina. Responsabilidades do novo cargo:
  - Gerenciar, do ponto de vista tecnológico, o relacionamento entre a IBM e desenvolvedores de software selecionados;
  - Ajudar os desenvolvedores de software a viabilizar e otimizar suas soluções em plataformas IBM;
  - Divulgar as tecnologias IBM entre os desenvolvedores de software, enfatizando o conceito de e-business.

- Novembro, 1999 - Ajudei a incluir o IBM SanFrancisco e outros produtos de software e hardware da IBM em proposta para concorrência por um grande projeto (US$ 300 milhões a 500 milhões) do Ministério da Saúde do Brasil. O consórcio foi liderado pela parceira da IBM Eurosoft, e incluía a Siemens, além da própria IBM. O trabalho foi desenvolvido em conjunto com outros IBMers, entre os quais Paulo Huggler.
- Julho-Dezembro, 1999 - Juntamente com Suely Nishi de Abreu, Representante de Marketing do IBM SanFrancisco para a América Latina, ajudei a divulgar o conceito do IBM SanFrancisco, motivando vários novos parceiros para assinar o IBM SanFrancisco TLA.
- 1 a 3 de dezembro, 1999 - Participei de uma versão atualizada para e-business do Curso IBM Solution Selling. Trata-se de um workshop intensivo sobre o processo de vendas da IBM, ministrado pelo IBMer Jim Abrahams e pelo parceiro Edward Ryan.
- Agosto, 1999 - Participei do SanFrancisco Basics Course, um curso introdutório a soluções de objetos distribuídos baseadas em Java e IBM SanFrancisco, com uso intensivo de laboratórios. Ministrado pelos IBMers Lennart Frantzell e Richard Cook.
- Outubro, 1999 - Austin, Texas - Participei do SanFrancisco Application Development Workshop, um curso prático de 40 horas sobre desenvolvimento de aplicações e-business usando IBM SanFrancisco e IBM VisualAge for Java.
- Outubro, 1999 - Rochester, Minnesotta - Visitei a sede do desenvolvimento do IBM SanFrancisco, onde cumpri uma intensa agenda de reuniões com as equipes técnica e de marketing do IBM SanFrancisco.
- Julho, 1999 - Las Vegas, Nevada - Participei da Conferência Solutions'99, onde tive uma visão abrangente do conceito de IBM e-business do ponto de vista do desenvolvedor.
- Setembro-Novembro, 1999 - Ajudei Steve Alvarez, Gerente do SDM da IBM para a América Latina, na implementação de uma intensa agenda de cursos sobre ferramentas de desenvolvimento IBM no SPC para o quarto trimestre de 1999.
- 14 de dezembro, 1999 - Fiz uma apresentação sobre desenvolvimento de soluções e-business usando IBM WebSphere, Java e IBM SanFrancisco no Congresso OD'99, um importante evento sobre objetos distribuídos promovido pela Visionnaire, parceira da IBM .

Atividades pré-IBM:

- Trabalhei como consultor e instrutor das linguagens **C++** e **Java**. Algumas companhias atendidas: **Telefonica** (antiga **Telesp**), **Olivetti** e **Unimed**.
- Mantenho um Web site [www.tarcisiolopes.com] dedicado a divulgar as linguagens **C++** e **Java**, e tecnologias relacionadas, como **Orientação a Objetos**, **Tecnologias da Internet** e **Sistema Operacional Linux**.
- Trabalhei como consultor para as publicações **InformationWeek Brasil** [www.informationweek.com.br] e **NetworkComputing Brasil** [www.networkcomputing.com.br]. Meu trabalho envolvia a seleção, tradução e revisão de textos técnicos das publicações originais americanas.
- Desenvolvi software em linguagem **C++**. Desenvolvi principalmente sistemas para pequenas empresas.
- Em 1996/1997 escrevi para o jornal o **Estado de São Paulo** dois livros sobre informática para leigos: **Help! Informática** e **Help! Aplicativos**. Os livros foram publicados na forma de fascículos semanais distribuídos juntamente com o jornal. Como resultado, as vendas do jornal registraram significativo aumento nos dias de distribuição dos fascículos. Posteriormente, os direitos de publicação dessas obras foram adquiridos por outros importantes jornais, como **O Globo**, do Rio de Janeiro, e **Zero Hora**, de Porto Alegre.
- Entre 1993 e 1998, colaborei com as revistas **BYTE Brasil** e **LAN Times Brasil**. Como colaborador dessas publicações, fui pioneiro no estudo da **Tecnologia Java**, já a partir de seu lançamento em 1995. Analisei várias versões do **JDK** da **Sun Microsystems**. Desenvolvi exemplos de código que foram publicados nas revistas. Analisei e escrevi matérias sobre diversos ambientes e várias versões de desenvolvimento Java, como **IBM VisualAge for Java**, **Borland JBuilder**, **Symantec Visual Café** e **Microsoft Visual J++**.
- Escrevi um abrangente artigo sobre o **IBM SanFrancisco**, enfocando as vantagens da **Tecnologia Java** para desenvolvedores e empresas, bem como o papel da **IBM** na divulgação dessa tecnologia.
- Em 1996, participei de um dos primeiros cursos de **Linguagem Java** no Brasil, promovido pela **Sun Microsystems**.
- Como colaborador das publicações acima mencionadas, participei de inúmeras palestras, conferências e seminários, no Brasil e nos EUA, sobre assuntos relacionados com minha área de interesse: **C++**, **Java**, **Objetos** e **Internet**.

- Analisei e escrevi sobre os seguintes ambientes de programação **C++**: **Borland C++Builder**, **Microsoft Visual C++**, **Symantec C++**, **Borland C++**.
- Escrevi artigos dirigidos a programadores **C++** enfocando aspectos como: uso de som e outros recursos multimídia no ambiente **Windows**, vantagens da **Orientação a Objetos**, papel da **Análise e Projeto Orientado a Objetos**.
- Desenvolvi sistemas que foram distribuídos junto com a revista **BYTE Brasil** para dezenas de milhares de leitores.
- Participei de um curso introdutório ao **IBM VisualAge for Java**, ministrado por **Mr. Edward Tuggle**, na **IBM** em São Paulo.
- Tenho **Certificados de Proficiência em Inglês**, tanto da **Cambridge University** quanto da **Michigan University**. Sou fluente para ler, escrever, falar e traduzir Inglês.
- Em 1990, antes da popularização da multimídia, escrevi em **Linguagem C** um programa pioneiro para o ensino de Inglês com auxílio de computador.
- Em 1980, participei de um curso pioneiro no Brasil sobre **Microcomputadores e Microprocessadores**, que enfocava a arquitetura e a programação do microprocessador **Intel 8080/85**, bem como de seus circuitos integrados periféricos.
- Aprendi a programar em **Fortran**, nos anos 70. Meus primeiros programas foram desenvolvidos em um mainframe **IBM 1130** da Universidade onde estudei **Engenharia**.
- Tenho passaporte em dia e Visto de Negócios válido para os EUA.

Índice

Olá, Mundo!

Teoria

Uma tradição seguida em muitos cursos de programação é iniciar a aprendizagem de uma nova linguagem com o programa `"Olá, Mundo!"`. Trata-se de um programa elementar, que simplesmente imprime essa mensagem na tela.

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

`OlaMundo.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 `OlaMundo.cpp`. Simplesmente escreva o programa exatamente como é mostrado neste livro.

Para escrever um programa C++, você deve utilizar um editor de texto que possa guardar ficheiros em formato texto puro. Porém, ao invés de guardar os programas com a extensão `.TXT` no nome, guarde-os com a extensão `.CPP`. Geralmente, os editores de texto que acompanham os ambientes de desenvolvimento C++ são os melhores para esse tipo de trabalho. Porém nada o impede de utilizar 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 do 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 OlaMundo.cpp
```

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

Este comando faz com que seja gerado um ficheiro executável, neste caso Ola`Mundo.exe`.

Para executar o programa Ola`Mundo.exe`, basta digitar na linha de comando:

```
OlaMundo
```

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

Exemplo

```
// OlaMundo.cpp

// Um programa elementar.

#include <iostream.h>

int main()

{

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

        return 0;

} // Fim de main()
```

Saída gerada por este programa:

```
Ola, Mundo!
```

Exercício

Modifique o programa `OlaMundo.cpp`, de maneira que ele exiba na tela a seguinte frase

```
Ola, Portugal!
```

Exercício

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

```
Olá, Portugal!

Estamos aqui!
```

Cada frase deve aparecer em uma linha.

Apresentando cout

Teoria

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 o 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 << "Ola, 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:

```
Ola, 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

Teoria

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
```

```
        estende-se e estende-se

        por várias linhas, até

        que o par de caracteres

        de fechamento seja

        encontrado */

        cout << "Ola, Portugal!\n";

        // Comentários neste estilo

        // vão somente até o

        // final da linha.

        cout << "Ola, Brasil!\n";

} // Fim de main()
```

Saída gerada por este programa:

```
Ola, Portugal!

Ola, Brasil!
```

Exercício

Acrescente um comentário inicial ao programa `Coment.cpp`. O comentário inicial deve conter o nome do programa, o nome do ficheiro 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

Teoria

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 realçar 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 que se segue cria variáveis dos tipos básicos e exibe os 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

Teoria

O tipo `char` (caractere) geralmente tem o tamanho de um byte (8 bits), 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

Teoria

Alguns caracteres podem ser representados por combinações especiais de outros caracteres. Essas combinações são conhecidas como sequências 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 sequências de escape mais comuns:

Sequência 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 sequências

 // de escape.

 #include <iostream.h>

 int main()

 {

         // Exibe frases

         // usando sequências 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

Teoria

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

Esse número sequencial 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`, subsequentes.

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 a variáveis

Teoria

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

Teoria

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

Teoria

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 uma variável signed

Teoria

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

Teoria

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 Ola`Mundo.cpp`, usando o tipo `string`.

Exemplo

```
// OlaStr.cpp

// Ilustra o uso

// do tipo string.

#include <iostream.h>

int main()

{

        // Declara e inicializa

        // uma variável do
```

```
        // tipo string.

        string aloTar = "Ola, Tarcisio!";

        // Exibe a string.

        cout << aloTar;

} // Fim de main()
```

Saída gerada por este programa:

```
Ola, Tarcisio!
```

Exercício

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

Funções

Teoria

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, `OlaFunc.cpp` contém duas funções: `main()` e `digaOla()`. 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 `OlaFunc.cpp` (e de qualquer outro programa C++) começa no começo da função `main()` e termina quando a função `main()` é encerrada.

Exemplo

```
// OlaFunc.cpp
```

```
// Ilustra uma

// função elementar.

#include <iostream.h>

// Definição da função

// digaOla()

void digaOla()

{

        cout << "\nOla, Mundo!";

} // Fim de digOla()

int main()

{

        // Chama a função

        // digaOla()

        digaOla();

        return 0;

} // Fim de main()
```

Saída gerada por este programa:

```
Ola, Mundo!
```

Exercício

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

```
Ola, Mundo!

Tchau!
```

Chamando uma função

Teoria

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

Teoria

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

Teoria

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

Teoria

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

Teoria

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

Teoria

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

Teoria

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 * SEGUNDOS_POR_MINUTO *

HORAS_POR_DIA *DIAS_POR_ANO;
```

Dica: velocidade da luz = 300.000 Km/s.

Constantes enumeradas

Teoria

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 subsequentes 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

Teoria

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

Teoria

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

Teoria

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

Teoria

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

Teoria

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

Teoria

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

Teoria

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`.

Indentação

Teoria

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

Teoria

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

Teoria

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

Teoria

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 ficheiro 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

Teoria

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

Teoria

As variáveis definidas fora de qualquer função têm escopo global e portanto estão disponíveis para qualquer função do programa, inclusive `main()`.

Se criarmos uma variável local com o mesmo nome de uma variável global já existente, isso não afeta a variável global. Porém, a variável local oculta a variável global. Ou seja, se uma função tem uma variável local com o mesmo nome de uma variável global, quando o nome for usado dentro da função ele refere-se à variável local, e não à variável global.

Exemplo

```
// Global.cpp

// Ilustra o uso

// de variáveis globais.

#include <iostream.h>

// Protótipo.

void UmaFuncao();

// Variáveis globais.

int var1 = 10, var2 = 20;

int main()
```

```
{
        cout << "\nEstamos em main().\n";
        cout << "var1 = " << var1 << "\n";
        cout << "var2 = " << var2 << "\n";
        // Chama a função.
        UmaFuncao();
        cout << "\nDe volta a main().\n";
        cout << "var1 = " << var1 << "\n";
        cout << "var2 = " << var2 << "\n";
        return 0;
} // Fim de main()
// Definição.
void UmaFuncao()
{
        // Variável local.
        int var1 = 30;
        cout << "\nEstamos em UmaFuncao().\n";
        cout << "var1 (local) = " << var1 << "\n";
        cout << "var2 = " << var2 << "\n";
} // Fim de UmaFuncao()
```

Saída gerada por este programa:

```
Estamos em main().

var1 = 10

var2 = 20

Estamos em UmaFuncao().

var1 (local) = 30

var2 = 20

De volta a main().

var1 = 10

var2 = 20
```

Exercício

Rescreva o programa `Global.cpp`, de maneira que o valor da variável global `var2` seja alterado dentro de `UmaFuncao()`.

Variável dentro de um bloco

Teoria

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()
```

Eis a 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

Teoria

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

Teoria

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

Teoria

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

Teoria

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

Teoria

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

Teoria

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 sequência 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 subsequente 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.

Apalavra-chave goto

Teoria

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

Teoria

O loop `while` faz com que o programa repita uma sequência 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

Teoria

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

Teoria

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

Teoria

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

Teoria

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

Teoria

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

Teoria

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

Teoria

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

Teoria

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

Teoria

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

Teoria

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

Teoria

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 subsequentes 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

Teoria

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

Teoria

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

Teoria

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

Teoria

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

Teoria

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.

Arrays de caracteres

Teoria

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

Porém no nível mais fundamental, uma string é uma sequência 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 arrays de caracteres

Teoria

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

Teoria

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

Teoria

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 sequencialmente. 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

Teoria

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

Teoria

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

Teoria

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

Teoria

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 arrys

Teoria

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

Teoria

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

Teoria

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()

Teoria

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()

Teoria

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

Teoria

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

Teoria

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

Teoria

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

Teoria

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

Teoria

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

Teoria

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

Teoria

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

Teoria

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

Teoria

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

Teoria

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

Teoria

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

Teoria

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

Teoria

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

Teoria

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

Teoria

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

Teoria

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

Teoria

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`).