Curso Prático de C++

Autor: Prof. Luiz Bianchi
Universidade Regional de Blumenau

Sumário

FUNDAMENTOS

C++ é uma linguagem de programação derivada da linguagem C. Assim, a linguagem C é um subconjunto de C++. Os principais elementos criados para dar origem a C++ estão relacionados à programação orientada a objetos.

O processo evolutivo da linguagem C++ começa com a linguagem BCPL, a partir da qual surge em 1970 a linguagem chamada B, desenvolvida por Dennis Ritchie. B é aprimorada e a nova versão dá origem a linguagem C, em 1978. Em 1992, a linguagem C é incrementada para dar suporte à programação orientada a objetos surgindo, desta feita, C++.

Um programa em C++ consiste de uma ou mais funções as quais se constituem nas unidades fundamentais da linguagem.

Todo programa deve ter no mínimo uma função principal denominada **main**. A execução do programa começa na função **main** e termina quando esta função for encerrada.

Estrutura básica de um programa:

```
#include <arquivo de inclusão>
void main( )
{
        bloco de comandos;
}
```

O bloco de comandos de toda função C++ deve começar com uma chave de abertura (**{**) de bloco e terminar com uma chave de fechamento de bloco (**}**).

Em C++, existem comandos que são pré-processados os quais não se constituem propriamente em instruções da linguagem C++ e são chamados de **diretivas de compilação** ou simplesmente **diretivas**. O pré-processador é um programa que examina o programa-fonte e executa os comandos genericamente chamados de diretivas. Toda diretiva é iniciada pelo símbolo #. A diretiva **#include** provê a inclusão de um arquivo da biblioteca de programas contendo definições e declarações que são incorporadas ao programa pelo pré-processador. Em outras palavras, o pré-processador C++ substitui a diretiva include pelo conteúdo do arquivo indicado antes de o programa ser compilado.

Por exemplo, os arquivos de cabeçalhos (header) **iostream.h** e **conio.h** permitem que se faça a utilização de diversos comandos de entrada e saída no programa, ou seja, tais arquivos servem para auxiliar no desenvolvimento do programa-fonte. O arquivo **iostream.h**, por exemplo, contém declarações necessárias ao uso do objeto **cin** e **cout**, entre outros.

Declaração de variáveis

Uma variável é um espaço de memória reservado para armazenar num certo tipo de dado e que possui um nome para referenciar seu conteúdo. Uma variável em C++ pode ser declarada em qualquer lugar no programa, no entanto ela deve obrigatoriamente ser declarada antes de ser usada no programa.

Um nome de variável pode ter qualquer número de caracteres, sendo que o primeiro caractere deve, obrigatoriamente, ser uma letra ou um caractere de sublinhado. Pode conter letras minúsculas e maiúsculas, os dígitos de 0 a 9 e o caractere de sublinhado. C++ considera somente os primeiros 32 caracteres do nome da variável, caso ela seja declarada com mais de 32 caracteres. C++ é sensível a letras maiúsculas e minúsculas, ou seja, as letras maiúsculas são consideradas diferentes das minúsculas, por exemplo, A é diferente de a. O nome da variável não pode coincidir com uma designação pré-definida chamada de palavra-chave ou palavra reservada que é de uso restrito da linguagem.

Tabela de palavras reservadas:

asm	auto	break	case	catch
_cdecl	cdecl	char	class	const
continue	_cs	default	do	double
_ds	else	enum	_es	_export
extern	_far	far	float	for
friend	goto	huge	if	inline
int	interrupt	_loadds	long	_near
near	new	operator	_pascal	pascal
private	protected	public	register	return
_saveregs	_seg	short	signed	sizeof
_ss	static	struct	switch	template
this	typedef	union	unsigned	virtual
void	volatile	while		

Tipos de dados

O tipo de uma variável determina a quantidade de memória que ela ocupará, em bytes, e o modo de armazenamento. C++ possui cinco tipos básicos que são:

Tipo	Tamanho em bytes	Escala de valores
char	1	-128 a 127
int	2	-32768 a 32767
float	4	3.4E-38 a 3.4E+38
double	8	1.7E-308 a 1.7E+308
void	0	nenhum valor

Operador de atribuição

O operador de atribuição é representado por = (sinal de igualdade). Atribui a expressão à direita do sinal de igualdade à variável a sua esquerda.

Exemplo:

```
x = 5;
```

é atribuído o valor 5 à variável de nome x. Outros exemplos:

```
y = 4 + 3;
z = x = 8;
```

esta última expressão, chamada de **atribuições múltiplas**, pode também ser escrita como segue:

```
z = (x = 8);
```

à variável x e z é atribuído o valor 8.

Comentários

Comentários são utilizados com a finalidade de documentar o programa-fonte. Eles não são tratados pelo compilador. Os símbolos utilizados para representar comentários inseridos no programa são os delimitados por /* e */ e o iniciado com duas barras (//) e terminado com o final da linha, conhecido como "comentário de linha".

O exemplo, a seguir, apresenta a declaração de variáveis, o operador de atribuição e comentários:

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *   Este programa mostra o uso de variáveis, do operador de atribuição           *
 *   e de comentários.                                                            *
 * * * ** * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * ** * * * * * * * * * **/
#include <iostream.h>

void main( )
{
        int num_int;            // declaração de variável do tipo int
        float num_real;         // declaração de variável do tipo float

        num_int = 20;           // atribuição do valor 20 à variável de nome num_int
        num_real = 100.43;      // atribuição do valor 100.43 à variável de nome num_real

        cout << "\nNumero inteiro = " << num_int;
        cout << "\nNumero real     =  " << num_real;
}
```

resultado da execução do programa:

```
Numero inteiro = 20
Numero real    = 100.43
```

Tipos modificados

Os tipos básicos podem ser acompanhados de modificadores na declaração de variáveis, exceto o tipo void. Os modificadores de tipo são: **long**, **short** e **unsigned**.

A tabela, a seguir, mostra os tipos com os modificadores:

Tipo	Tamanho em bytes	Escala de valores
unsigned char	1	0 a 255
unsigned	2	0 a 65535
short	2	-32768 a 32767
long	4	-2147483648 a 2147483647
unsigned long	4	0 a 4294967295
long double	10	3.4E-4932 a 1.1E+4932

O tamanho e a escala de valores podem variar segundo o processador ou compilador em uso. Os valores apresentados acima estão de acordo com o padrão ANSI.

```
// mostra o modificador de tipo de dado
#include <iostream.h>
void main( )
{
        unsigned int i = 65535;
        cout << "\nVariavel unsigned int = " << i;
        short int j = i;
        cout << "\nVariavel short int = " << j;
}
```

Resultado do exemplo:

```
        Variavel unsigned int = 65535
        Variavel short int = -1
```

Constantes

Uma variável declarada por meio do qualificador **const** significa que seu conteúdo não poderá ser alterado em todo programa. A constante deve ser inicializada, isto é, no momento de sua declaração deverá ser atribuído um valor a ela.

Exemplo:

```
const float pi = 3.1415;
```

Comandos de entrada

Os comandos de entrada recebem a entrada de dados inserida pelo usuário através do teclado. O comando **cin** trata a entrada de dados por meio do **operador de extração** >> (maior que duplos) que direciona os dados à variável designada na instrução.

Exemplo:

```
// mostra a utilização do comando de entrada cin
#include <iostream.h>

void main( )
{
        int anos;
        cout << "\nDigite sua idade em anos: ";
        cin >> anos;
        cout << "\nIdade em meses: " << (anos * 12);
}
```

Resultado do exemplo::

```
        Digite sua idade em anos: 20
        Idade em meses: 240
```

Para a leitura de textos contendo espaços em branco, por exemplo, é conveniente utilizar a função **gets( )** , cujas declarações para seu uso se encontram no arquivo de inclusão **stdio.h**. Contudo ela deve ser precedida do manipulador **flush** utilizado no objeto cout para liberar o conteúdo do buffer de saída carregado na instrução **cout** anterior a chamada à função.

Exemplo:

```
// mostra a utilização da função gets()
#include <iostream.h>
#include <stdio.h>

void main( )
{
        char nome[50];
        cout << "\nDigite seu nome: "<<flush;
        gets(nome);                             // para leitura do nome com espaço
        cout << "\nSeu nome é " << nome;
}
```

Resultado da execução do programa:

```
Digite seu nome: Luiz Bianchi
Seu nome é Luiz Bianchi
```

As funções **getche( )** e **getch( )** são destinadas a ler um caractere digitado no teclado sem a necessidade de se pressionar a tecla Enter.

Exemplo:

```
// mostra a utilização da função getche()
#include <iostream.h>
#include <conio.h>

void main( )
{
        char letra;
        cout << "\nPressione uma letra: ";
        letra = getche();                       // faz a leitura sem esperar Enter
        cout << "\nLetra pressionada " << letra;
}
```

A diferença entre **getche( )** e **getch( )** é que esta última não registra o caractere pressionado no vídeo. Diz-se que a função **getch( )** não ecoa no vídeo. Ambas as funções necessitam do arquivo de cabeçalho **conio.h**.

Comandos de saída

O comando de saída mais utilizado para exibir os dados na tela ou na impressora é o **cout**, cuja saída ou mensagem é conectada a este através do operador de inserção << (menor que duplos). As definições para seu uso estão contidas no arquivo de cabeçalho **iostream.h**.

Sua utilização já foi experimentada nos programas exemplos anteriores.

Caracteres de controle

A seguinte tabela apresenta os caracteres utilizados para controle de páginas e exibição de códigos especiais:

Caractere de controle	Significado
\n	nova linha (CR+LF)
\t	tabulação
\f	salto de página
\b	retrocesso
\a	aciona o Beep
\r	início da linha
\\	imprime barra invertida (\)
\’	imprime aspa simples
\”	imprime aspa dupla
\xdd	representação hexadecimal

A instrução **endl** (end line) encerra a linha de dados e coloca o cursor no início da próxima linha. Tem, portanto, a mesma função do caractere de controle **\n**.

Constantes numéricas

Uma constante é representada por um valor fixo que não pode ser alterado ao longo da execução do programa. Constantes numéricas em C++ podem ser escritas nos seguintes sistemas:

Decimal

Escreve-se na forma habitual, por exemplo:

40, 5890, 29.

Octal

Os números desse sistema numérico devem ser escritos antepondo-se um zero. Exemplos:

010, 042, 0500.

Hexadecimal

Os números do sistema de base 16 devem ser escritos precedidos dos caracteres 0x. Exemplos:

0x36, 0x4df, 0xa3b1.

Observação:

Um número iniciando com zero é considerado um número do sistema numérico de base 8. Portanto, o número 010 é diferente 10. O número 010 equivale ao número 8 em decimal.

String de caracteres

É uma cadeia de caracteres delimitada por aspas duplas. Um exemplo de cadeia de caracteres pode ser escrito como segue: “Programa de computador”.

Em C++, as aspas simples são utilizadas para representar um único caractere. Exemplos: ‘w’, ‘X’, ‘4’ e, também os caracteres de controle: ‘\n’, ‘\t’, ‘\r’.

Formato da impressão

O tamanho de um campo de dados pode ser definido no comando **cout** para permitir o alinhamento na impressão ou apresentação dos dados. O arquivo de inclusão **iomanip.h** define os manipuladores de alinhamento e são os seguintes:

setw determina o tamanho do campo a ser exibido;

setfill estabelece o caractere de preenchimento do espaço em branco;

setprecision determina a quantidade de casas decimais dos números reais ou de ponto flutuante (float);

A seguir são mostrados dois programas: um sem definições de comprimento de campos e outro que utiliza manipuladores de tamanho de campos:

```
#include <iostream.h>
void main()
{
        int ban=23, per=1200, lar=450, uva=14530;
        cout << '\n' << "Bananas  " << ban;
        cout << '\n' << "Peras       " << per;
        cout << '\n' << "Laranjas  " << lar;
        cout << '\n' << "Uvas       " << uva;
}
```

Resultado do exemplo:

```
        Bananas     23
        Peras       1200
        Laranjas    450
        Uvas        14530
```

```
#include <iostream.h>
#include <iomanip.h>
void main()
{
        int ban=23, per=1200, lar=450, uva=14530;
        cout << '\n' << "Bananas  " << setw(10) << ban;
        cout << '\n' << "Peras       " << setw(10) << per;
        cout << '\n' << "Laranjas  " << setw(10) << lar;
        cout << '\n' << "Uvas       " << setw(10) << uva;
}
```

Resultado do exemplo::

```
        Bananas        23
        Peras        1200
        Laranjas      450
        Uvas        14530
```

A seguir, um exemplo com preenchimento dos espaços em branco entre a descrição e quantidade citada:

```
#include <iostream.h>
#include <iomanip.h>
void main()
{
        int ban=23, per=1200, lar=450, uva=14530;
        cout << setfill('.');
```

```
        cout << '\n' << "Bananas  " << setw(10) << ban;
        cout << '\n' << "Peras    " << setw(10) << per;
        cout << '\n' << "Laranjas " << setw(10) << lar;
        cout << '\n' << "Uvas     " << setw(10) << uva;
}
```

Resultado do exemplo:

```
        Bananas . . . . . .23
        Peras     . . . . . .1200
        Laranjas . . . . . .450
        Uvas     . . . . . . 14530
```

O manipulador **setprecision** estabelece arredondamento e a quantidade de casas decimais a serem exibidas por um número de ponto flutuante.

Para inibir a impressão em notação científica das variáveis declaradas como tipo de dado **float**, utiliza-se o flag **fixed** da classe **ios**, definido no arquivo **iostream.h**. Segue um exemplo:

```
#include <iostream.h>
#include <iomanip.h>
void main()
{
        float ban=23.375, per=1200.060, lar=450.7, uva=14530.85;
        cout << setprecision(2);                // exibe duas casas decimais
        cout << setiosflags(ios::fixed);        // inibe a exibição em notação científica
        cout << '\n' << "Bananas  " << setw(10) << ban;
        cout << '\n' << "Peras      " << setw(10) << per;
        cout << '\n' << "Laranjas  " << setw(10) << lar;
        cout << '\n' << "Uvas       " << setw(10) << uva;
}
```

Resultado do exemplo:

```
        Bananas         23.38
        Peras         1200.06
        Laranjas       450.70
        Uvas         14530.85
```

Sistemas numéricos

O objeto **cout** permite exibir um número na base numérica especificada no programa. Os manipuladores de bases numéricas encontram-se definidos no arquivo **iostream.h** e são eles:
dec para o sistema decimal (default), **oct** para o sistema octal e **hex** para o hexadecimal.

Exemplo:

```
#include <iostream.h>
void main()
{
        int n=53;
        cout << '\n' << "Decimal:       " << dec << n;
        cout << '\n' << "Octal     :       " << oct << n;
        cout << '\n' << "Hexadecimal: " << hex << n;
}
```

Resultado do exemplo:

```
        Decimal:        53
        Octal:          65
        Hexadecimal:  35
```

O objeto **cin** também permite estabelecer a base numérica em que se deseja inserir um número:

```
#include <iostream.h>
void main()
{
        int n;
        cout << '\n' << "Insira um numero hexadecimal: ";
        cin >> hex >> n;
        cout <<"O nro.hexadecimal "<<hex<<n<<" equivale ao decimal "<<dec<<n;
}
```

OPERADORES

Operadores aritméticos

Existem cinco operadores aritméticos binários (que operam sobre dois operandos) e um operador aritmético unário (que opera sobre um operando). O operador (**%**), chamado operador de **módulo**, apresenta como resultado o resto da divisão e opera somente com operandos inteiros. Cada operador representa uma operação aritmética elementar: soma, subtração, divisão, multiplicação e módulo.

Os símbolos e as respectivas operações dos operadores aritméticos são:

Operador binário	Operação
+	soma
-	subtração
*	multiplicação
/	divisão
%	módulo
-	menos unário

Exemplo do operador que retorna o resto da divisão:

```
18 % 4
```

resulta no valor 2 (18 dividido por 4, restam 2).

Para efetuar a operação de potenciação, C++ conta com uma função pré-definida: **pow( )**.

Exemplo:

```
pow(4,2)
```

representa $4^2 = 16$.

Operadores de incremento e de decremento

Os operadores de incremento (++) e de decremento (--) são operadores unários que adicionam e subtraem uma unidade do conteúdo da variável respectiva. Existe duas formas: o operador pode ser escrito a direita ou à esquerda do nome da variável, em outras palavras, o operador pode ser prefixado ou pós-fixado à variável.

Sintaxe dos operadores:

Instrução	equivalência
var++	var = var + 1
++var	var = var + 1
var--	var = var – 1
--var	var = var – 1

Em qualquer caso, o conteúdo da variável será incrementado (ou decrementado) de uma unidade. Todavia, se o operador for pós-fixado, o valor da variável será incrementado (ou decrementado) depois da execução da instrução de que ela faz parte. Caso o operador seja prefixado, o valor da variável será incrementado (ou decrementado) antes da execução da instrução de que ela faz parte.

Operadores aritméticos de atribuição

Estes operadores combinam as operações aritméticas com atribuição. Os símbolos utilizados são:

+=, -=, *=, /= e %=.

Exemplos:

Aritmética de atribuição	Instrução equivalente
i += 5;	i = i + 5;
j -= x;	j = j – x;
k *= y + 2;	k = k * (y + 2);
m /= 4.2;	m = m / 4.2;
n %= 2;	n = n % 2;

Operadores relacionais

Operadores relacionais fazem comparações, ou seja, verificam a relação de magnitude e igualdade entre dois valores. São seis os operadores relacionais:

Operador	Significado
>	maior
>=	maior ou igual
<	menor
<=	menor ou igual
= =	igual
!=	diferente

Estes operadores permitem avaliar relações entre variáveis ou expressões para a tomada de decisões.
Ao contrário de outras linguagens, não existe um tipo de dado lógico que assuma um valor verdade ou falsidade. Qualquer valor diferente de zero é considerado verdade e zero, falsidade.

O resultado de uma expressão lógica é um valor numérico: uma expressão avaliada **verdadeira** recebe o valor 1, uma expressão lógica avaliada **falsa** recebe o valor 0.

Exemplo:

```
#include <iostream.h>
void main()
{
        int x = 12, y = 20, verdadeiro, falso;

        verdadeiro = (x < y);
        falso = (x = = y);

        cout << "\nVerdade:   " << verdadeiro;
        cout << "\nFalsidade: " << falso;
}
```

Resultado do exemplo::

```
        Verdade:   1
        Falsidade: 0
```

Operadores lógicos

Os três operadores lógicos são:

Operador	Significado	Descrição
&&	e	conjunção
||	ou	disjunção
!	não	negação

Os operadores && e || são binários e o operador ! é unário. Estes operadores avaliam os operandos como lógicos (0 ou 1), sendo o **valor lógico 0** considerado **falso** e o **valor lógico 1**, **verdadeiro**.

As tabelas-verdade, a seguir, expressam melhor essas operações lógicas:

&& (conjunção)	|| (disjunção)	! (negação)
0 e 0 = 0	0 ou 0 = 0	não 0 = 1
0 e 1 = 0	0 ou 1 = 1	não 1 = 0
1 e 0 = 0	1 ou 0 = 1	
1 e 1 = 1	1 ou 1 = 1	

Exemplos:

Admitindo que os valores das variáveis x, y e z do tipo int sejam, respectivamente, 0, 1 e 2, são realizadas as avaliações das expressões da tabela abaixo, suscitando os seguintes resultados:

Expressão	Resultado
!7	falsidade
x && y	falsidade
x > z || z = = y	falsidade
x + y && !z – y	verdade

```
#include <iostream.h>
void main()
{
        int x=5, y=6, expr1, expr2, resul;
        expr1=(x==9);
        expr2=(y==6);
        resul=(expr1 && expr2);
        cout<<"\nExpressao1: "<<expr1;
        cout<<"\nExpressao2: "<<expr2;
        cout<<"\n\nResultado:  "<<resul;
}
```

Resultado do exemplo:

```
        Expressão1:  0
        Expressão2:  1
        Resultado:   0
```

Operador condicional

O operador condicional (?:) opera sobre três expressões. A sintaxe de uma expressão condicional é:

```
condição ? expr1 : expr2
```

condição é uma expressão lógica que será avaliada primeiro. Se o valor for diferente de zero, isto é, verdadeiro, a **expr1** será o valor da condicional. Caso contrario a **expr2** será o valor da condicional.

Uma expressão condicional é equivalente a uma estrutura de decisão simples:

```
        if (condição)
                expr1
        else
                expr2
```

Exemplos:

Admitindo que x, y e z são variáveis do tipo int contendo respectivamente os valores 1, 2 e 3, são procedidas as avaliações das expressões da tabela a seguir e mostrados seus resultados:

Expressão	Valor
x ? y : z	2
x > y ? y : z	3
y > x ? ++z : --z	4
x >= z ? z : --y	1

Precedência dos operadores

A seguir, é apresentada a tabela de precedência dos operadores estudados, organizados em blocos. Os operadores de cada bloco possuem a mesma prioridade e são executados na ordem em que são escritos na instrução da esquerda para direita:

Bloco	Categoria	Operador	Tipo
1	negativo, incremento/ decremento e lógico	-	menos unário
		++	incremento
		--	decremento
		!	não lógico
2	aritmético	*	multiplicação
		/	divisão
		%	módulo
3	aritmético	+	adição
		-	subtração
4	relacional	<	menor
		<=	menor ou igual
		>	maior
		>=	maior ou igual
5	relacional	= =	igual
		!=	diferente
6	lógico	&&	e lógico
7	lógico	| |	ou lógico
8	condicional	?:	condicional
10	atribuição e aritmético de atribuição	=	atribuição
		*=	multiplicação
		/+	divisão
		%=	módulo
		+=	adição
		-=	subtração

Os operadores aritméticos atuam sobre operandos numéricos e produzem resultados também numéricos. Os operadores relacionais avaliam os operandos numéricos e produzem resultados lógicos. E os operadores lógicos avaliam operandos lógicos e apresentam resultados igualmente lógicos.

ESTRUTURAS DE CONTROLE

Muito do poder real de um computador vem de sua habilidade em tomar decisões e determinar um curso de ação durante a corrida de um programa. As estruturas de controle permitem controlar o curso das ações lógicas. Os dois tipos básicos de estruturas de controle são as estruturas de **seleção** e as estruturas de **repetição**. A estrutura de seleção ou decisão permite executar um entre dois ou mais blocos de instruções. A estrutura de repetição permite que a execução de um bloco de instruções seja repetida um determinado número de vezes.

Condição de controle

Uma condição de controle é representada por uma expressão lógica ou aritmética que controla qual bloco de comandos a executar e quantas vezes deverá ser executado. Como C++ não possui variáveis lógicas, quando uma expressão assume uma condição de controle ela será considerada **verdadeira** se seu valor for diferente de zero e **falsa** se seu valor for igual a zero.

Exemplo: O quadro a seguir é analisado, admitindo-se que os valores das variáveis inteiras i e j sejam respectivamente 0 e 3:

Condição	Valor numérico	Significado lógico
(i = = 0)	1	verdadeiro
(i > j)	0	falso
i	0	falso
j	3	verdadeiro

Estrutura de seleção

A seleção ou decisão é a habilidade que o sistema possui para escolher uma ação de um conjunto de alternativas específicas. Isto significa que se pode selecionar entre ações alternativas dependendo de critérios de uma expressão condicional.

Seguem os comandos de seleção:

Comando	Estrutura
if	simples
if...else	composta
switch	seletiva

Estrutura simples (if)

A estrutura de seleção simples permite que se execute (ou não) um bloco de instruções conforme o valor de uma condição seja verdadeiro ou falso. O bloco pode ter uma única instrução ou várias instruções entre chaves:

Sintaxe:

Bloco	
com uma instrução	**com n instruções**

if (condição)
  instrução;
	if (condição)
	{
	  instrução;
	  instrução;
	}

Se a condição for **verdadeira**, o *bloco* de instruções é executado. Caso contrário, o bloco não é executado. Observe que quando o bloco possui várias instruções, elas devem estar entre chaves.

Exemplo:

```
/*      O programa lê dois valores numéricos, efetua a soma
        e apresenta a mensagem "Soma maior que dez",
        caso o resultado da adição seja maior que 10.
*/
#include <iostream.h>
void main()
{
        int n1, n2, soma=0;
        cout<<"\nInsira dois numeros inteiros: ";
        cin>>n1>>n2;                    // leitura dos dois valores
        soma=n1+n2;
        if (soma > 10)                  // expressão de teste (condição)
                cout<<"Soma maior que dez";
}
```

Estrutura composta (if...else)

O comando **if...else** executa um entre dois blocos de instruções. Se a condição resultar em verdade o programa executa o primeiro bloco de instruções; se resultar em falsidade, executa o segundo bloco de instruções.

Sintaxe:

Bloco	
com uma instrução	**com n instruções**

if (condição)
  instrução;
else
  instrução;
	if (condição)
	{
	  instrução;
	  instrução;
	}
	else
	{
	  instrução;
	  instrução;
	}

Exemplo:

```
/*      O programa efetua o cálculo da média aritmética das notas
        de três provas e avalia a situação quanto à aprovação.
*/
#include <iostream.h>
void main()
{
        float n1, n2, n3, media;
        cout<<"\nIntroduza as três notas: ";
        cin>>n1>>n2>>n3;                    // leitura das três notas
        media=(n1+n2+n3)/3;                 // cálculo da média
        if (media >= 6.0)                   // expressão de teste (condição)
            cout<<"Aprovado";
        else
            cout<<"Reprovado";
}
```

Estrutura seletiva (switch)

O comando switch permite executar um conjunto de instruções, selecionado dentre vários casos rotulados por uma constante, conforme o valor de uma variável ou de um número constante escrito entre parentes.

Sintaxe:

```
switch (variável ou constante)
{
        case valor1:
                instrução;
                instrução;
                break;
        case valor2:
                instrução;
                instrução;
                break
                        . . .
        default:
                instrução;
                instrução;
}
```

Cada caso pode ser composto por qualquer número de instruções, seguidas pelo comando **break** para impedir a execução da instrução ou instruções dos casos subseqüentes. O comando default será executado somente quando o valor da variável do comando switch não calhar com os valores mencionados nos casos.

Exemplo:

```
/* programa que simula uma calculadora eletrônica, efetuando
   uma das operações básicas por vez a critério do usuário.
*/
#include <iostream.h>
void main()
{
        float n1, n2;
        char op;
        cout<<"\nIntroduza a operação (1o.numero operador 2o.numero): ";
        cin>>n1>>op>>n2;
        switch (op)
        {
                case '+':
                    cout<<"Soma: "<<n1 + n2;
                    break;
                case '-':
                    cout<<"Subtração: "<<n1 - n2;
                    break;
                case '*':                   // símbolos opcionais (* ou x) para
                case 'x':                   // o usuário comandar a multiplicação.
                    cout<<"Multiplicação: "<<n1 * n2;
                    break;
                case '/':                   // símbolos opcionais (/ ou :) para
                case ':':                   // o usuário comandar a divisão.
                    cout<<"Divisão: "<<n1 / n2;
                    break;
                default:
```

```
                cout<<"Operador invalido";
        }
}
```

Segue um exemplo de execução do programa:

```
Introduza a operação (1o.numero operador 2o.numero):  12 x 5 [ENTER]
Multiplicação:  60
```

Observação: O comando **if...else** poderia ser usado para programar a solução acima. No entanto, para avaliar uma expressão única com várias ações possíveis, o comando **switch** apresenta um formato mais adequado e claro.

Estrutura de repetição

A repetição é uma poderosa habilidade, ao lado da decisão, que o sistema tem para repetir um conjunto de ações específicas. A linguagem algorítmica

Seguem as três formas de repetição de C++:

for
while
do...while

Estrutura for

Esta estrutura de repetição é utilizada quando se conhece de antemão o número de vezes que um conjunto de instruções deve ser repetido.

sintaxe:

```
for(inicialização; condição; incremento ou decremento)
{
        instrução;
        instrução;
}
```

a **inicialização** corresponde a uma instrução que atribui um valor inicial à variável que controla o número de repetições.

a **condição** refere-se a uma expressão que é avaliada toda vez que a repetição é reiniciada e quando essa avaliação resultar em falso a repetição é encerrada.

o **incremento** ou **decremento** define como a variável de controle será alterada após cada execução dos comandos de repetição.

Exemplo:

```
for(i = 0; i < 10; i++)
    cout << "***";
```

Neste exemplo, o valor da variável de controle **i** é inicializada em 0, em seguida ela é testada para determinar se é menor que 10 e incrementada de uma unidade cada vez que a instrução **cout << "***"** é executada. Esse processo de avaliação, execução e incremento é repetido até o valor da variável **i** for igual a 10, momento em que o resultado da avaliação apresenta valor falso (igual a zero). Quando **i** passa a valer 10 a instrução sob o comando **for** não é executada. Na última execução o valor de **i** é igual a 9.

Outros exemplos:

```
//  Exibe a tabuada de 1 a 10 do número fornecido pelo usuário.
#include <iostream.h>
#include <iomanip.h>
void main()
```

```
{
        int n, i;
        cout << "\nInsira um numero: ";
        cin >> n;
        for(i = 1;  i <= 10; i++)
             cout << '\n' << setw(2) << i << " x " << n << " = " << setw(2) << (i*n);
}
```

Resultado da execução do programa:

```
  Insira um numero:  8

   1 x  8 =   8
   2 x 8 =  16
   3 x 8 =  24
   4 x 8 =  32
   5 x 8 =  40
   6 x 8 =  48
   7 x 8 =  56
   8 x 8 =  64
   9 x 8 =  72
  10 x 8 =  80
```

```
//   mostra os números múltiplos de 5 até o limite de 100.
#include <iostream.h>
void main( )
{
      for(int i = 5; i <= 100; i += 5)
          cout << i << '\t';
}
```

Note que a variável **i** foi declarada no interior do comando **for**. Em C++, é prática comum declarar as variáveis a serem utilizadas, imediatamente, antes ou próximas do local em que serão usadas.

Estrutura while

A repetição **while** é utilizada quando não se conhece previamente o número de iterações que serão realizadas. O bloco de instruções será repetido até que a expressão de teste condicional for falsa (igual a zero).

Sintaxe:

```
while(condição)
{
        instrução;
        instrução;
}
```

Enquanto a condição for verdadeira, as instruções do bloco serão executados. O bloco que contém várias instruções deve estar delimitado por chaves (**{ }**).

O teste condicional ocorre no início da estrutura, o que significa haver a possibilidade da repetição não ser executada nenhubma vez, se a avaliação da expressão condicional for falsa já no primeiro teste.

Exemplo:

```
// Conta os caracteres digitados
#include <iostream.h>
#include <conio.h>
void main( )
{       int cont = 0;
        while((getche()) != '\r')       // '\r' início da linha (enter)
                cont++;
        cout << "\nNumero de caracteres digitados: " << cont;
}
```

Outro exemplo:

```
/*
Multiplica dois valores fornecidos pelo usuário e
apresenta o resultado, repetindo esse processamento
por tantas vezes quantas forem a necessidade do usuário.
*/
#include <iostream.h>

void main( )
{
      int num1, num2, prod;
      char resp = 's';
      while(resp == 's')
      {
          cout<<"\nInsira dois numeros: ";
          cin>>num1>>num2;
          prod=num1*num2;
          cout<<"Produto: "<<prod;
          cout<<"\nDeseja continuar? ";
          cin>>resp;
      }
}
```

Estrutura do...while

A repetição **do...while** também pode ser utilizada quando não se conhece de antemão o número de iterações que serão realizadas. A execução do bloco será repetida até que a expressão condicional for falsa (igual a zero).

Sintaxe:

```
do
{
        instrução;
        instrução;
} while (condição);
```

O bloco de instrução é executado enquanto a condição for verdadeira. O bloco que contém várias instruções deve estar delimitado por chaves (**{ }**).

Diferentemente do comando while, o teste condicional ocorre no final da estrutura, sendo utilizado em situações que exigem pelo menos uma execução do bloco para depois criar o ciclo repetitivo, se a avaliação condicional for verdadeira.

Exemplo:

```
// Calcula e mostra o fatorial de um número fornecido pelo usuário.
#include <iostream.h>
void main()
{
    cout<< "\nIntroduza um numero para cálculo do fatorial: ";
    int num, fat;
    cin >> num;
    int i = fat = 1;
    do
        fat *= i++;
    while (i <= num);
    cout << "Fatorial de " << num << " = " << fat;
}
```

Observação sobre os comandos de repetição

A opção de repetição **while** serve para todas as situações, no entanto as outras duas em certas situações podem ter melhor aplicação.

Comandos de desvios

As instruções estudadas anteriormente podem sofrer **desvios** e **interrupções** em sua seqüência lógica normal através do uso de certas instruções. As instruções **break** e **continue** são usadas nas estruturas de repetição, não podendo ser utilizadas em outras partes do programa.

O comando break

O comando **break** é utilizado para terminar a execução das instruções de um laço de repetição (**for, while** e **do...while**) ou para terminar uma estrutura seletiva **switch**.

Quando utilizado em um laço de repetição, o comando break força a quebra imediata do laço independentemente da condição de teste; o controle passa para a próxima instrução após o laço. Em laços aninhados ou estruturados em hierarquias, o controle passa para o laço de nível imediatamente superior ou imediatamente externo.

A seguir, é apresentado outra versão do programa exemplo que simula uma calculadora para mostrar o uso da instrução **break**.

```
/* programa que simula uma calculadora eletrônica,
   efetuando uma das operações básicas.*/

#include <iostream.h>
void main()
{
        const int verdade=1;
        float n1, n2;
        char op;
        while(verdade)
        {
                cout << "\n\nDigite 0-0 (zero hífen zero) para terminar) ou";
                cout << "\nIntroduza a operacao (1o.num oper 2o.num): ";
                cin >> n1 >> op >> n2;

                if (n1 == 0.0)   break;      // força a saída do laço e o programa é encerrado

                if (op == '+')    cout << "Soma: " << n1 + n2;
                else  if (op == '-')  cout << "Subracao: " << n1 - n2;
                else  if ((op == '*') || (op == 'x'))  cout << "Multiplicacao: " << n1 * n2;
                else  if ((op == '/' ) || (op == ':'))   cout << "Divisao: " << n1 / n2;
                else  cout << "Operador invalido";
        }
}
```

O comando continue

O comando **continue** salta as instruções de um laço sem sair do mesmo, ou seja, o comando força a avaliação da condição de controle do laço e em seguida continua o processo do laço, nas estruturas de repetição **while** e **do...while**. No laço **for**, primeiro é incrementada ou decrementada a variável de controle e depois faz a avaliação da condição.

Exemplo:

```
// Lê valores para o cálculo da média, encerrando
// o processo sem considerar o valor negativo.
#include <iostream.h>
void main()
{
        cout<<"\nInsira valores para o cálculo da média.";
        cout<<"\nPara encerrar digite um valor negativo:\n";

        float valor,soma=0.0,num=0.0;
        do
        {
                cin>>valor;
                if (valor < 0.0)          // se negativo a instrução continue é executada
                        continue;         // salta para a avaliação do laço
                soma += valor;
                num++;
        } while (valor >= 0.0);                  // avaliação do laço
        cout<<"Soma = "<<(soma/num);
}
```

O comando goto

Este comando causa o desvio de fluxo do programa para a instrução seguinte ao *label* indicado no comando. O *label* ou rótulo é um nome seguido de dois pontos (:).

O uso deste comando deve ser evitado porque tende a dificultar a clareza e entendimento da lógica, mormente, de programas extensos.

A seguir, é apresentada uma versão do programa anterior para exemplificar o uso deste comando:

```
// Lê valores para o cálculo da média, encerrando
// o processo ao ler um valor negativo.
#include <iostream.h>
void main()
{
       cout<<"\nInsira valores para o cálculo da média.";
       cout<<"\nPara encerrar digite um valor negativo:\n";
       float valor,soma=0.0,num=0.0;
inicio:
       cin>>valor;
       if (valor < 0.0)
            goto fim;
       soma += valor;
       num++;
       goto inicio;
fim:
       cout<<"Soma = "<<(soma/num);
}
```

FUNÇÕES

Uma função é um segmento de programa que executa uma tarefa específica. Um recurso valioso que permite a modularização ou a divisão do programa em pequenas tarefas que simplifica a programação como um todo.

Um programa pode conter várias funções, devendo ter no mínimo uma função chamada main() através da qual é iniciada a execução do programa. Quando uma instrução contém um nome de uma função, ela é chamada, passando o controle das tarefas para essa função que executa suas instruções e volta à instrução seguinte à da chamada. Existem funções do sistema que são fornecidas junto com o compilador e funções do usuário escritas pelo programador.

A sintaxe de chamada a uma função possui o mesmo formato tanto para as definidas pelo programador quanto para as do sistema.

Neste trabalho já foram utilizadas as funções **gets()** e **getche()** do sistema. Outra função do sistema de largo uso é a **sqrt()** que calcula a raiz quadrada. Segue um exemplo:

```
#include <iostream.h>
#include <math.h>               //  protótipo para sqrt()

void main(void)
{
        cout << "\nRaiz quadrada de 25.0: "<< sqrt(25.0);
        cout << "\nRaiz quadrada de 50.0: "<< sqrt(50.0)<< endl;
}
```

A seguir é mostrado um exemplo de um programa com uma função criada pelo usuário para calcular a média aritmética de dois números reais:

```
// Efetua o cálculo da média aritmética de dois números reais.
#include<iostream.h>

float media(float n1, float n2);                        // protótipo

void main( )
{
          float num1, num2, med;
          cout<<"\nDigite dois números: ";
          cin >> num1 >> num2;

          med = media(num1, num2);              // chamada à função

          cout<<"\nMedia = " << med;
}

// Definição da função

float media(float n1, float n2)
```

```
{
        float m;
        m = (n1 + n2) / 2;
        return m;
}
```

Cada função especificada no programa apresenta os seguintes elementos: protótipo, chamada à função e definição.

Protótipo de funções

O protótipo refere-se a instrução de declaração que estabelece o tipo da função e os argumentos que ela recebe. O protótipo da função, geralmente é codificado no início do programa uma vez que ele deve preceder a definição e a chamada da função correspondente.

O tipo da variável especificado no protótipo é obrigatório, porém a informação do nome da variável é facultativa.

A chamada à função termina com a instrução **return()** que transfere o controle para a instrução seguinte a da chamadora. Esta instrução tem duas finalidades: determina o **fim lógico** e o **valor de retorno** da função.

Definição da função

O corpo ou o bloco de instruções que descreve o que a função faz é chamado de definição da função.

Sintaxe da função:

```
tipo  nome ( declaração dos argumentos)
{
        bloco de instruções da função
}
```

Na primeira linha, denominado cabeçalho da definição da função, o tipo especifica qual o tipo de dado retornado pela função. Os tipos de funções são os mesmos tipos das variáveis. Se a função não retorna nenhum valor para o programa que a chamou o tipo de retorno é definido como tipo **void**. Caso nenhum tipo seja declarado no cabeçalho o sistema considera que o valor de retorno será do tipo **int**.

A especificação do nome da função segue as mesmas regras descritas para nomear variáveis e constantes.

Cada função pode receber vários argumentos ou valores e pode retornar à função chamadora um único valor.

As informações transmitidas para uma função são chamadas de **parâmetros**. A função receptora deve declarar essas informações entre parênteses as quais podem ser utilizadas

no corpo da função.

Se a definição da função for escrita por inteiro antes da instrução de sua chamada, o correspondente protótipo não será necessário.

O exemplo anterior poderia ser escrito como segue:

```
// Efetua o cálculo da média aritmética de dois números reais.
#include<iostream.h>

// Definição da função

float media(float n1, float n2)
{
        float m;
        m = (n1 + n2) / 2;
        return m;
}

void main( )
{
        float num1, num2, med;
        cout<<"\nDigite dois números: ";
        cin >> num1 >> num2;

        med = media(num1, num2);            // chamada à função

        cout<<"\nMedia = " << med;
}
```

Passagem de parâmetros por valor

No exemplo anterior a função **media** declara as variáveis **n1** e **n2** para receber os valores passados. Este tipo de passagem de argumentos é chamado de **passagem por valor** pois os valores das variáveis do programa chamador são copiados para as correspondentes variáveis da função chamada.

Passagem de parâmetros por referência

Os parâmetros passados por valor são copiados para as variáveis argumentos; a função chamada não tem acesso as variáveis originais. Na passagem por referência a função chamada pode acessar e modificar o conteúdo das variáveis da função chamadora, não ocorrendo, sob esse aspecto, nenhuma cópia dos valores.

Segue exemplo do uso de referência como argumento de uma função:

```
// Converte a temperatura informada em graus Fahrenheit para Celsius.
#include<iostream.h>

void celsius (int& fahr, int& cels);    // protótipo

void main()
{
        int f, c;
        cout<<"\nInsira a temperatura em graus Fahrenheit: ";
        cin>>f;
        celsius(f,c);               // chamada à função
        cout<<"\nTemperatura em graus Celsius: "<<c;
}

// Definição da função

void celsius(int& fahr, int& cels)
{
        cels = (fahr - 32) * 5 / 9;
}
```

Observe que é utilizado o operador **&** na declaração do tipo do argumento da função receptora. As variáveis **fahr** e **cels** declaradas na função celsius() são as mesmas variáveis **f** e **c** da função principal, apenas referenciadas por outros nomes. Portanto, qualquer modificação de conteúdo das variáveis **fahr** e **cels** realizada pela função celsius() são automaticamente refletidas nas variáveis **f** e **c** da função main().

Outro exemplo:

```
// troca a ordem de dois números digitados
#include <iostream.h>

void troca(int& a, int&b);

void main(void)
{
        int n1,n2;
        cout<<"\nInsira dois numeros: ";
        cin>>n1>>n2;
        cout << "\nDigitados: "<<n1<<", "<<n2;
        troca(n1,n2);
        cout << "\nTrocados:  "<<n1<<", "<<n2;
}

void troca(int& a, int&b)
{
        int temp;
        temp = a;
        a = b;
        b = temp;
}
```

Passagem e retorno de valores

As funções do tipo void não retornam nenhum valor e podem também não receber nenhum argumento.

Segue um exemplo que recebe uma constante como argumento e não retorna nada:

```
// exibe um traçado em forma de caixa com um texto interno.
#include <iostream.h>

void linha(int num);

void main()
{
        cout<<"\n\n";
        linha(23);
        cout<<"\t\t| Isto e' uma moldura |\n";
        linha(23);
}

void linha(int num)
{
        cout<<"\t\t";                  // tabulação dupla
        for(int i=0; i<num; i++)
                cout<<'=';
        cout<<endl;
}
```

Resultado do exemplo::

```
        ====================
        | Isto e' uma moldura |
        ====================
```

A seguir, é apresentado um exemplo onde a função não recebe parâmetro e não retorna valor. O exemplo utiliza o caractere '\x07' que emite um beep:

```
// Emite sinal sonoro ao ser digitado um número impar.
#include<iostream.h>

void main()
{
        void beep(void);                    //protótipo local

        cout<<"\nDigite um numero par: ";
        int np;
        cin>>np;
        if (np%2==1)
```

```
        {
                cout<<"Voce digitou um numero impar"<<endl;
                beep();
        }
}

void beep(void)
{
        cout<<'\x07'<<'\x07';
        for(int i=0; i<5000; i++);
        cout<<'\x07'<<'\x07';
}
```

Neste exemplo, é mostrado também uma forma alternativa de informar o protótipo da função no programa: O protótipo pode ser declarado como **local** ou **externo**. O protótipo **externo** é escrito antes de qualquer função como fora utilizado em exemplos anteriores e é a forma mais usada. O protótipo **local** é escrito no corpo da função chamadora antes de sua chamada.

Sobrecarga de funções

As funções com o mesmo nome, mas com uma lista de parâmetros diferentes são chamadas de funções sobrecarregadas. A diferença pode residir no tipo ou na quantidade de argumentos da lista de parâmetros. O sistema identifica a função apropriada para execução no momento da chamada através da analise dos parâmetros passados.

Exemplo de funções sobrecarregadas que são distinguidas na chamada pelo número distintos de parâmetros:

```
#include <iostream.h>

int soma(int a, int b);                 // protótipo da função com dois argumentos
int soma(int a, int b, int c);          // protótipo da função com três argumentos

void main(void)
{
        cout << "Soma dos nros.: 200 + 801 = " << soma(200, 801) << endl;
        cout << "Soma dos nros.: 100 + 201 + 700 = "
       << soma(100, 201, 700) << endl;
}

int soma(int a, int b)                  // função soma com dois argumentos
{
        return(a + b);
}

int soma(int a, int b, int c)     // função soma com três argumentos
{
        return(a + b + c);
```

```
}
```

Funções inline

A função inline armazena tantas cópias de seu código quantas forem as chamadas feitas a ela, ou seja, onde ocorrer no programa uma chamada à função o compilador baixa uma cópia dessa função naquele local.

A função **inline** é reconhecida como tal por conter no cabeçalho da definição a palavra reservada inline e deve ser totalmente escrita antes de se fazer a primeira chamada para que compilador possa inserir nesse local uma cópia da função.

Exemplo de um programa que desenha dois retângulos de tamanhos distintos:

```
#include <iostream.h>

inline void retangulo(int lar, int alt)
{
	for (int i=1; i<=alt; i++)
	{
		cout<<"\t\t";
		for (int j=1; j<=lar; j++)
		    cout << '\xDB';
		cout<<'\n';
	}
}

void main()
{
	cout<<"\n1o.retangulo\n";
	retangulo(9,3);			// o compilador insere uma cópia da função inline
	cout<<"\n2o.retangulo\n";
	retangulo(7,2);			// o compilador insere uma cópia da função inline
}
```

Funções recursivas

Uma função recursiva é aquela que contem uma instrução que chama a si mesma repetidamente um número finito de vezes.

Uma situação em que a recursividade é apropriada é a do cálculo fatorial.

Exemplo:

```
#include <iostream.h>
int fatorial(int n);
void main()
{
```

```
	int n;
	do
	{
		cout<<"\nDigite um numero (negativo p/ encerrar): ";
		cin>>n;
		if (n<0) break;
		cout<<"\nFatorial: "<<fatorial(n);
	} while (1);
}

int fatorial(int n)
{
	return ((n = = 0) ? 1 : n * fatorial(n-1));
}
```

Funções do usuário armazenadas em bibliotecas

Funções de uso geral escritas pelo usuário, geralmente, são armazenadas em arquivos organizados em pastas específicas, constituindo-se nas chamadas **bibliotecas do usuário**. Os programas podem acessar essas bibliotecas para adicionar funções a partir de arquivos nomeados no programa. É conveniente citar que esse procedimento é bastante utilizado por programadores profissionais.

A inclusão de funções escritas e armazenadas em arquivo separado do programa principal é feita através da diretiva **#include**, onde é indicado, entre aspas duplas, o nome do arquivo precedido do endereço de sua localização.

Segue exemplo de um programa composto pela função principal e por outras duas funções armazenadas no arquivo de nome "mediaCxa.bib" da biblioteca cpp.

```
// Efetua o cálculo da média e exibe o resultado numa caixa de texto.
#include <iostream.h>
#include"c:\cpp\mediaCxa.bib"

void main()
{
	cout<<"\n\n";
	linha(20);
	int med = media(30,20);
	cout<<"\n\t\t||   Media = "<<med<<"   ||\n";
	linha(20);
}
```

Conteúdo do arquivo "mediaCxa.bib":

```
// Definição da função cálculo da média.

float media(float n1, float n2)
{
```

```
        float m;
        m = (n1 + n2) / 2;
        return m;
}

// Definição da função caixa de texto.

void linha(int num)
{
        cout<<"\t\t";
        for(int i=0; i<num; i++)
                cout<<'=';
}
```

Resultado da execução:

```
                ================
                ||   Média = 25      ||
                ================
```

Escopo das variáveis

O escopo define o **âmbito de visibilidade** de variáveis. Em outras palavras, determina de que locais do programa uma variável pode ser acessada e modificada.
Há quatro categorias de variáveis:

auto	-	automáticas;
extern	-	externas;
static	-	estáticas;
register	-	em registradores.

auto

As variáveis usadas em todos os exemplos até aqui apresentados são da categoria automática. São visíveis apenas às funções nas quais são declaradas. **Auto** é uma palavra reservada para declarar uma variável automática, no entanto, se nada for mencionado na definição da variável, ela é considerada auto. As variáveis automáticas são criadas em tempo de execução do bloco ou da função, não são inicializadas pelo sistema e ao terminar a execução são destruídas.

extern

As variáveis externas são declaradas fora da definição da função. Uma variável externa é visível a todas as funções, ou seja, pode ser acessada e modificada por qualquer função definida após a sua declaração. O espaço de memória alocado para esta variável é **criado** em tempo de compilação e **destruído** no encerramento do programa. Por *default*, as variáveis externas são inicializadas com zeros.

A palavra reservada **extern** deve ser precedida na declaração da variável somente no caso de ter sido criada em outro programa que fora compilado em separado.

```
// mostra o uso de variáveis externas e automáticas
#include <iostream.h>

int i = 10;             // variável externa
int j;                  // variável externa inicializada com zero

void soma(int x, int y);

void main()
{
        cout<<"\nInsira um numero: ";
        cin>>j;
        int k = 1;                      //  variável automática
        auto int m = 2;                 //  idem
        soma(k,m);
        cout<<"\nSoma: "<<j;
}

void soma(int x, int y)
{
        j += i + x - y;
}
```

Variáveis automática e externa com o mesmo identificador ao ser referenciada no programa, o sistema dá prioridade para a variável automática que tem precedência sobre a variável externa. Neste caso, para acessar a variável externa deve ser usado o **operador de escopo** (**::**) antes do nome da variável.

Exemplo:

```
// mostra o uso do operador de escopo
#include <iostream.h>

int i = 10;             // variável externa
int j;                  // variável externa inicializada com zero

void main()
{

        int i = 1;              //  variável automática
        int j = 2;              //  idem
        cout<<"\nExternas: "<<::i<<", "<<::j;

        cout<<"\nAutomaticas: "<<i<<", "<<j;
}
```

Resultado da execução:

```
        Externas: 10, 0
        Automáticas: 1, 2
```

static

As variáveis estáticas, como as automáticas, são visíveis apenas pela função em que foram declaradas, com a diferença de que elas mantêm armazenados seus valores quando a função é encerrada. As variáveis static são criadas em tempo de compilação e são inicializadas com zero na ausência de inicialização explícita.

Exemplo:

```
// mostra o uso do operador de escopo
#include <iostream.h>

void soma(void);

void main()
{
        soma(); soma();
}

void soma()
{
        static int i = 5;          //  variável estática
        cout<<"\nStatic i: "<<i++;
}
```

Resultado da execução:

```
Static i: 5
Static i: 6
```

Na primeira chamada, o sistema inicia e encerra a execução da função soma e ao reiniciar a função, na segunda chamada, o conteúdo anterior da variável continua preservado.

register

A classe register é aplicada a variáveis declaradas com tipos de dados **int** e **char** e são semelhantes às automáticas, exceto por serem armazenadas em componentes de memória de alta velocidade de acesso e processamento denominados **registradores**. São inicializadas com zero na ausência de inicialização explícita.

Exemplo:

```
#include <iostream.h>
void main( )
{
        register int x;
        …
}
```

Diretivas

As diretivas são instruções da biblioteca de programas do compilador C++ que são processadas antes da compilação propriamente dita. O pré-processador que analisa as diretivas faz parte do compilador. Ele substitui a indicação do arquivo declarado na diretiva do programa-fonte pelo seu conteúdo. As diretivas, geralmente, são escritas no início do programa, no entanto, podem ser escritas em qualquer lugar do programa antes de serem usadas.

As diretivas mais utilizadas são:

#include, #define, #undef.

A diretiva #include determina a inclusão do arquivo especificado no programa por meio do pré-processador.

Exemplo:

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

O nome do arquivo de inclusão pode ser delimitado por aspas: "iostream.h" em vez dos símbolos < e >. As aspas fazem com que a busca do arquivo comece no diretório atual do sistema e depois no diretório **include**.

Para usar as funções da biblioteca de programa C++, deve-se incluir um arquivo de cabeçalho da biblioteca que especifica o protótipo da função. Os nomes dos arquivos de inclusão possuem a extensão **.h** (header) e se encontram armazenados no diretório **include**.

Seguem os nomes de alguns arquivos da biblioteca de inclusão:

conio.h	ctype.h	dir.h	dirent.h	dos.h
errno.h	fcntl.h	float.h	fstream.h	generic.h
graphics.h	io.h	iomanip.h	iostream.h	limits.h
locale.h	malloc.h	math.h	mem.h	process.h
setjmp.h	share.h	signal.h	stdarg.h	stddef.h
stdio.h	stdiostr.h	stdlib.h	stream.h	string.h
strstrea.h	sys\stat.h	sys\time.h	sys\types.h	time.h

Algumas funções contidas no arquivo de inclusão math.h:

(O protótipo da função informa que tipo de dado é retornado e que tipo de dado deve ser enviado como parâmetro).

Funções matemáticas:

int **abs**(int) (determina o valor absoluto – inteiro)
double **fabs**(double) (determina o valor absoluto – real)
double **pow**(double *base*, double *exp*) (potenciação: pow(3.2,5.6) = $3.2^{5.6}$)
double **sqrt**(double) (raiz quadrada: sqrt(9.0) = 3.0)

Funções de arredondamento para inteiro:

double **ceil**(double); (arredonda para cima: ceil(3.2) = 4.0)
double **floor**(double); (arredonda para baixo: floor(3.2) = 3.0)

Funções trigonométricas do ângulo arco, em radianos:

double **sin**(double)
double **cos**(double)
double **tan**(double)
double **asin**(double)
double **acos**(double)
double **atan**(double)

Funções logarítmicas:

double **log**(double);
double **log10**(double);

VETORES E MATRIZES

Variáveis compostas do mesmo tipo de dado (inteiro, real ...) são denominadas de conjunto homogêneo de dados. Esses agrupamentos de variáveis são conhecidos como **vetores**, **matrizes** (arrays, em inglês), tabelas ou variáveis indexadas ou subscritas.

São variáveis do mesmo tipo declaradas com o mesmo identificador e referenciadas por um índice para determinar sua localização dentro da estrutura.

Em outras palavras, vetor ou matriz é uma série de elementos (variáveis) do mesmo tipo, declarados com um único nome, sendo que cada elemento ou variável é acessado individualmente através de índices.

É conveniente citar que muitos autores distinguem as estruturas compostas homogêneas, chamando as estruturas de uma dimensão de vetores e, as de mais de uma dimensão, de matrizes. Neste trabalho, achou-se apropriado usar essas nomenclaturas, tratando-as em seções separadas.

Vetores

Um vetor é constituído de variáveis compostas unidimensionais, também, chamado de matriz unidimensional e que necessitam de apenas um índice para que suas variáveis sejam endereçadas.

Um modo simples de entender um vetor é através da visualização de uma lista de elementos que contém um nome e um índice para referenciar os valores dessa lista.

Exemplo:

i	**nota**
0	9.5
1	7.4
2	5.8
3	8.0
4	6.0

i (índice) representa o número de referência e **nota** (identificador) é o nome do conjunto de elementos. Dessa forma, pode-se dizer que a 3ª nota é 5.8 e que pode ser representado como segue: nota[2] = 5.8. Como pode ser visto, a lista de notas contém números reais ou é do tipo de dados **float**.

Em C++, esta lista poderia ser declarada assim:

```
float nota[5];
```

A seguir é apresentado exemplo de um programa que calcula a média aritmética das notas de provas de cinco alunos e exibe as notas maiores ou iguais a média:

```
#include <iostream.h>
void main()
{
	float nota[5];			// declaração do vetor
	float media=0;
	for(int i=0; i<5; i++)
	{
		cout<<"Digite a "<<(i+1)<<"a.nota: ";
		cin>>nota[i];		//  armazena a nota no vetor
		media += nota[i];
	}
	media /= 5;
	cout<<"\nRelacao das notas maiores ou iguais a media: "<<media;
	for(i=0; i<5; i++)
	{
		if(nota[i]>=media)
			cout<<'\n'<<nota[i];   // lista a nota a partir do vetor
	}
}
```

Resultado da execução:

```
	Digite a 1a.nota: 9.5
	Digite a 2a.nota: 7.4
	Digite a 3a.nota: 5.8
	Digite a 4a.nota: 8.0
	Digite a 5a.nota: 6.0

	Relacao das notas maiores ou iguais a media: 7.34
	9.5
	7.4
	8.0
```

Declaração de vetor

Sintaxe da declaração:

```
tipo nome[n];
```

n entre colchetes representa o número de ocorrências ou a quantidade de elementos da variável **nome** de um **tipo** específico.

```
float nota[5];
```

é o vetor declarado no exemplo do programa anterior. Ele é formado por cinco elementos do tipo **float**. O índice utilizado para referenciar os elementos é iniciado por zero e pode ser o valor de uma constante ou de uma variável inteira. No exemplo acima foi utilizado como índice a variável de nome **i**.

Inicialização de um vetor

Um vetor pode receber valores na mesma instrução de sua declaração.

Exemplo:

```
int  brasil[ ] = {58,62,70,94,02};
```

O valores são escritos entre chaves, separados por vírgulas. Num vetor inicializado pode-se suprimir o número de elementos, deixando o par de colchetes vazio, ou informar de modo explícito o número de elementos, como segue:

```
int  brasil[5] = {58,62,70,94,02};
```

Atribuição de valores

Cada elemento do vetor é referenciado por um índice cuja numeração começa em zero. A instrução

```
nota[4] = 10.0;
```

atribui o valor 10.0 ao quinto elemento do vetor. Como o índice é numerado a partir de zero, o índice do último elemento do vetor é igual ao tamanho do vetor – 1.

Tamanho do vetor

Um vetor não pode ser declarado de tamanho variável. Para controlar o limite máximo do vetor pode-se declarar uma constante simbólica através da diretiva #define para controlar o tamanho do vetor.

O programa anterior poderia ser codificado como segue:

```
#include <iostream.h>
#define TAM 5                           // controla limite do vetor
void main()
{
        float nota[TAM];                // declaração do vetor
        float media=0;
        for(int i=0; i<TAM; i++)
        {
                cout<<"Digite a "<<(i+1)<<"a.nota: ";
                cin>>nota[i];
                media += nota[i];
        }
        media /= TAM;
        cout<<"\nRelacao das notas maiores ou iguais a media: "<<media;
        for(i=0; i<TAM; i++)
        {
                if(nota[i]>=media)
```

```
                    cout<<'\n'<<nota[i];
        }
}
```

Desta forma, é facilitada a alteração do tamanho do vetor. Caso se pretenda calcular a média de mais alunos, basta modificar o texto da constante simbólica. Por exemplo, para processar as notas de 20 alunos, pode-se apenas alterar a instrução da diretiva #define para #define TAM 20.

Matrizes

Pode-se tratar matrizes de várias dimensões. O limite dimensional depende dos recursos de processamento disponíveis. Matrizes unidimensionais, mais conhecidas como vetores, foram descritas na seção anterior. Nesta seção, serão discutidas as matrizes bidimensionais, as mais utilizadas na prática.

Por exemplo, a declaração

```
Int A[3][5]
```

define uma matriz bidimensional de 3 x 5 = 15 números inteiros. É conveniente imaginar, o primeiro índice como o índice de linha e o segundo como índice de coluna. Assim, entende-se como mostrado na seguinte ilustração:

```
[0][0]  [0][1]  [0][2]  [0][3]  [0][4]
[1][0]  [1][1]  [1][2]  [1][3]  [1][4]
[2][0]  [2][1]  [2][2]  [2][3]  [2][4]
```

Do mesmo modo como ocorre com os vetores, os índices começam em zero e vão até o tamanho das dimensões da matriz menos uma unidade.

Segue exemplo de um programa que tem por objetivo carregar uma matriz 5 x 3 com valores das vendas de uma empresa, em que cada linha represente as vendas de um vendedor e cada coluna um mês de vendas. Em seguida, o programa deve determinar e escrever o código do vendedor e o valor das vendas que estejam acima da média.
Esquema de armazenamento:

	1	2	3	⇐ mês
10	3800,00	4200,00	2460,00	
20	4150,00	3940,00	5120,00	
30	3870,00	4060,00	4310,00	
40	3600,00	3500,00	4600,00	
50	5270,00	4520,00	4090,00	

⇑
vendedor

```
#include <iostream.h>
void main()
{
	float vendas[5][3];			// declaração da matriz
	float soma=0, media;
	for(int i=0; i<5; i++)
		for(int j=0; j<3; j++)
		{
		    cout<<"Insira o valor da venda - vendedor: "<<((i+1)*10)<<" mês “
			<<j+1<<": ";
		    cin>>vendas[i][j];
		    soma += vendas[i][j];
		}
	media = soma/5;
	for(i=0; i<5; i++)
	{
		soma = 0;
		for(int j=0; j<3; j++)
		    soma += vendas[i][j];
		if (soma>media)
		    cout<<"\nVendedor: "<<((i+1)*10)<<" valor: "<<soma;
	}
}
```

Inicialização de matriz bidimensional

Uma matriz de duas dimensões contendo, por exemplo, as notas de quatro provas de três alunos pode ser visualizada como a lista apresentada abaixo: (os números colocados fora da tabela representam os valores dos índices).

Exemplo:

	0	1	2	3
0	8.5	7.0	7.8	9.0
1	7.3	8.2	5.9	6.5
2	6.0	8.0	7.0	7.2

Segue a inicialização da matriz com os valores da tabela acima:

```
float notas[3][4] = {   {8.5, 7.0, 7.8, 9.0},
                        {7.3, 8.2, 5.9, 6.5},
                        {6.0, 8.0, 7.0, 7.2} };
```

O valores de toda matriz são escritos entre chaves, podendo ser numa única linha, e, igualmente, entre chaves aparecem os valores das linhas separados por vírgulas e, por sua vez, as linhas também são separadas por vírgulas após o seu fechamento de chave.

Passando vetores e matrizes para funções

Assim como as variáveis, os vetores e matrizes podem ser passados como argumento para uma função.

Sintaxe da passagem para a função:

```
nome_função (nome_vetor, elementos_vetor)
```

Sintaxe do recebimento pela função:

```
tipo nome_função (tipo nome_vetor[ ], tipo elementos_vetor )
```

A seguir é mostrado um exemplo que passa vetores para a função exibe() para relacionar o conteúdo desses vetores:

```
#include <iostream.h>

void exibe (int lista[], int elementos);          // protótipo da função

void main()
{
        int par[10] = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20};      // declaração e inicialização
        int impar[10] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};    // idem
        cout<<"\nNumeros pares: ";
        exibe(par,10);                            // chama função exibe
        cout<<"\nNumeros impares: ";
        exibe(impar, 10);                         // chama a função exibe
}

void exibe (int lista[], int elementos)          // recebe os agumentos
{
        for(int i=0; i<elementos; i++)
                cout<<lista[i]<<' ';
}
```

Como se vê, na chamada à função exibe(), o nome da matriz é escrito sem colchetes, cujo argumento é recepcionado numa variável seguida de colchetes vazios.

De modo semelhante ao vetor é feita a passagem de matriz para função. A sintaxe da passagem de valores de uma matriz de duas dimensões para a função é a mesma apresentada para o vetor. No entanto, aos parâmetros recebidos pela função deve ser incluído o índice da segunda dimensão após os colchetes vazios, sem separá-los por vírgula.

A seguir é apresentado um exemplo onde a função somaLinha() efetua a soma dos valores de cada linha da matriz e as exibe:

```
#include <iostream.h>

const lin=5, col=3;
```

```
void somaLinha(int somaLinha[][col], int linha);

void main()
{
        int matriz[lin][col] = {{1,2,3},{4,5,6},{7,8,9},
                                {10,11,12},{13,14,15}};
        somaLinha(matriz,lin);
}

void somaLinha(int mat[][col], int linha)
{
        int soma=0;
        for(int i=0; i<linha; i++)
        {
                soma = 0;
                for(int j=0; j<col; j++)
                    soma += mat[i][j];
                cout<<"\nSoma da linha "<<(i+1)<<": "<<soma;
        }
}
```

Resultado do exemplo:

```
        Soma da linha 1: 6
        Soma da linha 2: 15
        Soma da linha 3: 24
        Soma da linha 4: 33
        Soma da linha 5: 42
```

Convém notar que tanto os vetores como as matrizes podem ter seus valores modificados pela função chamada, uma vez que a passagem de parâmetros dessas estruturas sempre se faz por referência e não por valor.

STRING DE CARACTERES

Uma string ou cadeia de caracteres pode ser entendida como uma série de letras, números e simbolos especias escritos entre aspas, como por exemplo, a expressão "Curso prático de C++".

Uma string de caracteres é declarada como um **vetor** do tipo **char** capaz de armazenar um texto de um comprimento específico cujo término é marcado com o caractere zero ('\0'). O modo de indicar o último elemento do vetor é a principal diferença entre um vetor de carateres e outros tipos de vetores.

O caractere '\0' representa o símbolo NULL ou o decimal zero na tabela ASCII. É conveniente dizer que os caracteres que não podem ser obtidos diretamente do teclado como a tecla TAB, por exemplo, em C++ eles são codificados através da combinação de outros caracteres, geralmente, antecedidos por uma barra invertida (\).

Cada caractere de uma string ocupa um byte na memória e todos são armazenados consecutivamente, sendo que o último byte armazena o caractere NULL ('\0'), como mostra a ilustração que segue:

S	e	x	t	a		f	e	i	r	a	\0

Inicializando string de caracteres

O vetor de caracteres apresentado na ilustração acima pode ser inicializado através do modo padrão, como segue:

```
char texto [] = {'S', 'e', 'x', 't', 'a', ' ', 'f', 'e', 'i', 'r', 'a', '\0'}
```

ou do modo simplificado:

```
char texto[] = "Sexta feira"
```

No modo padrão os caracteres são escritos entre apóstrofos e é informado o caractere de fim ('\0'). No segundo caso, modo simplificado, a constante é escrita entre aspas e o caractere '\0' é inserido automaticamente pelo sistema.

Um caractere colocado entre apóstrofos é denominado **constante de caractere** e entre aspas, **constante string de caracteres** que sempre é seguida do caractere '\0' acrescentado pelo compilador.

A seguir, um exemplo que armazena as letras de A a Z no vetor de nome letras através do laço **for** e, em seguida, as exibe na tela:

```
#include <iostream.h>

void main()
{
        char letras[27];                // 26 letras mais o caractere NULL
```

```
        for (int i=0, char let='A'; let<='Z'; let++, i++)
                letras[i] = let;
        letras[i] = '\0';           // acrescenta o caractere NULL no fim

        for(i=0; letras[i] != '\0'; i++)
                cout<<letras[i];
        // ou
        cout<<'\n'<<letras;
}
```

Resultado do exemplo::

```
        ABCDEFGHIJKLMNOPQRSTUVWXYZ
        ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Atribuição de string

Para atribuir um valor a uma string de caracteres o método prático é utilizar a função que apresenta a seguinte sintaxe:

strcpy (string1, string2);

Este comando faz uma cópia do conteúdo da string2 para a string1. A utilização dessa função exige a indicação do arquivo de cabeçalho **string.h** através da diretiva #include.

Exemplo:

```
#include <iostream.h>
#include <string.h>             // protótipo da função
void main()
{
       char nome[50];

       strcpy(nome, " Yasmin Gabrielle");

       cout<<nome;
}
```

O nome Yasmin Gabrielle é atribuido ou copiado para o vetor com o identificador nome.

Entrada de string a partir do teclado

O objeto **cin** como até agora fora utilizado para dar entrada de valores oriundos do teclado não é apropriado para ler string de caracteres constituída de palavras e espaços, porquanto a entrada é encerrada quando o comando **cin** encontra um espaço em branco.

Segue um exemplo:

```
#include <iostream.h>
void main()
{
        char nome[50];

        cout<<"Digite um nome: ";

        cin>>nome;

        cout<<nome;
}
```

Resultado do exemplo:

```
Digite um nome:  Yasmin Gabrielle
Yasmin
```

Como se vê, o comando **cin** encerra a entrada ao encontrar o espaço em branco entre os dois nomes digitados.
Para introduzir o nome ou um texto completo é comum a utilização do método **getline** do objeto **cin** que encerra a entrada ao ser pressionada a tecla ENTER. O seguinte exemplo corrige o anterior:

```
#include <iostream.h>
void main()
{
        char nome[50];
        cout<<"Digite um nome: ";

        cin.getline(nome,50);

        cout<<nome;
}
```

Resultado do exemplo:

```
Digite um nome:  Yasmin Gabrielle
Yasmin Gabrielle
```

O primeiro parâmetro da instrução **cin.getline (nome,50)** é o nome da variável (como o vetor, neste caso) onde o dado de entrada do usuário será armazenado e o segundo parâmetro indica o tamanho máximo dessa variável.

Uma string de caracteres pode ser considerada como uma matriz de uma linha por n colunas onde é referenciada apenas a coluna. Assim sendo, é possível inserir outras linhas, transformando-a numa matriz de duas dimensões. Segue um exemplo onde são codificados os meses do ano:

```
#include <iostream.h>
void main()
{
```

```
        char meses[12][4] ={"jan","fev","mar","abr","mai","jun",
                            "jul","ago","set","out","nov","dez"};
        int mes;
        do
        {
                cout<<"Digite o numero do mes: ";
                cin>>mes;
        } while(mes<1 || mes>12);
        cout<<"\nSigla do mes: "<<meses[(mes-1)]<<endl;
}
```

Resultado do exemplo:

```
        Digite o numero do mes: 10
        Sigla do mes: out
```

Observe que embora a matriz **char meses[12][4]** apresente dois índices (matriz bidimensional), a instrução para impresão na tela:

```
cout<<"\nSigla do mes: "<<meses[(mes-1)]<<endl;
```

utiliza apenas um índice **[(mes-1)]**.

Passando strings para funções

A passagem de uma string de caracteres é semelhante a passagem de um vetor como um parâmetro.

Segue um exemplo de um programa que exibe na tela uma string de caracteres:

```
#include <iostream.h>

void exibe(char txt[]);

void main()
{
        cout<<"\nDigite uma linha de texto: ";
        char texto[80];

        cin.getline(texto,sizeof(texto));

        exibe(texto);
}

void exibe(char txt[])
{
        cout<<"\nTexto digitado: "<<txt;
}
```

Resultado do exemplo:

```
Digite uma linha de texto: Passagem de parametro.
Texto digitado: Passagem de parametro.
```

Como se vê, especifíca-se na função **void exibe(char txt[])** o tipo e o nome do vetor seguidos de dois colchetes, sem a especificação do tamanho da string.

Verifica-se também neste exemplo que a instrução **cin.getline(texto,sizeof(texto));** aparece ligeiramente modificada em relação àquela utilizada no programa da secão anterior. Esse formato torna o programa mais flexível, uma vez que não demanda nenhuma alteração caso o tamanho da variável **texto** seja modificado.

Outras funções de strings

Os compiladores C++ fornecem uma coleção de funções diferentes de manipulação de strings. A biblioteca referente ao arquivo de cabeçalho **string.h** define várias funções para executar operações de manipulação de strings como a função strcpy já apresentada. A seguir, são descritas brevemente algumas delas:

strcat(str1, str2);

Concatena a string de carateres **str1** à string **str2**.

strcmp(str1, str2);

Compara as strings **str1** e **str2**. Retorna zero, se as strings forem iguais; retorna um valor maior que zero, se **str1** for maior que **str2**; e retorna um número menor que zero, se **str1** for menor que **str2**.

srtlen(str1)

Retorna o tamanho ou o número de carateres armazenados em **str1**.

ESTRUTURAS DE DADOS

Estruturas são agrupamentos de dados relacionados que podem ser de diferentes tipos básicos (int, float, double, char e void) sob uma única declaração. Já foi visto que as matrizes são tipos de variáveis que agrupam dados do mesmo tipo básico. As estruturas também são tipos de variáveis, mas que podem agrupar dados de tipos básicos diferentes. Os itens de dados das estruturas são denominados **membros**, ao passo que das matrizes são chamados de **elementos**.

Os agrupamentos de dados mais importantes disponíveis em C++ são as **estruturas** com seus **dados-membros** e as **classes,** que caracterizam a programação orientada a objetos, com seus **dados-membros** e **funções-membros**. Como a declaração das estruturas se assemelha à das classes, a compreensão daquelas ajuda sobremaneira no entendimento de classes e objetos.

Definição da estrutura

O formato da estrutura é discriminado como segue:

```
struct nome
{
        tipo membro1;
        tipo membro2;
        ...
} variáveis;
```

A estrutura é definida com a palavra-chave **struct** seguida de um nome identificador. Entre chaves é especificado um ou mais itens de dados (tipo e membro) e, por último, as variáveis que tanto podem ser declaradas aqui como em outra parte do programa que será mostrado mais adiante.

Exemplo de um programa que cria uma estrutura com descrição e preço, introduz e exibe seus conteúdos:

```
#include <iostream.h>
#include <string.h>

struct produto
{
        char descr[50];
        float preco;
} hd, cd;

void main()
{
        strcpy(hd.descr,"Hard disk Maxtor");
        hd.preco = 250.42;
```

```
        cout<<"Descricao: ";
        cin.getline(cd.descr,50);
        cout<<"Preco: ";
        cin>>cd.preco;

        cout<<"\nHD -\tDescricao: "<<hd.descr;
        cout<<"\n\tPreco: "<<hd.preco<<endl;

        cout<<"\nCD -\tDescricao: "<<cd.descr;
        cout<<"\n\tPreco: "<<cd.preco;
}
```

Outro modo de declarar as variáveis da estrutura é usando o nome-tipo da estrutura seguido das variáveis dentro da função:

```
#include <iostream.h>
#include <string.h>

struct produto
{
        char descr[50];
        float preco;
};                      // ponto e vírgula após o fechar chaves

void main()
{
        produto hd, cd;         // tipo da estrutura e suas variáveis
        ...
}
```

Ressalte-se que a criação da estrutura apenas caracteriza a definição de um novo tipo de dado criado pelo usuário. Para sua utilização, deve-se declarar ao menos uma variável do tipo produto, neste caso. À cada variável, **hd** e **cd** declaradas no exemplo, é reservado espaço contíguo de memória necessário para armazenar todos os membros da estrutura.

A definição da estrutura fora de qualquer função, como a que foi criada no programa acima, permite acesso a seus membros por todas as funções especificadas no programa. Por outro lado, a estrutura criada dentro da função restringe o acesso a esta função.

Acesso a membros da estrutura

Para acessar um membro da estrutura, deve-se informar os nomes da variável e do respectivo membro conectados por meio do operador ponto (**.**).

Exemplo de uma instrução de atribuição:

```
hd.preco = 250.42;
```

Este exemplo atribui o valor 250.42 ao membro preco da variável hd.

Inicialização de estruturas

As estruturas podem ser inicializadas na mesma instrução de sua definição e se parecem com a inicialização de vetores e matrizes.
Segue um exemplo:

```
#include <iostream.h>

struct contrato
{
        int matricula;
        char nome[20];
        float salario;
};

void main()
{
        contrato func = {123, "Ana Maria da Silva", 2500.00};
}
```

Os valores atribuídos à variável **func** devem seguir a ordem de acordo com a declaração dos membros na estrutura, separados por vírgulas e entre chaves.

Aninhamento e matriz de estruturas

O aninhamento de estruturas refere-se àquelas estruturas que participam como membros de outras estruturas. No exemplo, a seguir, vê-se a estrutura **data** definida como membro da estrutura **contrato**. Verifica-se também no mesmo exemplo uma matriz de 50 elementos declarada na variável func da estrutura contrato.
O exemplo objetiva cadastrar e listar elementos da variável func (funcionário) e exibir o valor total dos salários listados, segundo opção dada a escolher ao usuário.

```
#include <iostream.h>
#include <iomanip.h>

struct data                     // estrutura do tipo data
{
        int dia, mes, ano;      // membros da estrutura
};
struct contrato                 // estrutura do tipo contrato
{
        char nome[30];
        float salario;
        data nasc;              // membro da estrutura do tipo data
```

```
};

void incluir();                 // protótipo da função incluir()
void listar();                  // protótipo da função listar()

contrato func[50];              // declara uma matriz de estrutura
int i=0;

void main()
{
        int oper;
        int verdade=1;
        while(verdade)
        {
                cout<<"1 para incluir";
                cout<<"\n2 para listar";
                cout<<"\n3 para encerrar : ";
                cin>>oper;
                if (oper==1)    incluir();
                else
                        if (oper==2)listar();
                        else
                                verdade=0;
                cout<<'\n';
        }
}
void incluir()
{
        cout<<"\nNome: ";     cin>>func[i].nome;
        cout<<"Salario: ";      cin>>func[i].salario;
        cout<<"Dia nasc.: ";cin>>func[i].nasc.dia;
        cout<<"Mes nasc.: ";cin>>func[i].nasc.mes;
        cout<<"Ano nasc.: ";cin>>func[i++].nasc.ano;
}
void listar()
{
        float totsal=0;
        cout<<"\n   Salario"<<setw(13)<<"Data nasc."<<setw(12)<<"Nome";
        cout<<setprecision(2);                  // exibe duas casas decimais
        cout << setiosflags(ios::fixed);        // inibe a exibição em notação científica
        for(int j=0; j<i; j++)
        {
                cout<<'\n'<<setw(10)<<func[j].salario<<setw(6)
                        <<func[j].nasc.dia<<setw(3)
                        <<func[j].nasc.mes<<setw(3)
                        <<func[j].nasc.ano<<setw(20)<<func[j].nome;
                totsal += func[j].salario;
        }
        cout<<"\nTotal salario: "<<totsal<<endl;
}
```

As estruturas assim como a matriz e a variável índice (i) foram declaradas como componentes externos para permitir o acesso a eles pelas funções incluir() e listar().

Resultado de duas opções de cadastramento (opção 1) e uma de listagem (opção 2) :

```
1 para incluir
2 para listar
3 para encerrar : 1

Nome: Antonio
Salario: 2500.00
Dia nasc.: 12
Mês nasc.: 10
Ano nasc.: 72

1 para incluir
2 para listar
3 para encerrar : 1

Nome: Maria
Salario: 1850.42
Dia nasc.: 30
Mês nasc.: 05
Ano nasc.: 78

1 para incluir
2 para listar
3 para encerrar : 2

   Salario   Data nasc.         Nome
 2500.00   12 10 72              Antonio
 1850.42   30   5 78             Maria
Total salario: 4350.42

1 para incluir
2 para listar
3 para encerrar : 3
```

Estruturas e funções

Estruturas também podem ser passadas como argumento de funções de modo semelhante ao procedimento realizado com quaisquer variáveis comuns.

Passagem por valor

A seguir, um exemplo de uma função que efetua um cálculo e exibe a média de alunos por sala, a partir de duas variáveis do tipo estrutura passadas por valor:

```
#include <iostream.h>

struct curso
{
        int sala;
        float aluno;
};

void media(curso n1, curso n2);

void main()
{
        curso c1, c2;

        cout<<"Curso 1";
        cout<<"\n\tQuant. de salas:  ";        cin>>c1.sala;
        cout<<"\tNumero de alunos: ";        cin>>c1.aluno;

        cout<<"\nCurso 2";
        cout<<"\n\tQuant. de salas:  ";        cin>>c2.sala;
        cout<<"\tNumero de alunos: ";        cin>>c2.aluno;
        media(c1, c2);          // variáveis tipo estrutura como argumento
}

void media(curso n1, curso n2)
{
        float med;
        med = (n1.aluno + n2.aluno) / (n1.sala + n2.sala);
        cout<<"\nMedia de alunos por sala: "<<med;
}
```

Resultado do exemplo::

```
Curso 1
          Quant de salas: 12
          Numero de alunos: 260
Curso 2
          Quant de salas: 8
          Numero de alunos: 240

Media de alunos por sala: 25
```

Passagem por referência

De modo análogo à operação de passagem de variáveis comuns, procede-se com as variáveis tipo estrutura na passagem destas como argumento por **referência**. É bom lembrar que o método por referência economiza memória uma vez que o sistema não faz nenhuma cópia das variáveis para serem operadas pela função. Para apresentar um exemplo desta técnica, foi modificado o programa do exemplo anterior para receber argumentos por **referência**:

```
#include <iostream.h>

struct curso
{       int sala;
        float aluno;
};

void media(curso& n1, curso& n2);

void main()
{
        curso c1, c2;

        cout<<"Curso 1";
        cout<<"\n\tQuant. de salas:  ";        cin>>c1.sala;
        cout<<"\tNumero de alunos: ";        cin>>c1.aluno;

        cout<<"\nCurso 2";
        cout<<"\n\tQuant. de salas:  ";        cin>>c2.sala;
        cout<<"\tNumero de alunos: ";        cin>>c2.aluno;

        media(c1, c2);                  // variáveis tipo estrutura como argumento
}

void media(curso& n1, curso& n2)
{
        float med;
        med = (n1.aluno + n2.aluno) / (n1.sala + n2.sala);
        cout<<"\nMedia de alunos por sala: "<<med;
}
```

As variáveis **n1** e **n2** compartilham o mesmo espaço de memória ocupado pelas variáveis **c1** e **c2**. Observar que o operador & (operador de endereços) é escrito junto ao nome da estrutura que aparece no argumento da função média.

Retorno de variável tipo estrutura

A ilustração seguinte apresenta o mesmo exemplo modificado para que a função retorne uma variável tipo estrutura:

```
#include <iostream.h>

struct curso
{
        int sala;
        float aluno;
};

void media(curso& n1, curso& n2);

curso insere(void);

void main()
{
        curso c1, c2;

        cout<<"Curso 1";
        c1=insere();
        cout<<"Curso 2";
        c2=insere();

        media(c1, c2);
}

curso insere(void)
{
        curso sa;

        cout<<"\n\tQuant. de salas:  ";        cin>>sa.sala;
        cout<<"\tNumero de alunos: ";        cin>>sa.aluno;

        return (sa);
}

void media(curso& n1, curso& n2)
{
        float med;
        med = (n1.aluno + n2.aluno) / (n1.sala + n2.sala);
        cout<<"\nMedia de alunos por sala: "<<med;
}
```

Notar que a função insere( ) foi definida como uma função do tipo curso pois ela retorna uma variável do tipo especificado.

CLASSES E OBJETOS

Uma classe é um método lógico de organização de dados e funções numa mesma estrutura e é considerada como instrumento principal da programação orientada a objetos. A variável de uma classe é chamada **objeto** que fará uso dos dados e funções.

Classes

As classes são funcionalmente similares às estruturas, porém enquanto estas definem apenas dados, aquelas definem os chamados **dados-membro** e **funções-membro**. As funções-membro também conhecidas como **métodos** são responsáveis pelas operações realizadas sobre os **dados-membro** da classe.

Exemplo:

```
#include <iostream.h>

class area       // define a classe
{
private:         // dados-membro:
        float largura;
        float altura;
public:          // funções-membro:
        void obtemdados(float larg, float alt);         // protótipo da função
        float calcula()                                 // função inline
        { return(largura*altura);}
};

void area::obtemdados(float larg, float alt)
{
        largura=larg;
        altura=alt;
}

void main()
{
        area parede;            // declara o objeto

        parede.obtemdados(3.5, 5.2);
        float areapar = parede.calcula();
        cout<<"Area da parede: "<<areapar;
}
```

Resultado do exemplo::

```
        Area da parede: 18.2
```

No exemplo é declarada a classe **area** que contém quatro membros: **duas variáveis** do tipo float e **duas funções**: obtemdados e calcula. A função obtemdados tem apenas seu protótipo definido dentro da declaração da classe, as operações são especificadas fora dela, ao passo que a função calcula tem seus comandos definidos dentro da classe.

Para definir funções fora da classe deve ser usado o **operador de resolução de escopo (::)** após o nome da classe. Ele especifica a classe a qual pertence à função-membro. No exemplo tem-se:

```
void area :: obtemdados(float larg, float alt)
```

que significa que a função obtemdados é uma função-membro da classe area.

As funções-membros, também chamadas de **métodos**, definidas dentro da classe são criadas como **inline** por default, no entanto uma função-membro **inline** também pode ser definida fora da classe por meio da palavra reservada **inline**.

O exemplo traz duas seções **private:** e **public:** que especificam a visibilidade dos membros da classe. (Note-se que na codificação as palavras devem ser seguidas de dois pontos). Na parte **private**, geralmente, são escritos os membros de dados, os quais só podem ser acessados por funções pertencentes à classe. Desse modo, diz-se que os dados estão escondidos ou encapsulados por não permitir que funções extra classe os acessem. A palavra reservada private por ser default não precisaria ser explicitada no programa, contudo sua colocação torna mais evidente essa circunstância. Na seção **public** da classe, geralmente, são especificadas as funções ou métodos que atuam sobre os dados.

O acesso aos membros públicos é feito através do operador ponto de modo análogo ao acesso requerido aos membros de uma estrutura.

A instrução

```
parede.obtemdados(3.5, 5.2);
```

usa o operador ponto (**.**) para chamar a função obtemdados e atribuir valores aos dados membro.

Objetos

Um objeto é um tipo especial de variável que contêm dados e códigos e representa um elemento específico. Um objeto é uma instância ou uma ocorrência específica de uma classe. Ele geralmente possui vários **atributos** que definem suas características, e **métodos** que operam sobre esses atributos. No exemplo anterior, parede é o nome do objeto do tipo classe que contém os atributos largura e altura e os métodos obtemdados e calcula. A classe é o mecanismo que reúne objetos que podem ser descritos com os mesmos atributos e as mesmas operações. Ela agrupa dados e funções, é a peça fundamental da programação orientada a objeto. No exemplo, a codificação:

```
area parede;
```

define como objeto o elemento **parede** pertencente a classe **area**, ou seja, declara uma instância da classe.

Os objetos se comunicam através de **mensagem**, isto é, de chamada a uma **função-membro** requisitando um serviço através da execução de uma operação. A instrução

```
parede.obtemdados(3.5, 5.2);
```

solicita ao objeto parede que receba os dados ou as medidas para o cálculo da área.

Uma classe permite declarar vários objetos diferentes. Continuando com o mesmo exemplo que se acaba de examinar, será adicionado a este outro objeto à classe area e o programa modificado fica como segue:

```
#include <iostream.h>

class area        // define a classe
{
private:          // dados-membro:
        float largura;
        float altura;
public:           // funções-membro:
        void obtemdados(float larg, float alt);         // protótipo da função
        float calcula()                                 // função inline
        { return(largura*altura);}
};

void area::obtemdados(float larg, float alt)
{
        largura=larg;
        altura=alt;
}

void main()
{
        area parede, tijolo;      // declara os objetos

        parede.obtemdados(3.5, 5.2);
        float areapar = parede.calcula();
        cout<<"Area da parede: "<<areapar;

        tijolo.obtemdados(0.12, 0.30);
        float areatij = tijolo.calcula();
        cout<<"\nArea do tijolo: "<<areatij;

        cout<<"\nQuantidade de tijolos necessarios para construir a parede,";
        cout<<"\nconsiderando que 10% da area e' de argamassa: "
                <<(areapar/areatij*0.9)<<" tijolos."<<endl;
}
```

Resultado do exemplo::

```
        Area da parede: 18.2
        Area do tijolo: 0.036
        Quantidade de tijolos necessarios para construir a parede,
        considerando que 10% da area e' de argamassa: 455 tijolos.
```

Na instrução que define o objeto foi incluído outro objeto de nome tijolo:

```
area parede, tijolo;
```

Por meio do operador ponto, o objeto tijolo também acessa os mesmos métodos do objeto parede para atribuir a seus dados-membro os devidos valores, como no caso da instrução:

```
tijolo.obtemdados(0.12, 0.30);
```

O objeto tijolo possui as mesmas características do objeto parede, ou seja, uma área definida pela largura e altura.

O programa tem por objetivo determinar a quantidade de tijolos necessários para contruir uma parede considerando uma proporção da área constituída de argamassa.

FUNÇÕES CONSTRUTORAS E DESTRUTORAS

O processo de inicialização dos dados-membro da classe é simplificado pela definição de uma função especial chamada construtora e de forma semelhante, outra função especial denominada destrutora é executada automaticamente toda vez que um objeto for destruído ou liberado da memória.

Funções construtoras

Uma função construtora é uma função-membro especial que é chamada automaticamente pelo sistema sempre que um objeto for criado. Ela não retorna nenhum valor e na sua definição não é precedida do tipo void. Deve ser declarada com o mesmo nome da classe e é prontamente executada toda vez que uma nova instância da classe for criada a fim de inicilizar seus atributos.

Como exemplo, será modificado o programa anterior para implementar uma função construtora:

```
#include <iostream.h>

class area
{
private:
        float largura;
        float altura;
public:
        area(float larg, float alt);        // função construtora
        float calcula()
        { return(largura*altura);}
};

area :: area(float larg, float alt)
{
        largura=larg;
        altura=alt;
}

void main()
{
        area parede(3.5, 5.2), tijolo(0.12, 0.30);        // declara os objetos

        float areapar = parede.calcula();
        cout<<"Area da parede: "<<areapar;
```

```
        float areatij = tijolo.calcula();
        cout<<"\nArea do tijolo: "<<areatij;

        cout<<"\nQuantidade de tijolos necessarios para construir a parede,";
        cout<<"\nconsiderando que 10% da area e' de argamassa: "
                <<(areapar/areatij*0.9)<<" tijolos."<<endl;
}
```

Como pode ser visto, a função-membro obtemdados foi substituída pela função construtora que traz o nome da classe. Os argumentos são passados para a construtora no momento em que a instância da classe for criada.

Funções destrutoras

A função destrutora é uma função-membro especial que é executada automaticamente quando um objeto é liberado da memória. Ela é definida com o mesmo nome da classe precedida de um til (~). Como a construtora, a função destrutora não retorna valor e nem pode receber argumentos. O uso de destrutores é especialmente apropriado para liberar memória usada por um objeto que aloca memória dinamicamente.

A seguir, um exemplo da função destrutora implementado no modelo que vem sendo usado:

```
#include <iostream.h>

class area
{
private:
        float largura;
        float altura;
public:
        area(float larg, float alt);       // função construtora
        ~area();                           // função destrutora
        float calcula()
        { return(largura*altura);}
};

area::area(float larg, float alt)
{
        largura=larg;
        altura=alt;
}

area :: ~area()
{
        if (largura==3.5 && altura==5.2)
                cout<<"\nDestruindo o objeto parede";
        else
```

```
                cout<<"\nDestruindo o objeto tijolo";
}

void main()
{
        area parede(3.5, 5.2), tijolo(0.12, 0.30);      // declara os objetos

        float areapar = parede.calcula();
        cout<<"Area da parede: "<<areapar;

        float areatij = tijolo.calcula();
        cout<<"\nArea do tijolo: "<<areatij;

        cout<<"\nQuantidade de tijolos necessarios para construir a parede,";
        cout<<"\nconsiderando que 10% da area e' de argamassa: "
                <<(areapar/areatij*0.9)<<" tijolos."<<endl;
}
```

Neste exemplo a função destrutora incorpora um teste de seleção apenas para mostrar que ela está sendo executada e quais objetos estão sendo destruídos.

Sobrecarga de funções construtoras

Como qualquer outra, uma função construtora pode ser sobrecarregada apresentando o mesmo nome com número de argumentos diferentes. Supondo que se pretenda que o usuário insira as medidas no momento em que um objeto for criado e desejando manter a função construtora para os objetos definidos com valores iniciais, pode-se declarar uma nova função construtora, como mostrado no exemplo a seguir:

```
#include <iostream.h>

class area
{
private:
        float largura;
        float altura;
public:
        area(float larg, float alt);      // função construtora
        area();                           // função construtora sobrecarregada

        float calcula()
        { return(largura*altura);}
};

area::area(float larg, float alt)
{
        largura=larg;
        altura=alt;
```

```
}

area::area()
{
        cout<<"Insira as medidas - largura e altura: ";
        cin>>largura>>altura;
}

void main()
{
        area parede(3.5, 5.2), tijolo;

        float areapar = parede.calcula();
        cout<<"\nArea da parede: "<<areapar;

        float areatij = tijolo.calcula();
        cout<<"\nArea do tijolo: "<<areatij;

        cout<<"\n\nQuantidade de tijolos necessarios para construir a parede,";
        cout<<"\nconsiderando que 10% da area e' de argamassa: "
                <<(areapar/areatij*0.9)<<" tijolos."<<endl;
}
```

Resultado do exemplo::

```
        Insira as medidas - largura e altura: 0.1 0.2

        Area da parede: 18.2
        Area do tijolo: 0.2

        Quantidade de tijolos necessarios para construir a parede,
        considerando que 10% da area e' de argamassa: 819 tijolos.
```

Como o objeto tijolo foi declarado sem parâmetros, ele é inicializado com a função construtora sobrecarregada que não possui parâmetros. Assim será solicitado ao usuário que insira as medidas do objeto tijolo.

SOBRECARGA DE OPERADORES

A sobrecarga de um operador é o processo de modificar a funcionalidade desse operador para uso de uma determinada classe, com o propósito de melhorar a legibilidade do programa. Sobrecarregar um operador significa redefini-lo para aplicá-lo a tipos de dados definidos pelo usuário como estruturas e classes. A sobrecarga é estabelecida por meio da palavra-chave **operator** em funções chamadas operadoras. A função recebe como nome o símbolo do operador.

A seguir é mostrado um exemplo que utiliza o operador aritmético + binário para adicionar, por disciplina, as notas de duas disciplinas de três alunos que correspondem aos objetos a, b e c:

```
// sobrecarga do operador aritmético +
#include <iostream.h>

class nota
{
private:
        float n1, n2;
public:
        nota () {};                     // função construtora vazia
        nota (float,float);             // função construtora com argmentos
        nota operator + (nota);         // função operadora de adição
        void exibe();
};

nota::nota (float nt1, float nt2)
{
        n1 = nt1;
        n2 = nt2;
}

nota nota::operator + (nota m)
{
        float d1 = n1 + m.n1;
        float d2 = n2 + m.n2;
        return nota(d1,d2);
}

void nota::exibe()
{
        cout<<"Soma = ("<<n1<<", "<<n2<<')'<<endl;
}

void main ()
{
        nota a (8.5, 10.0);
```

```
        nota b (4.0,  6.5);
        nota c (6.0,  9.0);
        nota soma;

        soma = a + b + c;              // efetua a soma dos valores dos objetos

        soma.exibe();
}
```

Resultado do exemplo::

```
        Soma = (18.5, 25.5)
```

A função operadora declarada fora da classe e destacada a seguir:

```
nota nota::operator + (nota m)
{
        float d1 = n1 + m.n1;
        float d2 = n2 + m.n2;
        return nota(d1,d2);
}
```

recebe apenas um argumento e retorna o tipo nota. Os objetos da instrução

```
soma = a + b + c;
```

são passados como argumentos para a função operadora.

O próximo exemplo mostra a sobrecarga do operador – para a função operadora que trata string de caracteres. O programa inicialmente exibe um texto e depois retira do texto a letra C onde ela aparece.

```
// sobrecarga do operador -
#include <iostream.h>
#include <string.h>

class str
{
private:
        char texto[50];
public:
        str(char []);              // função construtora
        str operator - (char);
        void exibe();
};
str::str(char *car)
{
        strcpy(texto, car);
}
str str::operator - (char let)
{
```

```
        char temp[50];
//      for(int i=0, j=0; texto[i]; i++)
        for(int i=0, j=0; texto[i]!='\0'; i++)
                if(texto[i] != let)
                        temp[j++] = texto[i];
        temp[j]=NULL;
        return(strcpy(texto,temp));
}
void str::exibe()
{
        cout<<texto<<endl;
}
void main()
{
        str nome("Curso pratico de C++");
        nome.exibe();

        nome = nome - 'C';

        nome.exibe();
}
```

Resultado do exemplo::

```
Curso pratico de C++
urso pratico de ++
```

Como se pode ver, a sintaxe utilizada na instrução

```
nome = nome - 'C';
```

melhora sobremaneira a legibilidade do programa por apresentar o estilo de uma operação normal de atribuição, subtraindo do objeto nome a letra C. Lembrar que o propósito das sobrecargas de operadores é de tornar o programa claro e fácil de entender. No entanto, nem todos os operadores podem ser sobrecarregados. Operadores que não podem ser sobrecarregados: o operador ponto (**.**) de acesso a membros, o operador de resolução de escopo (**::**) e o operador de expressão condicional (**?:**).

BIBLIOGRAFIA

- ASCENCIO, Ana Fernanda Gomes, CAMPOS, Edilene Aparecida Veneruchi de. Fundamentos da programação de computadores : algoritmos, Pascal e C/C. São Paulo : Prentice Hall, 2002. xviii, 355p.
- FORBELLONE, André Luiz Villar; EBERSPACHER, Henri Frederico. Lógica de programação : a construção de algoritmos e estruturas de dados. 2.ed. São Paulo : Makron Books, 2000. 197p.
- JAMSA, Kris A. Aprendendo C. São Paulo : Makron Books do Brasil, 1999. 271p.
- MIZRAHI, Victorine Viviane. Treinamento em linguagem C. Sao Paulo : Makron, 1994. v.
- Complementar
- BERRY, John Thomas. Programando em C. Sao Paulo : Makron Books, 1991. xvi, 385p.
- UCCI, Waldir; SOUSA, Reginaldo Luiz; KOTANI, Alice Mayumi, et al. . Lógica de programação : os primeiros passos. 8.ed. Sao Paulo : Erica, 1999. 339p.

Eletrônica

- Algoritmos: http://www.manzano.pro.br/menu.html
- C++: http://www.cplusplus.com/doc/