

# **Engenharia**

## *Introdução à Informática*

# **UNISAL**

**Universidade Salesiana de Campinas**

**Prof. Gale / Prof. Tizzei**

## Introdução a Algoritmos

### Noções de Lógica

#### O que é Lógica?

Lógica trata da correção do pensamento. Como filosofia, ela procura saber por que pensamos assim e não de outro jeito. Com arte ou técnica, ela nos ensina a usar corretamente as leis do pensamento.

Poderíamos dizer também que a Lógica é a arte de pensar corretamente e, visto que a forma mais complexa do pensamento é o raciocínio, a Lógica estuda ou tem em vista a **correção do raciocínio**. Podemos ainda dizer que a lógica tem em vista a **ordem da razão**. Isto dá a entender que a nossa razão pode funcionar desordenadamente, pode pôr as coisas de pernas para o ar. Por isso a Lógica ensina a colocar **Ordem no Pensamento**.

Exemplos:

a) Todo o mamífero é animal.

Todo cavalo é mamífero.

Portanto, todo cavalo é animal.

b) Todo mamífero bebe leite.

O homem bebe leite.

Portanto, todo homem é mamífero e animal.

Existe Lógica no dia-a-dia?

Sempre que pensamos, o raciocínio e a lógica nos acompanham necessariamente. Também quando falamos, pois a palavra falada é a representação do pensamento; e, visto que a palavra escrita é a representação da palavra falada, também pensamos quando escrevemos, utilizando a Lógica ou a Ilógica. Daí percebemos a importância da Lógica na nossa vida não só na teoria, como também na prática, já que quando queremos pensar, falar ou escrever corretamente precisamos colocar em **Ordem o Pensamento**, isto é, utilizar a Lógica.

Exemplos:

a) A gaveta está fechada.

A bala está na gaveta.

Preciso primeiro abrir a gaveta, para depois pegar a bala.

b) João é mais velho que José.

Marcelo é mais novo que José.

Portanto, João é mais velho que Marcelo.

## Algoritmizando a Lógica

Construir algoritmos é o objetivo fundamental de toda a programação, mas, afinal,

O que é Algoritmo?

“Algoritmo é uma seqüência de passos que visam atingir um objetivo bem definido.”

(Ordem do Pensamento e, portanto, Lógica)

Apesar de achar este nome estranho, algoritmos são comuns em nosso cotidiano, como, por exemplo, uma receita de bolo. Nela está descrita uma série de ingredientes necessários, uma seqüência de diversos passos - ações - a serem cumpridos para que se consiga fazer determinado tipo de bolo - objetivo bem definido aguardado ansiosamente por todos. Para aprimorar nosso conceito de algoritmo, vamos tornar mais evidente alguns outros conceitos, como, por exemplo, o de ação:

“Ação é um acontecimento que a partir de um estado inicial, após um período de tempo finito, produz um estado final previsível e bem definido”, em que:

“Estado é a situação atual de dado objeto.”

Portanto, podemos redefinir Algoritmo como:

“Algoritmo é a descrição de um conjunto de ações que, obedecidas, resultam numa sucessão finita de passos, atingindo o objetivo.”

Em geral um algoritmo destina-se a resolver um problema: fixa um padrão de comportamento a ser seguido, uma norma de execução a ser trilhada, com o objetivo de alcançar a solução de um problema.

O que é padrão de comportamento?

Imagine a seguinte seqüência de números: 1, 6, 11, 16, 21, 26....

Para determinar o sétimo elemento da série, precisamos descobrir qual a sua regra de formatação, isto é, seu padrão de comportamento.

Para tal, observamos que a série obedece uma constância; visto que existe uma diferença constante entre cada elemento, a qual pode ser facilmente determinada, somos capazes de determinar o sétimo e qualquer outro termo.

Podemos, então, descrever uma atividade bem cotidiana, como, por exemplo, trocar uma lâmpada. Apesar de aparentemente óbvias demais, muitas vezes fazemos esse tipo de atividade inconscientemente. Sem percebermos seus pequenos detalhes. Vejamos se ela fosse descrita passo a passo:

- ⇒pegue a escada;
- ⇒posicione-a embaixo da lâmpada;
- ⇒busque uma lâmpada nova;
- ⇒suba na escada;
- ⇒retire a lâmpada velha;
- ⇒coloque a lâmpada nova.

Involuntariamente, já seguimos uma determinada seqüência de ações que, representadas neste algoritmo, fazem com que ele seja seguido naturalmente por pessoas, estabelecendo um padrão de comportamento.

É assim também com os algoritmos escritos para computador, você deve especificar todos os passos, para que o computador possa chegar ao objetivo.

Por exemplo:

Dados os números naturais(N)

0, 1, 2, 3, 4, 5, 6, ...

passo1    faça N igual a zero  
passo2    some 1 a N  
passo3    volte ao passo 2

Soma dos primeiros 100 números naturais:

passo1    faça N igual a zero  
passo2    some 1 a N  
passo3    se N for menor ou igual a 100  
          então volte ao passo 2  
          senão pare

Nos dois exemplos acima, o primeiro possui repertório bem definido mas não finito, enquanto que o segundo tem um critério de parada, ou seja, é finito e descreve um padrão de comportamento, ou seja, temos um algoritmo.

## Fases de um programa

Como tudo na terra, o programa tem um tempo de vida, chamado de ciclo de vida de um programa.

### Planejamento

É a fase onde definimos o problema a ser resolvido utilizando um computador. Nesta fase relacionamos a entrada e a saída do futuro programa, assim como a definição dos arquivos auxiliares que ele venha a utilizar.

### Projeto

É a fase onde a resolução do problema é concebida. Neste ponto são definidos detalhes do algoritmo, estrutura de dados empregados pelo programa.

### Escrita

Consiste em codificar o programa em uma linguagem de programação apropriada.

### Depuração

Ao final da escrita estaremos com o programa quase pronto; mas será que ele funciona? Esta é a fase onde depuramos o programa, ou seja, corrigimos os erros.

## Manutenção

Passada a fase de depuração, o programa será então liberado para utilização. Mas durante sua vida útil, um programa pode ser alterado; neste caso teremos que fazer novas mudanças, ou seja, manutenção.

## Programação Estruturada

É um método de projeto que tem por objetivo gerar um produto (programa), que tem certas características desejáveis, tais como:

1. **correção** - os programas devem antes de mais nada, dar respostas certas para entradas certas.
- 2.
3. **complitude** - os programas devem dar respostas inteligíveis para entradas erradas.
- 4.
5. **flexibilidade** - os possíveis erros de programação devem ser fáceis de serem removidos, e as alterações devido a mudanças devem ser implementadas facilmente.
- 6.
7. **eficiência** - programas devem ser eficientes quanto aos recursos computacionais como economia de memória, tempo de processamento.
- 8.
9. **transparente** - programas devem ser fáceis de serem compreendidos.

## Pseudo-linguagem

Escrever um algoritmo em português (portugol) visa principalmente facilitar o projetista, pensar no problema e não na máquina.

## Algoritmo X Qualidade

Todo algoritmo deve ser feito de maneira lógica e racional, visando principalmente a sua eficiência e clareza.

Ao construir algoritmos devemos:

1. Saber que estes serão lidos por outras pessoas, além de nós mesmos, permitindo sua fácil correção.
2. Escrever comentários na sua elaboração. Algoritmos sem comentários é sinal de amadorismo, e um dos grandes erros que programadores cometem. Sua elaboração deve ser clara e resumida, limitando-se às ocasiões de maior detalhamento. Devem acrescentar alguma coisa, não apenas frasear.
3. Os comandos nos dizem o que está sendo feito, os comentários dizem o **porquê**.
4. Todo algoritmo deve possuir comentários no prólogo, explicando o que ele faz e dar instruções para seu uso.
5. Utilizar espaços e/ou linhas em branco para melhorar a legibilidade.
10. Nomes representativos para variáveis. **“Uma seleção adequada de nomes de variáveis é o princípio mais importante da legibilidade de algoritmos”**.
11. Um comando por linha é suficiente.
12. Uso de parênteses aumenta a legibilidade e previne erros.
13. Utilize indentação, pois mostra a estrutura lógica do algoritmo. Deve ser feita segundo certos padrões estabelecidos.

## Método para construção de algoritmos

A. Ler atentamente o enunciado.

É justamente o enunciado do exercício que fornece o encaminhamento necessário à resolução do problema, que se torna, portanto, dependente de sua completa compreensão.

B. Retirar do enunciado a relação das entradas de dados.

C. Retirar do enunciado a relação das saídas de dados.

D. Determinar o que deve ser feito para transformar as entradas determinadas nas saídas específicas.

Nesta fase é que determinamos a construção de algoritmos propriamente dito, pois, a partir de alguns requisitos especificados, devemos determinar qual a seqüência de ações é capaz de transformar um conjunto definido de dados nas informações de resultado. Para isso, podemos:

**D.1.** Utilizar o Método Cartesiano quando a complexidade (variedade) não estiver totalmente absorvida, conhecida.

### Método Cartesiano

Nosso principal objetivo enquanto programadores é vencer a complexidade, o que mantém célebre a frase de Descartes “**Dividir para Conquistar**”. Este método consiste justamente em atacar o problema abrangente dividindo-o em partes menores, a fim de torná-lo mais simples ou específico e, se necessário, dividir novamente as partes não compreendidas.

Podemos esquematizar o seguinte procedimento (algoritmo) para o método:

- i. Dividir o problema em suas partes principais.
- ii. Analisar a divisão obtida para garantir coerência.
- iii. Se alguma parte não for bem compreendida, aplicar a ela o método.
- iv. Analisar o objeto para garantir entendimento e coerência.

**D.2.** Aplicar o Planejamento Reverso.

### Planejamento Reverso

Processo utilizado que, a partir das saídas (informações de resultado), procura desagregar, desmontando a informação, a fim de atingir os dados de entrada, quando então teríamos (do fim para o início) todas as ações.

**D.3.** Montar uma tabela de decisão quando uma ou mais ações dependentes de um conjunto de condições assumirem determinadas combinações de valores.

### Tabelas de decisão

Objetiva basicamente relacionar as ações que dependem de alguma condição com as próprias condições, a fim de esclarecer e visualizar facilmente quais valores o conjunto de condições deve assumir para que se efetue sua respectiva ação.

E. Construir o algoritmo.

F. Executar o algoritmo.

Implica executar todas as ações descritas seguindo o fluxo de execução estabelecido, verificando se os resultados obtidos correspondem ao esperado quando da montagem do algoritmo, detectando então algum possível erro no desenvolvimento deste. Essa atividade é

conhecida por “**teste de mesa**”.

## Exercícios

1. Um homem precisa atravessar um rio com um barco que possui capacidade de carregar apenas ele mesmo e mais uma de suas três cargas, que são: um lobo, um bode e um maço de alfafa. O que o homem deve fazer para conseguir atravessar o rio sem perder suas cargas?
2. Elabore um algoritmo que mova três discos de uma Torre de Hanói, que consiste em três hastes (a - b - c), uma das quais serve de suporte para três discos diferentes (1 - 2 - 3), os menores sobre os maiores. Pode-se mover um disco de cada vez para qualquer haste, contanto que nunca seja colocado um disco maior sobre um menor. O objetivo é transferir os três discos para outra haste.
3. Três jesuítas e três canibais precisam atravessar um rio; para tal, dispõem de um barco com capacidade para duas pessoas. Por medidas de segurança não se permite que em alguma margem a quantidade de jesuítas seja inferior à de canibais. Qual a seqüência de passos que permitiria a travessia com segurança?

## Conceitos Básicos

Em cada linguagem a frase de construção envolve dois aspectos:

- **a sintaxe** - forma como pode ser escrita;
- **a semântica** - conteúdo - a lógica de cada comando.

## Tipos Primitivos

As informações manipuladas pelo computador se apresentam através dos dados (informações) e das instruções (comandos).

Os dados possuem diferentes tipos, que podem ser classificados em quatro tipos primitivos:

1. **Inteiro:** valores de -32,768 até +32,768
2. **Real:** valores com vírgulas negativos ou positivos com grande abrangência.
3. **Caracter:** toda e qualquer informação composta de caracteres alfanuméricos (0..9, a..z, A..Z) e/ou caracteres especiais (!, @, #, \$, %, ^, &, \*, etc). Estes caracteres (podendo ser um só ou uma cadeia) aparecerão sempre entre apóstrofes, ou aspas. Exemplo: "ASD", '2e3', "V".
4. **Lógico:** também conhecido como tipo Booleano, toda e qualquer informação que pode assumir somente dois valores (aberto/fechado, acesso/apagado), no nosso caso apenas pode-se usar **V** (verdadeiro) ou **F** (falso).

### Exercícios:

1) Classifique os dados de acordo com o seu tipo, sendo (I = inteiro, R = real, C = caracter, L = lógico).

- ( ) 0            ( ) + 36            ( ) "+3257"    ( ) F  
( ) 1            ( ) + 32            ( ) "+3257"    ( ) 'F'  
( ) 0,0            ( ) - 0,001        ( ) "-0,0"      ( ) ".V."  
( ) 0            ( ) + 0,05        ( ) ".V."        ( ) F  
( ) -1            ( ) + 3257        ( ) V            ( ) -32  
( ) "a"            ( ) "abc"          ( ) -1,9E123    ( ) '0'

## Formação de Identificadores

Podemos imaginar a memória como sendo um armário repleto de gavetas, no qual as gavetas seriam os locais físicos responsáveis por armazenar objetos; os objetos (que podem ser substituídos ou não) seriam as informações (dados), e as gavetas as variáveis ou constantes.

Visto que na memória (armário) existem inúmeras variáveis (gavetas), precisamos diferenciá-las, o que é feito por meio de identificadores (etiquetas). Cada variável (gaveta), no entanto, pode guardar apenas uma informação (objeto) de cada vez, sendo sempre do mesmo tipo (material).

Os identificadores possuem certas regras para serem formados: Devem começar por um caracter alfabético;

5. Podem ser seguidos por mais caracteres alfabéticos e/ou alfanuméricos;

6. Não é permitido o uso de caracteres especiais ou espaços, com exceção do underscore( \_).

Variáveis e Constantes

São entidades que armazenam valores. A diferença entre variáveis e constantes está na possibilidade de alteração do valor armazenado, durante todo o tempo de duração do programa.

## Variáveis

Entendemos que uma informação é variável quando sofre variações durante o decorrer do tempo.

A declaração de variáveis é feita da seguinte maneira:

Variáveis

dólar : real

endereço : caracter

existe : lógico

## Exercício

Assinale os identificadores válidos:

- |                      |           |                 |            |
|----------------------|-----------|-----------------|------------|
| 1 - abc              | 2 - AB/C  | 3 - “João”      | 4 - [x]    |
| 5 - 123 <sup>a</sup> | 6 - 080   | 7 - 1 a 3       | 8 - (x)    |
| 9 - #55              | 10 - AH!  | 11 - Etc...     | 12 - ...a  |
| 13 - BAC             | 14 - xyz  | 15 - Porta_mala | 16 - A_B-C |
| 17 - U2              | 18 - p{0} | 19 - A123       | 20 - A.    |

## Expressões

É um conjunto de constantes e/ou variáveis e/ou funções ligadas por operadores aritméticos ou lógicos.

### Expressões Aritméticas

Denominamos expressão aritmética aquela cujos operadores são aritméticos e cujos operandos são constantes e/ou variáveis do tipo numérico (inteiro e/ou real).

## Operadores Aritméticos

Chamamos de operadores aritméticos o conjunto de símbolos que representa as operações básicas da matemática:

Operadores Binários:

- ☞ **+** adição
- ☞ **\*** multiplicação
- ☞ **\*\*** potenciação
- ☞ **-** subtração
- ☞ **/** divisão
- ☞ **//** radicação
- ☞ **mod** resto da divisão
- ☞ **div** quociente da divisão inteira

Exemplo:

$$9/4 = 2,25$$

$$9 \text{ div } 4 = 2$$

$$9 \text{ mod } 4 = 25$$

$$15 \text{ div } 7 = 2$$

$$15 \text{ mod } 7 = 1$$

## Funções Matemáticas

Além das operações aritméticas básicas anteriormente citadas, podemos usar nas expressões aritméticas algumas funções da matemática:

- ☞ **sen(x)** - seno de x;
- ☞ **cos(x)** - coseno de x;
- ☞ **tg(x)** - tangente de x;
- ☞ **arctg(x)** - arco cuja tangente é x;
- ☞ **arccos(x)** - arco cujo coseno é x;
- ☞ **arcsen(x)** - arco cujo seno é x;
- ☞ **abs(x)** - valor absoluto (módulo) de x;
- ☞ **int(x)** - a parte inteira de um número fracionário;
- ☞ **frac(x)** - a parte fracionária de x;
- ☞ **ard(x)** - transforma por arredondamento, um número fracionário em inteiro;
- ☞ **sinal(x)** - fornece o valor -1, +1 ou 0 conforme o valor de x seja negativo, positivo ou nulo;
- ☞ **rnd(x)** - valor randômico de x;

Onde x pode ser um número, variável, expressão aritmética ou também outra função matemática.



## Operadores Lógicos

Utilizaremos três conectivos básicos para a formação de novas proposições a partir de outras já conhecidas. Os operadores lógicos são:

- ☞ **e** conjunção
- ☞ **ou** disjunção não exclusiva
- ☞ **xou** disjunção exclusiva
- ☞ **não** negação

Exemplo:

a) não  $2^{**}3 < 4^{**}2$  ou  $\text{abs}(\text{int}(15/-2)) < 10$   
não  $8 < 16$  ou  $\text{abs}(\text{int}(-7,5)) < 10$   
não F ou  $\text{abs}(-7) < 10$   
V ou  $7 < 10$   
V ou V  
V

## Comandos de Atribuição

Um comando de atribuição permite-nos fornecer um valor a uma certa variável, onde o tipo dessa informação deve ser compatível com o tipo da variável, isto é, somente podemos atribuir um valor lógico a uma variável capaz de comportá-lo, ou seja, uma variável declarada do tipo lógico.

O comando de atribuição é uma seta apontando para a variável ou dois pontos e o sinal de igual ( := ou ← ):

Exemplo:

**variáveis**

A, B : lógico;

X : inteiro;

A := V;

X := 8 + 13 div 5;

B := 5 = 3;

## Comandos de Entrada e Saída de Dados

### Entrada de Dados

Para que nossos algoritmos funcionem, em quase todos os casos precisaremos de informações que serão fornecidas somente após o algoritmo pronto, e que sempre estarão mudando de valores, para que nossos algoritmos recebem estas informações,

devemos então construir entradas de dados, pelas quais o usuário (pessoa que utilizar o programa) poderá fornecer todos os dados necessários.

A sintaxe do comando de entrada de dados é a seguinte:

**leia** (variável);

**leia** ("Entre com o valor de var1 e var 2: ", var1, var2);

## **Saída de Dados**

Da mesma forma que nosso algoritmo precisa de informações, o usuário precisa de respostas as suas perguntas, para darmos estas respostas usamos um comando de saída de dados para informar a resposta.

A sintaxe do comando de saída de dados é a seguinte:

**escreva** (variável);

**escreva** ("cadeia de caracteres" );

**escreva** ("cadeia", variável);

**escreva** (número, "cadeia", variável);

## **Blocos**

Delimitam um conjunto de ações com uma função definida; neste caso, um algoritmo pode ser definido como um bloco.

Exemplo:

**início** início do algoritmo

seqüência de ações

**fim.** Fim do bloco (algoritmo)

## Estruturas de Controle

### Estrutura Seqüencial

É o conjunto de ações primitivas que serão executadas numa seqüência linear de cima para baixo e da esquerda para direita, isto é, na mesma ordem em que foram escritas. Como podemos perceber, todas as ações devem ser seguidas por um ponto-e-vírgula (;), que objetiva separar uma ação de outra.

#### **Variáveis;**

(declaração de variáveis);

#### **início**

comando 1;

comando 2;

comando 3;

#### **fim.**

Exemplo:

Construa um algoritmo que calcule a média aritmética entre quatro notas quaisquer fornecidas pelo usuário.

Resolvendo através do Método de Construção de Algoritmos, temos:

- 1.Dados de entrada: quatro notas bimestrais (N1, N2, N3, N4);
- 2.Dados de saída: média aritmética anual;
- 3.O que devemos fazer para transformar quatro notas bimestrais em uma média anual?
- 4.Resposta: utilizar média aritmética.
- 5.O que é média aritmética?
- 6.Resposta: a soma dos elementos divididos pela quantidade deles. Em nosso caso particular:  $(N1 + N2 + N3 + N4)/4$ ;
- 7.Construindo o algoritmo:

**variáveis** (declaração das variáveis)

N1, N2, N3, N4, MA : **real**;

**início** (começo do algoritmo)

**conheça** (N1, N2, N3, N4); {entrada de dados}

Ma :=  $(N1 + N2 + N3 + N4) / 4$ ; {processamento}

**escreva** ("A média anual e: ", MA); {saída de dados}

**fim.**

### Exercícios

1. Dados dois números inteiros, achar a média aritmética entre eles.
2. Dados dois números inteiros, trocar o conteúdo desses números.
3. Dados três notas inteiras e seus pesos, encontrar a média ponderada entre elas.
4. Calcular a área de um triângulo reto.
5. Escreva um algoritmo que calcule:  $C = (A + B) * B$ .
6. Faça um algoritmo que calcule o valor a ser pago em uma residência, que consumiu uma quantidade X de energia elétrica.
7. Faça um algoritmo para calcular e imprimir a tabuada.
8. Fazer a transformação de um valor em dólar, para a moeda corrente do Brasil.

## Estruturas de Seleção ou Decisão

Uma estrutura de decisão permite a escolha de um grupo de ações e estruturas a ser executado quando determinadas condições, representadas por expressões lógicas, são ou não satisfeitas.

### Decisão Simples

**Se** <condição> **então**

**início** {bloco verdade}

C; {comando único (ação primitiva)}

**fim** {bloco}

<condição> é uma expressão lógica, que, quando inspecionada, pode gerar um resultado falso ou verdadeiro.

Se V, a ação primitiva sob a cláusula será executada; caso contrário, encerra o comando, neste caso, sem executar nenhum comando.

Exemplo

**variáveis**

altura1, altura2 : **real**

**início**

**leia** (altura1, altura2);

**se** altura1 > altura2 **então**

**início**

**escreva**('Altura 1 maior');

**fim** {se}

**fim.** {algoritmo}

### Exercícios

1. Faça um algoritmo que conheça as quatro notas bimestrais e, informe a média do aluno e se ele passou; média para aprovação = 6.
2. Conheça três números inteiros, e informe qual é o maior.

3. Encontrar o dobro de um número se este for par, se ímpar encontrar o triplo.

### Decisão Composta

**Se** <condição> **então**

**início**

C;

B;

**fim;**

**senão**

**início**

A;

**fim;**

Notamos agora que se a condição for satisfeita (Verdadeira), os comandos C e B serão executados, mas se a condição for falsa, também podem ser executados comandos, neste caso o comando A entrando no senão.

### Exercícios

1. Conhecendo-se três números, encontrar o maior.
2. Dados três números inteiros, colocá-los em ordem crescente.

### Seleção Múltipla

**Escolha** variável

**caso** valor1 : comando1;

**caso** valor2 : comando2;

**caso** valor3 : comando3;

**caso** valor4 : comando4;

**caso contrário** comando\_F

**fimescolha;**

Esta estrutura evita que façamos muitos blocos **se**, quando o teste será sempre em cima da mesma variável.

Exemplo:

**se** X = V1 **então início**

C1;

**fim**

**senão início**

**se X = V2 então início**

C2;

**fim;**

**senão início**

**se X = V3 então início**

C2;

**fim**

**senão início**

**se X = V4 então início**

C3;

**fim**

**senão início**

C4;

**fim;**

**fim;**

**fim;**

**fim;**

Com a estrutura de escolha múltipla, o algoritmo ficaria da seguinte maneira:

**escolha X**

**caso V1 : C1;**

**caso V2 : C2;**

**caso V3 : C2;**

**caso V4 : C3;**

**senão C4;**

**fimescolha;**

## Exercício

1. Numa festinha de fim de curso, foi feito um sorteio para distribuir o dinheiro restante em caixa. Dez pessoas foram sorteadas com direito a tentar a sorte mais uma vez, da seguinte forma: Deveriam apanhar uma bola numerada de 0 a 9 e conforme o algarismo sorteado o prêmio seria:

Número da Bola	Prêmio (% do valor do caixa)
0	05
1	25

2	10
3	07
4	08
5	05
6	15
7	12
8	03
9	10

Faça um algoritmo que calcule o prêmio recebido individualmente por pessoa.

2. Considerando três notas inteiras, encontrar a média aritmética simples entre as que correspondem a números pares.
3. Dados 4 números, colocá-los em ordem crescente.
4. Conhecer a idade de três pessoas, informar quem é o mais velho e quem é o mais novo.

5. Sendo conhecidos os valores de Z e W encontrar:

$$y = (7x^2 - 3x - 8t) / 5(x + 1)$$

sabendo-se que os valores de x são assim definidos:

se  $w > 0$  ou  $z < 7$

$$x = (5w + 1) / 3;$$

$$t = (5z + 2);$$

caso contrário

$$x = (5z + 2);$$

$$t = (5w + 1) / 3;$$

## Estruturas de Repetição

Estas estruturas possibilitam que nosso algoritmo seja muito mais enxuto e fácil de se programar. Imagine um algoritmo de fatorial de 8:

### variáveis

fat : real;

### início

fat := 8 \* 7;

fat := fat \* 6 ;

fat := fat \* 5;

```
fat := fat * 4;  
fat := fat * 3;  
fat := fat * 2;  
escreva(fat);
```

**fim.**

O resultado será o fatorial com certeza mas, imagine se fosse o fatorial de 250. Ou ainda, o usuário deseja fornecer o número e o algoritmo deve retornar o fatorial, qual número será digitado? Quantas linhas deverão ser escritas?

Para isso servem as estruturas de repetição, elas permitem que um determinado bloco de comandos seja repetido várias vezes, até que uma condição determinada seja satisfeita.

### **Repita ... Até - Estrutura com teste no final**

Esta estrutura faz seu teste de parada após o bloco de comandos, isto é, o bloco de comandos será repetido, até que a condição seja **V**. Os comandos de uma estrutura repita .. até sempre serão executados pelo menos uma vez.

#### **Repita**

```
comando1;  
comando2;  
comando3;  
até <condição>;
```

### **Exercício**

1. Construa o algoritmo de cálculo do fatorial com a estrutura Repita .. Até.

### **Enquanto .. Faça - Estrutura com teste no Início**

Esta estrutura faz seu teste de parada antes do bloco de comandos, isto é, o bloco de comandos será repetido, até que a condição seja **F**. Os comandos de uma estrutura **enquanto .. faça** poderão ser executados uma vez, várias vezes ou nenhuma vez.

#### **Enquanto** < condição > **Faça**

#### **início**

```
< bloco de comandos >;
```

**fim;**

### **Exercício**

1. Construa o algoritmo de cálculo do fatorial com a estrutura Enquanto .. Faça.

## Para .. Passo - Estrutura com variável de Controle

Nas estruturas de repetição vistas até agora, ocorrem casos em que se torna difícil determinar quantas vezes o bloco será executado. Sabemos que ele será executado enquanto uma condição for satisfeita - **enquanto..faça**, ou até que uma condição seja satisfeita - **repita...até**. A estrutura **para .. passo** repete a execução do bloco um número definido de vezes, pois ela possui limites fixos:

**Para** <variável> := <valor> **até** <valor> **passo** N **faça**

**início**

< bloco de comandos >;

**fim;**

## Exercício

1. Construa o algoritmo de cálculo do fatorial com a estrutura Para .. Passo.
2. Dado um conjunto de números inteiros, obter a soma e a quantidade de elementos.
3. Encontrar os N primeiros termos de uma progressão geométrica, onde o primeiro termo e a razão são conhecidos.
4. Idem ao exercício 1, calculando também a soma dos números negativos, positivos, pares e ímpares.
5. Determinar o menor entre um conjunto de números inteiros fornecidos um de cada vez.
6. A conversão de graus Fahrenheit para centígrados é obtida pela fórmula  $C = 5/9 (F-32)$ .  
Escreva um algoritmo que calcule e escreva uma tabela de graus centígrados em função de graus Fahrenheit que variem de 50 a 150 de 1 em 1.
7. Prepare um algoritmo que calcule o valor de H, sendo que ele é determinado pela série
8.  $H = 1/1 + 3/2 + 5/3 + 7/4 + \dots + 99/50$ .
9. Elabore um algoritmo que determine o valor de S, onde:
10.  $S = 1/1 - 2/4 + 3/9 - 4/16 + 5/25 - 6/36 \dots - 10/100$
11. Escreva um algoritmo que calcule e escreva a soma dos dez primeiros termos da seguinte série:  $2/500 - 5/450 + 2/400 - 5/350 + \dots$
12. Construa um algoritmo que calcule o valor aproximado de PI utilizando a fórmula
13.  $PI = 3/(H*32)$ , onde:  $H = 1/1^{**3} - 1/3^{**3} + 1/5^{**3} - 1/7^{**3} + 1/9^{**3} - \dots$
14. Fulano tem 1,50 metro e cresce 2 cm pôr ano, enquanto Ciclano tem 1,10 metro e cresce 3 cm por ano. Construa um algoritmo que calcule e imprima quantos anos serão para que Ciclano seja maior que Fulano.
15. Em uma eleição presidencial, existem quatro candidatos. Os votos são informados através de código. Os dados utilizados para a escrutinagem obedecem à seguinte codificação:
  - 1, 2, 3, 4 = voto para os respectivos candidatos;
  - 5 = voto nulo;
  - 6 = voto em branco;Elabore um algoritmo que calcule e escreva: total de votos para cada candidato;
  - total de votos nulos;
  - total de votos em branco;
  - percentual dos votos em branco e nulos sobre o total;
  - situação do candidato vencedor sobre os outros três, no caso, se ele obteve ou não mais votos que os outros somados;Como finalizador do conjunto de votos, tem-se o valor 0.

14. Realizou-se uma pesquisa para determinar o índice de mortalidade infantil em um certo período. Construa um algoritmo que leia o número de crianças nascidas no período e, depois, num número indeterminado de vezes, o sexo de uma criança morta (masculino, feminino) e o número de meses da vida da criança.

Como finalizador, teremos a palavra "fim" no lugar do sexo da criança.

Determine e imprima:

- a porcentagem de crianças mortas no período;
- a porcentagem de crianças do sexo masculino mortas no período;
- a porcentagem de crianças que viveram dois anos ou menos no período.

15. Suponha que exista um prédio sem limites de andares, ou seja, um prédio infinito, onde existam três elevadores, denominados A, B e C. Para otimizar o sistema de controle dos elevadores, foi realizado um levantamento no qual cada usuário respondia:

- o elevador que utilizava com maior frequência;
- o andar ao qual se dirigia;
- o período que utilizava o elevador, entre:
  - M = matutino;
  - V = vespertino;
  - N = noturno.

Construa um algoritmo que calcule e imprima:

- qual é o andar mais alto a ser utilizado;
- qual é o elevador mais freqüentado e em que horário se encontra seu maior fluxo;
- qual o horário mais usado de todos e a que elevador pertence;
- qual a diferença percentual entre o mais usado dos horários e o menos usado (especificando qual o menos usado);
- qual a porcentagem sobre o total de serviços prestados do elevador de média utilização.

## **INTRODUÇÃO – Linguagem C**

Vamos, neste curso, aprender os conceitos básicos da linguagem de programação C a qual tem se tornado cada dia mais popular, devido à sua versatilidade e ao seu poder. Uma das grandes vantagens do C é que ele possui tanto características de "alto nível" quanto de "baixo nível".

Apesar de ser bom, não é pré-requisito do curso um conhecimento anterior de linguagens de programação. É importante uma familiaridade com computadores. O que é importante é que você tenha vontade de aprender e dedicação ao curso.

O C nasceu na década de 70. Seu inventor, Dennis Ritchie, implementou-o pela primeira vez usando um DEC PDP-11 rodando o sistema operacional UNIX. O C é

derivado de uma outra linguagem: o B, criado por Ken Thompson. O B, por sua vez, veio da linguagem BCPL, inventada por Martin Richards.

O C é uma linguagem de programação genérica que é utilizada para a criação de programas diversos como processadores de texto, planilhas eletrônicas, sistemas operacionais, programas de comunicação, programas para a automação industrial, gerenciadores de bancos de dados, programas de projeto assistido por computador, programas para a solução de problemas da Engenharia, Física, Química e outras Ciências, etc ... É bem provável que o Navegador que você utiliza hoje tenha sido escrito em C ou C++.

Estudaremos a estrutura do ANSI C, o C padronizado pela ANSI. Veremos ainda algumas funções comuns em compiladores para alguns sistemas operacionais. Quando não houver equivalentes para as funções em outros sistemas, apresentaremos formas alternativas de uso dos comandos.

Sugerimos que o aluno realmente use o máximo possível dos exemplos, problemas e exercícios aqui apresentados, gerando os programas executáveis com o seu compilador. Quando utilizamos o compilador aprendemos a lidar com mensagens de aviso, mensagens de erro, bugs, etc. Apenas ler os exemplos não basta. O conhecimento de uma linguagem de programação transcende o conhecimento de estruturas e funções. O C exige, além do domínio da linguagem em si, uma familiaridade com o compilador e experiência em achar "bugs" nos programas. É importante digitar, compilar e executar os exemplos apresentados.

## Primeiros Passos

### O C é "Case Sensitive"

Vamos começar o nosso curso ressaltando um ponto de suma importância: o C é "Case Sensitive", isto é, *maiúsculas e minúsculas fazem diferença*. Se declarar uma variável com o nome soma ela será diferente de **Soma**, **SOMA**, **SoMa** ou **sOmA**. Da mesma maneira, os comandos do C **if** e **for**, por exemplo, só podem ser escritos em minúsculas pois senão o compilador não irá interpretá-los como sendo comandos, mas sim como variáveis.

### Dois Primeiros Programas

Vejamos um primeiro programa em C:

```
#include <stdio.h>
/* Um Primeiro Programa */
void main (void)
{
    printf ("Ola! Eu estou vivo!\n");
}
```

Compilando e executando este programa você verá que ele coloca a mensagem *Ola! Eu estou vivo!* na tela.

### Vamos analisar o programa por partes.

A linha **#include <stdio.h>** diz ao compilador que ele deve incluir o arquivo-cabeçalho **stdio.h**. Neste arquivo existem declarações de funções úteis para entrada e saída de dados (std = standard, padrão em inglês; io = Input/Output, entrada e saída ==> stdio = Entrada e saída padronizadas). Toda vez que você quiser usar uma destas funções deve-se incluir este comando. O C possui diversos Arquivos-cabeçalho.

Quando fazemos um programa, uma boa idéia é usar comentários que ajudem a elucidar o funcionamento do mesmo. No caso acima temos um comentário: **/\* Um Primeiro Programa \*/**. O compilador C desconsidera qualquer coisa que esteja começando com **/\*** e terminando com **\*/**. Um comentário pode, inclusive, ter mais de uma linha.

A linha **void main(void)** indica que estamos definindo uma função de nome **main**. Todos os programas em C têm que ter uma função **main**, pois é esta função que será chamada quando o programa for executado. O conteúdo da função é delimitado por chaves **{ }**. O código que estiver dentro das chaves será executado seqüencialmente quando a função for chamada.

A única coisa que o programa *realmente* faz é chamar a função **printf()**, passando a string (uma string é uma seqüência de caracteres, como veremos brevemente) **"Ola! Eu estou vivo!\n"** como argumento. É por causa do uso da função **printf()** pelo programa que devemos incluir o arquivo-cabeçalho **stdio.h**. A função **printf()** neste caso irá apenas colocar a string na tela do computador. O **\n** é uma constante chamada de *constante barra invertida*. No caso, o **\n** é a constante barra invertida de "new line" e ele é interpretado como um comando de mudança de linha, isto é, após imprimir *Ola! Eu estou*

*vivo!* o cursor passará para a próxima linha. É importante observar também que os *comandos* do C terminam com ; .

Podemos agora tentar um programa mais complicado:

```
#include <stdio.h>
void main (void)
{
    int Dias;          /* Declaracao de Variaveis */
    float Anos;
    printf ("Entre com o número de dias: "); /* Entrada de Dados */
    scanf ("%d",&Dias);
    Anos=Dias/365.25;   /* Conversao Dias->Anos */
    printf ("\n\n%d dias equivalem a %f anos.\n",Dias,Anos);
}
}
```

Vamos entender como o programa acima funciona. São declaradas duas variáveis chamadas **Dias** e **Anos**. A primeira é um **int** (inteiro) e a segunda um **float** (ponto flutuante). As variáveis declaradas como ponto flutuante existem para armazenar números que possuem casas decimais, como 5,1497.

É feita então uma chamada à função **printf()**, que coloca uma mensagem na tela.

Queremos agora ler um dado que será fornecido pelo usuário e colocá-lo na variável inteira **Dias**. Para tanto usamos a função **scanf()**. A string **"%d"** diz à função que iremos ler um inteiro. O segundo parâmetro passado à função diz que o dado lido deverá ser armazenado na variável **Dias**. É importante ressaltar a necessidade de se colocar um **&** antes do nome da variável a ser lida quando se usa a função **scanf()**. O motivo disto só ficará claro mais tarde. Observe que, no C, quando temos mais de um parâmetro para uma função, eles serão separados por vírgula.

Temos então uma expressão matemática simples que atribui a **Anos** o valor de **Dias** dividido por 365.25 (365.25 é uma constante ponto flutuante 365,25). Como **Anos** é uma variável **float** o compilador fará uma conversão automática entre os tipos das variáveis (veremos isto com detalhes mais tarde).

A segunda chamada à função **printf()** tem três argumentos. A string **"\n\n%d dias equivalem a %f anos.\n"** diz à função para pular duas linhas, colocar um inteiro na tela, colocar a mensagem " **dias equivalem a** ", colocar um valor **float** na tela, colocar a mensagem " **anos.**" e pular outra linha. Os outros parâmetros são as variáveis, **Dias** e **Anos**, das quais devem ser lidos os valores do inteiro e do **float**, respectivamente.

## Introdução Básica às Entradas e Saídas

### - Caracteres

Os caracteres são um tipo de dado: o **char**. O C trata os caracteres ('a', 'b', 'x', etc ...) como sendo variáveis de um *byte* (8 *bits*). Um *bit* é a menor unidade de armazenamento de informações em um computador. Os inteiros (**ints**) têm um número maior de *bytes*. Dependendo da implementação do compilador, eles podem ter 2 *bytes* (16 *bits*) ou 4 *bytes* (32 *bits*). Na linguagem C, também podemos usar um **char** para armazenar valores numéricos inteiros, além de usá-lo para armazenar caracteres de

texto. Para indicar um caractere de texto usamos apóstrofes. Veja um exemplo de programa que usa caracteres:

```
#include <stdio.h>
void main ()
{
    char Ch;
    Ch='D';
    printf ("%c",Ch);
}
```

No programa acima, **%c** indica que **printf()** deve colocar um caractere na tela. Como vimos anteriormente, um **char** também é usado para armazenar um número inteiro. Este número é conhecido como o código ASCII correspondente ao caractere. Veja o programa abaixo:

```
#include <stdio.h>
void main ()
{
    char Ch;
    Ch='D';
    printf ("%d",Ch); /* Imprime o caracter como inteiro */
}
```

Este programa vai imprimir o número 68 na tela, que é o código ASCII correspondente ao caractere 'D' (d maiúsculo).

Muitas vezes queremos ler um caractere fornecido pelo usuário. Para isto as funções mais usadas, quando se está trabalhando em ambiente DOS ou Windows, são **getch()** e **getche()**. Ambas retornam o caractere pressionado. **getche()** imprime o caractere na tela antes de retorná-lo e **getch()** apenas retorna o caractere pressionado sem imprimí-lo na tela. Ambas as funções podem ser encontradas no arquivo de cabeçalho **conio.h**. Geralmente estas funções **não estão disponíveis em ambiente Unix** (compiladores cc e gcc), pois não fazem parte do padrão ANSI. Podem ser substituídas pela função scanf(), porém sem as mesmas funcionalidades. Eis um exemplo que usa a função **getch()**, e seu correspondente em ambiente Unix:

```
#include <stdio.h>
#include <conio.h>
/* Este programa usa conio.h . Se você não tiver a conio, ele não
funcionará no Unix */
void main ()
{
    char Ch;
    Ch=getch();
    printf ("Voce pressionou a tecla %c",Ch);
}
```

Equivalente ANSI-C para o ambiente Unix do programa acima, sem usar getch():

```
#include <stdio.h>
void main ()
{
    char Ch;
    scanf ("%c", &Ch);
}
```

```
printf ("Voce pressionou a tecla %c",Ch);
}
```

A principal diferença da versão que utiliza `getch()` para a versão que não utiliza `getch()` é que no primeiro caso o usuário simplesmente aperta a tecla e o sistema lê diretamente a tecla pressionada. No segundo caso, é necessário apertar também a tecla <ENTER>. **Lembre-se que, se você quiser manter a portabilidade de seus programas, não deve utilizar as funções `getch` e `getche`, pois estas não fazem parte do padrão ANSI C !!!**

Vamos agora fazer uma abordagem inicial às duas funções que já temos usado para fazer a entrada e saída.

### - printf

A função **printf()** tem a seguinte forma geral:

```
printf (string_de_controle,lista_de_argumentos);
```

Teremos, na string de controle, uma descrição de tudo que a função vai colocar na tela. A string de controle mostra não apenas os caracteres que devem ser colocados na tela, mas também quais as variáveis e suas respectivas posições. Isto é feito usando-se os códigos de controle, que usam a notação %. Na string de controle indicamos quais, de qual tipo e em que posição estão as variáveis a serem apresentadas. É muito importante que, para cada código de controle, tenhamos um argumento na lista de argumentos. Apresentamos agora alguns dos códigos %:

Código	Significado
%d	Inteiro
%f	Float
%c	Caractere
%s	String
%%	Coloca na tela um %

Vamos ver alguns exemplos de **printf()** e o que eles exibem:

```
printf ("Teste %% %") -> "Teste % %"
printf ("%f",40.345) -> "40.345"
printf ("Um caractere %c e um inteiro %d", 'D',120) -> "Um
caractere D e um inteiro 120"
printf ("%s e um exemplo", "Este") -> "Este e um exemplo"
printf ("%s%d%", "Juros de ",10) -> "Juros de 10%"
```

### - scanf

O formato geral da função **scanf()** é:

```
scanf (string-de-controle,lista-de-argumentos);
```

Usando a função **scanf()** podemos pedir dados ao usuário. Devemos ficar atentos a fim de colocar o mesmo número de argumentos que o de códigos de controle na string de controle. Outra coisa importante é lembrarmos de colocar o & antes das variáveis da lista de argumentos.

```
scanf ("%d",&a); // variável inteira  
scanf ("%f",&b); // variável real (float)
```

## Introdução a Alguns Comandos de Controle de Fluxo

Os comandos de controle de fluxo são aqueles que permitem ao programador alterar a sequência de execução do programa. Vamos dar uma breve introdução a dois comandos de controle de fluxo. Outros comandos serão estudados posteriormente.

### - if

O comando **if** representa uma tomada de decisão do tipo "SE isto ENTÃO aquilo". A sua forma geral é:

*if (condição) declaração;*

A condição do comando **if** é uma expressão que será avaliada. Se o resultado for zero a declaração não será executada. Se o resultado for qualquer coisa diferente de zero a declaração será executada. A declaração pode ser um bloco de código ou apenas um comando. É interessante notar que, no caso da declaração ser um bloco de código, não é necessário (e nem permitido) o uso do ; no final do bloco. Isto é uma regra geral para blocos de código. Abaixo apresentamos um exemplo:

```
#include <stdio.h>  
void main ()  
{  
    int num;  
    printf ("Digite um numero: ");  
    scanf ("%d",&num);  
    if (num>10) printf ("\n\nO numero e maior que 10");  
    if (num==10)  
    {  
        printf ("\n\nVoce acertou!\n");  
        printf ("O numero e igual a 10.");  
    }  
    if (num<10) printf ("\n\nO numero e menor que 10");  
}
```

No programa acima a expressão **num>10** é avaliada e retorna um valor diferente de zero, se verdadeira, e zero, se falsa. No exemplo, se num for maior que 10, será impressa a frase: "O número e maior que 10". Repare que, se o número for igual a 10, estamos executando dois comandos. Para que isto fosse possível, tivemos que agrupá-los em um bloco que se inicia logo após a comparação e termina após o segundo printf. Repare também que quando queremos testar igualdades usamos o operador **==** e não **=**. Isto porque o operador **=** representa *apenas* uma atribuição. Pode parecer estranho à primeira vista, mas se escrevêssemos

```
if (num=10) ... /* Isto esta errado */
```

o compilador iria *atribuir* o valor 10 à variável **num** e a expressão **num=10** iria retornar 10, fazendo com que o nosso valor de **num** fosse modificado e fazendo com que a declaração fosse executada sempre. Este problema gera erros frequentes entre iniciantes e, portanto, muita atenção deve ser tomada.

Os operadores de comparação são: **==** (igual), **!=** (diferente de), **>** (maior que), **<** (menor que), **>=** (maior ou igual), **<=** (menor ou igual).

### - for

O loop (laço) **for** é usado para repetir um comando, ou bloco de comandos, diversas vezes, de maneira que se possa ter um bom controle sobre o loop. Sua forma geral é:

*for (inicialização;condição;incremento) declaração;*

A declaração no comando for também pode ser um bloco ( { } ) e neste caso o ; é omitido. O melhor modo de se entender o loop **for** é ver de que maneira ele funciona "por dentro". O loop **for** é equivalente a se fazer o seguinte:

```
inicialização;
if (condição)
{
    declaração;
    incremento;
    "Volte para o comando if"
}
```

Podemos ver que o **for** executa a inicialização incondicionalmente e testa a condição. Se a condição for falsa ele não faz mais nada. Se a condição for verdadeira ele executa a declaração, o incremento e volta a testar a condição. Ele fica repetindo estas operações até que a condição seja falsa. Abaixo vemos um programa que coloca os primeiros 100 números na tela:

```
#include <stdio.h>
void main ()
{
    int count;
    for (count=1;count<=100;count++)
    {
        printf ("%d ",count);
    }
}
```

### Comentários

Como já foi dito, o uso de comentários torna o código do programa mais fácil de se entender. Os comentários do C devem começar com **/\*** e terminar com **\*/**. O C padrão não permite comentários aninhados (um dentro do outro), mas alguns compiladores os aceitam.

## Palavras Reservadas do C

Todas as linguagens de programação têm palavras reservadas. As palavras reservadas não podem ser usadas a não ser nos seus propósitos originais, isto é, não podemos declarar funções ou variáveis com os mesmos nomes. Como o C é "case sensitive" podemos declarar uma variável **For**, apesar de haver uma palavra reservada **for**, mas isto não é uma coisa recomendável de se fazer pois pode gerar confusão.

Apresentamos a seguir as palavras reservadas do ANSI C. Veremos o significado destas palavras chave à medida em que o curso for progredindo:

<i>auto</i>	<i>double</i>	<i>int</i>	<i>struct</i>
<i>break</i>	<i>else</i>	<i>long</i>	<i>switch</i>
<i>case</i>	<i>enum</i>	<i>register</i>	<i>typedef</i>
<i>char</i>	<i>extern</i>	<i>return</i>	<i>union</i>
<i>const</i>	<i>float</i>	<i>short</i>	<i>unsigned</i>
<i>continue</i>	<i>for</i>	<i>signed</i>	<i>void</i>
<i>default</i>	<i>goto</i>	<i>sizeof</i>	<i>volatile</i>
<i>do</i>	<i>if</i>	<i>static</i>	<i>while</i>

## **VARIÁVEIS, CONSTANTES, OPERADORES E EXPRESSÕES**

### Nomes de Variáveis

As variáveis no C podem ter qualquer nome se duas condições forem satisfeitas: o nome deve começar com uma letra ou sublinhado (\_) e os caracteres subsequentes devem ser letras, números ou sublinhado (\_). Há apenas mais duas restrições: o nome de uma variável não pode ser igual a uma palavra reservada, nem igual ao nome de uma função declarada pelo programador, ou pelas bibliotecas do C. Variáveis de até 32 caracteres são aceitas. Mais uma coisa: é bom sempre lembrar que o C é "case sensitive" e portanto deve-se prestar atenção às maiúsculas e minúsculas.

*Dicas quanto aos nomes de variáveis...*

- É uma prática tradicional do C, usar letras minúsculas para nomes de variáveis e maiúsculas para nomes de constantes. Isto facilita na hora da leitura do código;
- Quando se escreve código usando nomes de variáveis em português, evita-se possíveis conflitos com nomes de rotinas encontrados nas diversas bibliotecas, que são em sua maioria absoluta, palavras em inglês.

### Os Tipos do C

O C tem 5 tipos básicos: **char**, **int**, **float**, **void**, **double**. Destes não vimos ainda os dois últimos: O **double** é o ponto flutuante duplo e pode ser visto como um ponto flutuante com muito mais precisão. O **void** é o tipo vazio, ou um "tipo sem tipo".

Para cada um dos tipos de variáveis existem os modificadores de tipo. Os modificadores de tipo do C são quatro: **signed**, **unsigned**, **long** e **short**. Ao **float** não se pode aplicar nenhum e ao **double** pode-se aplicar apenas o **long**. Os quatro modificadores podem ser aplicados a inteiros. A intenção é que **short** e **long** devam prover tamanhos diferentes de inteiros onde isto for prático. Inteiros menores (**short**) ou maiores (**long**). **int** normalmente terá o tamanho natural para uma determinada máquina. Assim, numa máquina de 16 bits, **int** provavelmente terá 16 bits. Numa máquina de 32, **int** deverá ter 32 bits. Na verdade, cada compilador é livre para escolher tamanhos adequados para o seu próprio hardware, com a única restrição de que **shorts ints** e **ints** devem ocupar pelo menos 16 bits, **longs ints** pelo menos 32 bits, e **short int** não pode ser maior que **int**, que não pode ser maior que **long int**. O modificador **unsigned** serve para especificar variáveis sem sinal. Um **unsigned int** será um inteiro que assumirá apenas valores positivos. A seguir estão listados os tipos de dados permitidos e seu valores máximos e mínimos em um compilador típico para um hardware de 16 bits. Também nesta tabela está especificado o formato que deve ser utilizado para ler os tipos de dados com a função `scanf()`:

Tipo	Num de bits	Formato para leitura com scanf	Intervalo	
			Início	Fim
char	8	%c	-128	127
unsigned char	8	%c	0	255
signed char	8	%c	-128	127
int	16	%i	-32.768	32.767
unsigned int	16	%u	0	65.535
signed int	16	%i	-32.768	32.767
short int	16	%hi	-32.768	32.767
unsigned short int	16	%hu	0	65.535
signed short int	16	%hi	-32.768	32.767
long int	32	%li	-2.147.483.648	2.147.483.647
Signed long int	32	%li	-2.147.483.648	2.147.483.647
unsigned long int	32	%lu	0	4.294.967.295
float	32	%f	3,4E-38	3.4E+38
double	64	%lf	1,7E-308	1,7E+308
long double	80	%Lf	3,4E-4932	3,4E+4932

O tipo **long double** é o tipo de ponto flutuante com maior precisão. É importante observar que os intervalos de ponto flutuante, na tabela acima, estão indicados em faixa de *expoente*, mas os números podem assumir valores tanto positivos quanto negativos.

### Declaração e Inicialização de Variáveis

As variáveis no C devem ser declaradas antes de serem usadas. A forma geral da declaração de variáveis é:

*tipo\_da\_variável lista\_de\_variáveis;*

As variáveis da lista de variáveis terão todas o mesmo tipo e deverão ser separadas por vírgula. Como o tipo default do C é o **int**, quando vamos declarar variáveis

**int** com algum dos modificadores de tipo, basta colocar o nome do modificador de tipo. Assim um **long** basta para declarar um **long int**.

Por exemplo, as declarações

```
char ch, letra;
long count;
float pi;
```

declaram duas variáveis do tipo **char** (ch e letra), uma variável **long int** (count) e um **float** pi.

Há três lugares nos quais podemos declarar variáveis. O primeiro é fora de todas as funções do programa. Estas variáveis são chamadas **variáveis globais** e podem ser usadas a partir de qualquer lugar no programa. Pode-se dizer que, como elas estão fora de todas as funções, todas as funções as vêem. O segundo lugar no qual se pode declarar variáveis é **no início** de um bloco de código. Estas variáveis são chamadas **locais** e só têm validade dentro do bloco no qual são declaradas, isto é, só a função à qual ela pertence sabe da existência desta variável, dentro do bloco no qual foram declaradas. O terceiro lugar onde se pode declarar variáveis é na **lista de parâmetros** de uma função. Mais uma vez, apesar de estas variáveis receberem valores externos, estas variáveis são conhecidas apenas pela função onde são declaradas.

## Constantes

Constantes são valores que são mantidos fixos pelo compilador. Já usamos constantes neste curso. São consideradas constantes, por exemplo, os números e caracteres como 45.65 ou 'n', etc...

### - Constantes dos tipos básicos

Abaixo vemos as constantes relativas aos tipos básicos do C:

Tipo de Dado	Exemplos de Constantes
char	'b' '\n' '\0'
int	2 32000 -130
long int	100000 -467
short int	100 -30
unsigned int	50000 35678
float	0.0 23.7 -12.3e-10
double	12546354334.0 -0.0000034236556

### - Constantes hexadecimais e octais

Muitas vezes precisamos inserir constantes hexadecimais (base dezesseis) ou octais (base oito) no nosso programa. O C permite que se faça isto. As constantes hexadecimais começam com 0x. As constantes octais começam em 0.

Alguns exemplos:

Constante	Tipo
0xEF	Constante Hexadecimal (8 bits)

0x12A4	Constante Hexadecimal (16 bits)
03212	Constante Octal (12 bits)
034215432	Constante Octal (24 bits)

Nunca escreva portanto 013 achando que o C vai compilar isto como se fosse 13. Na linguagem C 013 é diferente de 13!

### - Constantes de barra invertida

O C utiliza, para nos facilitar a tarefa de programar, vários códigos chamados códigos de barra invertida. Estes são caracteres que podem ser usados como qualquer outro. Uma lista com alguns dos códigos de barra invertida é dada a seguir:

Código	Significado
\b	Retrocesso ("back")
\f	Alimentação de formulário ("form feed")
\n	Nova linha ("new line")
\t	Tabulação horizontal ("tab")
\"	Aspas
\'	Apóstrofo
\0	Nulo (0 em decimal)
\\	Barra invertida
\v	Tabulação vertical
\a	Sinal sonoro ("beep")
\N	Constante octal (N é o valor da constante)
\xN	Constante hexadecimal (N é o valor da constante)

### Operadores Aritméticos e de Atribuição

Os operadores aritméticos são usados para desenvolver operações matemáticas. A seguir apresentamos a lista dos operadores aritméticos do C:

Operador	Ação
+	Soma (inteira e ponto flutuante)
-	Subtração ou Troca de sinal (inteira e ponto flutuante)
*	Multiplicação (inteira e ponto flutuante)
/	Divisão (inteira e ponto flutuante)
%	Resto de divisão (de inteiros)
++	Incremento (inteiro e ponto flutuante)
--	Decremento (inteiro e ponto flutuante)

O C possui operadores unários e binários. Os unários agem sobre uma variável apenas, modificando ou não o seu valor, e retornam o valor final da variável. Os binários usam duas variáveis e retornam um terceiro valor, sem alterar as variáveis originais. A soma é um operador binário pois pega duas variáveis, soma seus valores, sem alterar as variáveis, e retorna esta soma. Outros operadores binários são os operadores - (subtração), \*, / e %. O operador - como troca de sinal é um operador unário que não altera a variável sobre a qual é aplicado, pois ele retorna o valor da variável multiplicado por -1.

O operador / (divisão) quando aplicado a variáveis inteiras, nos fornece o resultado da divisão inteira; quando aplicado a variáveis em ponto flutuante nos fornece o resultado da divisão "real". O operador % fornece o resto da divisão de dois inteiros.

Assim seja o seguinte trecho de código:

```
int a = 17, b = 3;
int x, y;
float z = 17. , z1, z2;
x = a / b;
y = a % b;
z1 = z / b;
z2 = a/b;
```

ao final da execução destas linhas, os valores calculados seriam  $x = 5$ ,  $y = 2$ ,  $z1 = 5.666666$  e  $z2 = 5.0$ . Note que, na linha correspondente a  $z2$ , primeiramente é feita uma divisão inteira (pois os dois operandos são inteiros). Somente após efetuada a divisão é que o resultado é atribuído a uma variável float.

Os operadores de incremento e decremento são unários que alteram a variável sobre a qual estão aplicados. O que eles fazem é incrementar ou decrementar, a variável sobre a qual estão aplicados, de 1. Então

```
x++;
x--;
```

são equivalentes a

```
x=x+1;
x=x-1;
```

O operador de atribuição do C é o =. O que ele faz é pegar o valor à direita e atribuir à variável da esquerda. Além disto ele retorna o valor que ele atribuiu. Isto faz com que as seguintes expressões sejam válidas:

```
x=y=z=1.5;          /* Expressao 1 */
if (k=w) ...        /* Expressão 2 */
```

A expressão 1 é válida, pois quando fazemos **z=1.5** ela retorna 1.5, que é passado adiante, fazendo  $y = 1.5$  e posteriormente  $x = 1.5$ . A expressão 2 será verdadeira se **w** for diferente de zero, pois este será o valor retornado por **k=w**. Pense bem antes de usar a expressão dois, pois ela pode gerar erros de interpretação. Você *não* está comparando **k** e **w**. Você está atribuindo o valor de **w** a **k** e usando este valor para tomar a decisão.

## Operadores Relacionais e Lógicos

Os operadores relacionais do C realizam **comparações** entre variáveis.

São eles:

Operador	Ação
>	Maior do que
>=	Maior ou igual a
<	Menor do que
<=	Menor ou igual a
==	Igual a
!=	Diferente de

Os operadores relacionais retornam verdadeiro (1) ou falso (0). Para verificar o funcionamento dos operadores relacionais, execute o programa abaixo:

```

/* Este programa ilustra o funcionamento dos operadores
relacionais. */
#include <stdio.h>
void main()
{
    int i, j;
    printf("\nEntre com dois números inteiros: ");
    scanf("%d%d", &i, &j);
    printf("\n%d == %d é %d\n", i, j, i==j);
    printf("\n%d != %d é %d\n", i, j, i!=j);
    printf("\n%d <= %d é %d\n", i, j, i<=j);
    printf("\n%d >= %d é %d\n", i, j, i>=j);
    printf("\n%d < %d é %d\n", i, j, i<j);
    printf("\n%d > %d é %d\n", i, j, i>j);
}

```

Você pode notar que o resultado dos operadores relacionais é sempre igual a 0 (falso) ou 1 (verdadeiro).

Para fazer **operações com valores lógicos** (verdadeiro e falso) temos **os operadores lógicos**:

Operador	Ação
&&	AND (E)
	OR (OU)
!	NOT (NÃO)

O programa a seguir ilustra o funcionamento dos operadores lógicos. Compile-o e faça testes com vários valores para i e j:

```
#include <stdio.h>
void main()
{
    int i, j;
    printf("informe dois números (cada um sendo 0 ou 1): ");
    scanf("%d%d", &i, &j);
    printf("%d AND %d é %d\n", i, j, i && j);
    printf("%d OR %d é %d\n", i, j, i || j);
    printf("NOT %d é %d\n", i, !i);
}
```

Mais um exemplo. O programa abaixo, imprime na tela somente os números pares entre 1 e 100, apesar da variação de *i* ocorrer de 1 em 1:

```
/* Imprime os números pares entre 1 e 100. */
#include <stdio.h>
void main()
{
    int i;
    for(i=1; i<=100; i++)
        if(!(i%2)) printf("%d ", i);    /* o operador de resto
dará falso (zero) */
}                                        /* quando usada c/ número
par. Esse resultado*/
                                        /* é invertido pelo ! */
```

## Expressões

Expressões são combinações de variáveis, constantes e operadores. Quando montamos expressões temos que levar em consideração a ordem com que os operadores são executados, conforme a tabela de precedências da linguagem C.

Exemplos de expressões:

```
Anos=Dias/365.25;
i = i+3;
c= a*b + d/e;
c= a*(b+d)/e;
```

### **- Conversão de tipos em expressões**

Quando o C avalia expressões onde temos variáveis de tipos diferentes o compilador verifica se as conversões são possíveis. Se não são, ele não compilará o programa, dando uma mensagem de erro. Se as conversões forem possíveis ele as faz, seguindo as regras abaixo:

1. Todos os **chars** e **short ints** são convertidos para **ints**. Todos os **floats** são convertidos para **doubles**.
2. Para pares de operandos de tipos diferentes: se um deles é **long double** o outro é convertido para **long double**; se um deles é **double** o outro é convertido para

**double**; se um é **long** o outro é convertido para **long**; se um é **unsigned** o outro é convertido para **unsigned**.

**- Expressões que Podem ser Abreviadas**

O C admite as seguintes equivalências, que podem ser usadas para simplificar expressões ou para facilitar o entendimento de um programa:

Expressão Original	Expressão Equivalente
x=x+k;	x+=k;
x=x-k;	x-=k;
x=x*k;	x*=k;
x=x/k;	x/=k;
x=x>>k;	x>>=k;
x=x<<k;	x<<=k;
x=x&k;	x&=k;
etc...	

**- Encadeando expressões: o operador ,**

O operador , determina uma lista de expressões que devem ser executadas seqüencialmente. Em síntese, a vírgula diz ao compilador: execute as duas expressões separadas pela vírgula, em seqüência. O valor retornado por uma expressão com o operador , é sempre dado pela expressão mais à direita. No exemplo abaixo:

```
x = (y=2, y+3) ;
```

o valor 2 vai ser atribuído a **y**, se somará 3 a **y** e o retorno (5) será atribuído à variável **x**. Pode-se encadear quantos operadores , forem necessários.

O exemplo a seguir mostra um outro uso para o operador , dentro de um for:

```
#include<stdio.h>
void main()
{
    int x, y;
    for(x=0 , y=0 ; x+y < 100 ; ++x , y++) /* Duas variáveis de
controle: x e y . Foi atribuído o valor zero a cada uma delas na
inicialização do for e ambas são incrementadas na parte de
incremento do for */
        printf("\n%d ", x+y); /* o programa imprimirá os números
pares de 2 a 98 */
}
```

**- Tabela de Precedências do C**

Esta é a tabela de precedência dos operadores em C. Alguns (poucos) operadores ainda não foram estudados, e serão apresentados em aulas posteriores.

<b>Maior precedência</b>	( ) [ ] ->
	! ~ ++ -- . -(unário)
	(cast) *(unário) &(unário)
	sizeof

```

* / %
+ -
<< >>
<<= >>=
== !=
&
^
|
&&
||
?
= += -= *= /=
'

```

**Menor  
precedência**

## ESTRUTURAS DE CONTROLE DE FLUXO

As estruturas de controle de fluxo são fundamentais para qualquer linguagem de programação. Sem elas só haveria uma maneira do programa ser executado: de cima para baixo comando por comando. Não haveria condições, repetições ou saltos. A linguagem C possui diversos comandos de controle de fluxo. É possível resolver todos os problemas sem utilizar todas elas, mas devemos nos lembrar que a elegância e facilidade de entendimento de um programa dependem do uso correto das estruturas no local certo.

### O Comando if

Já introduzimos o comando **if**. Sua forma geral é:

*if (condição) declaração;*

A expressão, na condição, será avaliada. Se ela for zero, a declaração não será executada. Se a condição for diferente de zero a declaração será executada. Aqui rerepresentamos o exemplo de um uso do comando **if** :

```

#include <stdio.h>
void main ()
{
    int num;
    printf ("Digite um numero: ");
    scanf ("%d",&num);
    if (num>10)
        printf ("\n\nO numero e maior que 10");
    if (num==10)
    {
        printf ("\n\nVoce acertou!\n");
        printf ("O numero e igual a 10.");
    }
}

```

```
    if (num<10)
        printf ("\n\nO numero e menor que 10");
}
```

### - O else

Podemos pensar no comando **else** como sendo um complemento do comando **if**. O comando **if** completo tem a seguinte forma geral:

*if (condição) declaração\_1;*  
*else declaração\_2;*

A expressão da condição será avaliada. Se ela for diferente de zero a declaração 1 será executada. Se for zero a declaração 2 será executada. É importante nunca esquecer que, quando usamos a estrutura **if-else**, estamos garantindo que uma das duas declarações será executada. Nunca serão executadas as duas ou nenhuma delas. Exemplo:

```
#include <stdio.h>
void main ()
{
    int num;
    printf ("Digite um numero: ");
    scanf ("%d",&num);
    if (num==10)
    {
        printf ("\n\nVoce acertou!\n");
        printf ("O numero e igual a 10.\n");
    }
    else
    {
        printf ("\n\nVoce errou!\n");
        printf ("O numero e diferente de 10.\n");
    }
}
```

### - O if-else-if

A estrutura **if-else-if** é apenas uma extensão da estrutura **if-else**. Sua forma geral pode ser escrita como sendo:

*if (condição\_1) declaração\_1;*  
*else if (condição\_2) declaração\_2;*  
*else if (condição\_3) declaração\_3;*  
*...*  
*...*  
*else if (condição\_n) declaração\_n;*  
*else declaração\_default;*

A estrutura acima funciona da seguinte maneira: o programa começa a testar as condições começando pela 1 e continua a testar até que ele ache uma expressão cujo resultado dê diferente de zero. Neste caso ele executa a declaração correspondente. Só

uma declaração será executada, ou seja, só será executada a declaração equivalente à *primeira* condição que der diferente de zero. A última declaração (default) é a que será executada no caso de todas as condições darem zero e é opcional. Um exemplo da estrutura acima:

```
#include <stdio.h>
void main ()
{
    int num;
    printf ("Digite um numero: ");
    scanf ("%d",&num);
    if (num>10)
        printf ("\n\nO numero e maior que 10");
    else if (num==10)
    {
        printf ("\n\nVoce acertou!\n");
        printf ("O numero e igual a 10.");
    }
    else if (num<10)
        printf ("\n\nO numero e menor que 10");
    return(0);
}
```

### - ifs aninhados

O **if** aninhado é simplesmente um **if** dentro da declaração de um outro **if** externo. O único cuidado que devemos ter é o de saber exatamente a qual **if** um determinado **else** está ligado.

Vejamos um exemplo:

```
#include <stdio.h>
void main ()
{
    int num;
    printf ("Digite um numero: ");
    scanf ("%d",&num);
    if (num==10)
    {
        printf ("\n\nVoce acertou!\n");
        printf ("O numero e igual a 10.\n");
    }
    else
    {
        if (num>10)
        {
            printf ("O numero e maior que 10.");
        }
        else
        {
            printf ("O numero e menor que 10.");
        }
    }
}
```

## - O Operador ?

Uma expressão como:

```
if (a>0)
    b=-150;
else
    b=150;
```

pode ser simplificada usando-se o operador ? da seguinte maneira:

```
b=a>0?-150:150;
```

De uma maneira geral expressões do tipo:

```
if (condição)
    expressão_1;
else
    expressão_2;
```

podem ser substituídas por:

```
condição?expressão_1:expressão_2;
```

O operador ? é limitado (não atende a uma gama muito grande de casos) mas pode ser usado para simplificar expressões complicadas. Uma aplicação interessante é a do contador circular.

## O Comando switch

O comando **if-else** e o comando **switch** são os dois comandos de tomada de decisão. Sem dúvida alguma o mais importante dos dois é o **if**, mas o comando **switch** tem aplicações valiosas. Mais uma vez vale lembrar que devemos usar o comando certo no local certo. Isto assegura um código limpo e de fácil entendimento. O comando **switch** é próprio para se testar uma variável em relação a diversos valores pré-estabelecidos. Sua forma geral é:

```
switch (variável)
{
    case constante_1:
        declaração_1;
        break;
    case constante_2:
        declaração_2;
        break;
    .
    .
    .
    case constante_n:
        declaração_n;
        break;
    default
        declaração_default;
}
```

Podemos fazer uma analogia entre o **switch** e a estrutura **if-else-if** apresentada anteriormente. A diferença fundamental é que a estrutura **switch** não aceita expressões. Aceita apenas constantes. O **switch** testa a variável e executa a declaração cujo **case** corresponda ao valor atual da variável. A declaração **default** é opcional e será executada apenas se a variável, que está sendo testada, não for igual a nenhuma das constantes.

O comando **break**, faz com que o **switch** seja interrompido assim que uma das declarações seja executada. Mas ele não é essencial ao comando **switch**. Se após a execução da declaração não houver um **break**, o programa continuará executando. Isto pode ser útil em algumas situações, mas eu recomendo cuidado. Veremos agora um exemplo do comando **switch**:

```
#include <stdio.h>
void main ()
{
    int num;
    printf ("Digite um numero: ");
    scanf ("%d", &num);
    switch (num)
    {
        case 9:
            printf ("\n\nO numero e igual a 9.\n");
            break;
        case 10:
            printf ("\n\nO numero e igual a 10.\n");
            break;
        case 11:
            printf ("\n\nO numero e igual a 11.\n");
            break;
        default:
            printf ("\n\nO numero nao e nem 9 nem 10 nem
11.\n");
    }
}
```

## O Comando for

**for** é a primeira de uma série de três estruturas para se trabalhar com loops de repetição. As outras são **while** e **do**. As três compõem a segunda família de comandos de controle de fluxo. Podemos pensar nesta família como sendo a das estruturas de repetição controlada.

Como já foi dito, o loop **for** é usado para repetir um comando, ou bloco de comandos, diversas vezes, de maneira que se possa ter um bom controle sobre o loop. Sua forma geral é:

*for (inicialização;condição;incremento) declaração;*

Abaixo vemos um programa que coloca os primeiros 100 números inteiros na tela:

```
#include <stdio.h>
void main ()
{
    int count;
    for (count=1; count<=100; count++)
    {
        printf ("%d ",count);
    }
}
```

O **for** na linguagem C é bastante flexível. Temos acesso à inicialização, à condição e ao incremento. Qualquer uma destas partes do for pode ser uma expressão qualquer do C, desde que ela seja válida. Isto nos permite fazer o que quisermos com o comando. As três formas do for abaixo são válidas:

```
for ( count = 1; count < 100 ; count++) { ... }
for (count = 1; count < NUMERO_DE_ELEMENTOS ; count++) { ... }
for (count = 1; count < BusqueNumeroDeElementos() ; count+=2) { ... }
etc ...
```

Preste atenção ao último exemplo: o incremento está sendo feito de dois em dois. Além disto, no teste está sendo utilizada uma função (BusqueNumeroDeElementos() ) que retorna um valor que está sendo comparado com count.

### - O loop infinito

O loop infinito tem a forma

*for (inicialização; ;incremento) declaração;*

Este loop chama-se loop infinito porque será executado para sempre (não existindo a condição, ela será sempre considerada verdadeira), a não ser que ele seja interrompido. Para interromper um loop como este usamos o comando **break**. O comando **break** vai quebrar o loop infinito e o programa continuará sua execução normalmente.

### O Comando while

O comando **while** tem a seguinte forma geral:

*while (condição) declaração;*

Podemos ver que a estrutura **while** testa uma condição. Se esta for verdadeira a declaração é executada e faz-se o teste novamente, e assim por diante. Assim como no caso do **for**, podemos fazer um loop infinito. Para tanto basta colocar uma expressão eternamente verdadeira na condição. Pode-se também omitir a declaração e fazer um loop sem conteúdo. Vamos ver um exemplo do uso do **while**. O programa abaixo é executado enquanto i for menor que 100. Veja que ele seria implementado mais naturalmente com um for ...

```
#include <stdio.h>
void main ()
{
    int i = 0;
    while ( i < 100)
    {
        printf(" %d", i);
        i++;
    }
}
```

## O Comando do-while

A terceira estrutura de repetição que veremos é o **do-while** de forma geral:

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

Mesmo que a declaração seja apenas um comando é uma boa prática deixar as chaves. O ponto-e-vírgula final é obrigatório. Vamos, como anteriormente, ver o funcionamento da estrutura **do-while** "por dentro":

```
        declaração;
    if (condição) "Volta para a declaração"
```

Vemos pela análise do bloco acima que a estrutura **do-while** executa a declaração, testa a condição e, se esta for verdadeira, volta para a declaração. A grande novidade no comando **do-while** é que ele, ao contrário do **for** e do **while**, garante que a declaração será executada pelo menos uma vez.

Um dos usos da estrutura **do-while** é em menus, nos quais você quer garantir que o valor digitado pelo usuário seja válido, conforme apresentado abaixo:

```
#include <stdio.h>
void main ()
{
    int i;
    do
    {
        printf ("\n\nEscolha a fruta pelo numero:\n\n");
        printf ("\t(1)...Mamao\n");
        printf ("\t(2)...Abacaxi\n");
        printf ("\t(3)...Laranja\n\n");
        scanf("%d", &i);
    } while ((i<1)|| (i>3));

    switch (i)
    {
        case 1:
            printf ("\t\tVoce escolheu Mamao.\n");
            break;
        case 2:
```

```
        printf ("\t\tVoce escolheu Abacaxi.\n");
break;
case 3:
    printf ("\t\tVoce escolheu Laranja.\n");
break;
}
}
```

## VETORES E MATRIZES

### Vetores

Vetores nada mais são que matrizes unidimensionais. Vetores são uma estrutura de dados muito utilizada. É importante notar que vetores, matrizes bidimensionais e matrizes de qualquer dimensão são caracterizadas por terem todos os elementos pertencentes ao mesmo tipo de dado. Para se declarar um vetor podemos utilizar a seguinte forma geral:

*tipo\_da\_variável nome\_da\_variável [tamanho];*

Quando o C vê uma declaração como esta ele reserva um espaço na memória suficientemente grande para armazenar o número de células especificadas em tamanho. Por exemplo, se declararmos:

```
float exemplo [20];
```

o C irá reservar  $4 \times 20 = 80$  bytes. Estes bytes são reservados de maneira contígua.

Na linguagem C a numeração começa sempre em zero. Isto significa que, no exemplo acima, os dados serão indexados de 0 a 19. Para acessá-los vamos escrever:

```
exemplo[0]
exemplo[1]
.
.
.
exemplo[19]
```

Mas ninguém o impede de escrever:

```
exemplo[30]
exemplo[103]
```

Por quê? Porque o C não verifica se o índice que você usou está dentro dos limites válidos. Este é um cuidado que *você* deve tomar. Se o programador não tiver atenção com os limites de validade para os índices ele corre o risco de ter variáveis sobreescritas ou de ver o computador travar. Bugs terríveis podem surgir. Vamos ver agora um exemplo de utilização de vetores:

```
#include <stdio.h>
void main ()
{
    int num[100]; /* Declara um vetor de inteiros de 100 posicoes */
    int count=0;
    int totalnums;
    do
    {
```

```
        printf ("\nEntre com um numero (-999 p/ terminar): ");
        scanf ("%d",&num[count]);
        count++;
    } while (num[count-1]!=-999);
totalnums=count-1;
printf ("\n\n\n\t Os números que você digitou foram:\n\n");
for (count=0;count<totalnums;count++)
    printf (" %d",num[count]);
}
```

No exemplo acima, o inteiro *count* é inicializado em 0. O programa pede pela entrada de números até que o usuário entre com o Flag -999. Os números são armazenados no vetor **num**. A cada número armazenado, o contador do vetor é incrementado para na próxima iteração escrever na próxima posição do vetor. Quando o usuário digita o flag, o programa abandona o primeiro loop e armazena o total de números gravados. Por fim, todos os números são impressos. É bom lembrar aqui que nenhuma restrição é feita quanto a quantidade de números digitados. Se o usuário digitar mais de 100 números, o programa tentará ler normalmente, mas o programa os escreverá em uma parte não alocada de memória, pois o espaço alocado foi para somente 100 inteiros. Isto pode resultar nos mais variados erros no instante da execução do programa.

## Matrizes

### - Matrizes bidimensionais

Já vimos como declarar matrizes unidimensionais (vetores). Vamos tratar agora de matrizes bidimensionais. A forma geral da declaração de uma matriz bidimensional é muito parecida com a declaração de um vetor:

*tipo\_da\_variável nome\_da\_variável [altura][largura];*

É muito importante ressaltar que, nesta estrutura, o índice da esquerda indexa as linhas e o da direita indexa as colunas. Quando vamos preencher ou ler uma matriz no C o índice mais à direita varia mais rapidamente que o índice à esquerda. Mais uma vez é bom lembrar que, na linguagem C, os índices variam de zero ao valor declarado, menos um; mas o C não vai verificar isto para o usuário. Manter os índices na faixa permitida é tarefa do programador. Abaixo damos um exemplo do uso de uma matriz:

```
#include <stdio.h>
void main ()
{
int mtrx [20][10];
int i, j, count;
count=1;
for (i=0;i<20;i++)
    for (j=0;j<10;j++)
    {
        mtrx[i][j]=count;
        count++;
    }
}
```

No exemplo acima, a matriz **mtx** é preenchida, sequencialmente por linhas, com os números de 1 a 200. Você deve entender o funcionamento do programa acima antes de prosseguir.

### - Matrizes multidimensionais

O uso de matrizes multidimensionais na linguagem C é simples. Sua forma geral é:

```
tipo_da_variável nome_da_variável [tam1][tam2] ... [tamN];
```

Uma matriz N-dimensional funciona basicamente como outros tipos de matrizes. Basta lembrar que o índice que varia mais rapidamente é o índice mais à direita.

### - Inicialização

Podemos inicializar matrizes, assim como podemos inicializar variáveis. A forma geral de uma matriz como inicialização é:

```
tipo_da_variável nome_da_variável [tam1][tam2] ... [tamN] = {lista_de_valores};
```

A lista de valores é composta por valores (do mesmo tipo da variável) separados por vírgula. Os valores devem ser dados na ordem em que serão colocados na matriz. Abaixo vemos alguns exemplos de inicializações de matrizes:

```
float vect [6] = { 1.3, 4.5, 2.7, 4.1, 0.0, 100.1 };
int matr [3][4] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
char str [10] = { 'J', 'o', 'a', 'o', '\0' };
char str [10] = "Joao";
char str_vect [3][10] = { "Joao", "Maria", "Jose" };
```

O primeiro demonstra inicialização de vetores. O segundo exemplo demonstra a inicialização de matrizes multidimensionais, onde **matr** está sendo inicializada com 1, 2, 3 e 4 em sua primeira linha, 5, 6, 7 e 8 na segunda linha e 9, 10, 11 e 12 na última linha. No terceiro exemplo vemos como inicializar uma string e, no quarto exemplo, um modo mais compacto de inicializar uma string. O quinto exemplo combina as duas técnicas para inicializar um vetor de strings. Repare que devemos incluir o ; no final da inicialização.

### - Inicialização sem especificação de tamanho

Podemos, em alguns casos, inicializar matrizes das quais não sabemos o tamanho *a priori*. O compilador C vai, neste caso verificar o tamanho do que você declarou e considerar como sendo o tamanho da matriz. Isto ocorre na hora da compilação e não poderá mais ser mudado durante o programa, sendo muito útil, por exemplo, quando vamos inicializar uma string e não queremos contar quantos caracteres serão necessários. Alguns exemplos:

```
char mess [] = "Linguagem C: flexibilidade e poder.";
int matr [][] = { 1, 2, 2, 4, 3, 6, 4, 8, 5, 10 };
```

No primeiro exemplo, a string **mess** terá tamanho 36. Repare que o artifício para realizar a inicialização sem especificação de tamanho é não especificar o tamanho! No segundo exemplo o valor não especificado será 5.

## Entrada e Saída Formatada

As funções que resumem todas as funções de entrada e saída formatada no C são as funções **printf()** e **scanf()**. Um domínio destas funções é fundamental ao programador.

### - printf

Protótipo:

```
int printf (char *str,...);
```

As reticências no protótipo da função indicam que esta função tem um número de argumentos variável. Este número está diretamente relacionado com a string de controle **str**, que deve ser fornecida como primeiro argumento. A string de controle tem dois componentes. O primeiro são caracteres a serem impressos na tela. O segundo são os comandos de formato. Como já vimos, os últimos determinam uma exibição de variáveis na saída. Os comandos de formato são precedidos de %. A cada comando de formato deve corresponder um argumento na função **printf()**. Se isto não ocorrer podem acontecer erros imprevisíveis no programa.

Abaixo apresentamos a tabela de códigos de formato:

Código	Formato
%c	Um caracter (char)
%d	Um número inteiro decimal (int)
%i	O mesmo que %d
%e	Número em notação científica com o "e"minúsculo
%E	Número em notação científica com o "e"maiúsculo
%f	Ponto flutuante decimal
%g	Escolhe automaticamente o melhor entre %f e %e
%G	Escolhe automaticamente o melhor entre %f e %E
%o	Número octal
%s	String
%u	Decimal "unsigned" (sem sinal)
%x	Hexadecimal com letras minúsculas
%X	Hexadecimal com letras maiúsculas
%%	Imprime um %
%p	Ponteiro

Vamos ver alguns exemplos:

Código	Imprime
printf ("Um %%%c %s",'c',"char");	Um %c char
printf ("%X %f %e",107,49.67,49.67);	6B 49.67 4.967e1
printf ("%d %o",10,10);	10 12

É possível também indicar o tamanho do campo, justificação e o número de casas decimais. Para isto usa-se códigos colocados entre o % e a letra que indica o tipo de formato.

Um inteiro indica o tamanho mínimo, em caracteres, que deve ser reservado para a saída. Se colocarmos então **%5d** estamos indicando que o campo terá cinco caracteres de comprimento *no mínimo*. Se o inteiro precisar de mais de cinco caracteres para ser exibido então o campo terá o comprimento necessário para exibi-lo. Se o comprimento do inteiro for menor que cinco então o campo terá cinco de comprimento e será preenchido com espaços em branco. Se se quiser um preenchimento com zeros pode-se colocar um zero antes do número. Temos então que **%05d** reservará cinco casas para o número e se este for menor então se fará o preenchimento com zeros.

O alinhamento padrão é à direita. Para se alinhar um número à esquerda usa-se um sinal - antes do número de casas. Então **%-5d** será o nosso inteiro com o número mínimo de cinco casas, só que justificado a esquerda.

Pode-se indicar o número de casas decimais de um número de ponto flutuante. Por exemplo, a notação **%10.4f** indica um ponto flutuante de comprimento total dez e com 4 casas decimais. Entretanto, esta mesma notação, quando aplicada a tipos como inteiros e strings indica o número mínimo e máximo de casas. Então **%5.8d** é um inteiro com comprimento mínimo de cinco e máximo de oito.

Vamos ver alguns exemplos:

Código	Imprime
<code>printf ("%5.2f",456.671);</code>	456.67
<code>printf ("%5.2f",2.671);</code>	2.67
<code>printf ("%10s","Ola");</code>	Ola

Nos exemplos o "pipe" ( | ) indica o início e o fim do campo mas não são escritos na tela.

### - scanf

Protótipo:

```
int scanf (char *str,...);
```

A string de controle str determina, assim como com a função **printf()**, quantos parâmetros a função vai necessitar. Devemos sempre nos lembrar que a função **scanf()** deve receber ponteiros como parâmetros. Isto significa que as variáveis que não sejam por natureza ponteiros devem ser passadas precedidas do operador **&**. Os especificadores de formato de entrada são muito parecidos com os de **printf()**. Os caracteres de conversão *d*, *i*, *u* e *x* podem ser precedidos por *h* para indicarem que um apontador para *short* ao invés de *int* aparece na lista de argumento, ou pela letra *l* (letra ele) para indicar que que um apontador para *long* aparece na lista de argumento. Semelhantemente, os caracteres de conversão *e*, *f* e *g* podem ser precedidos por *l* para indicarem que um apontador para *double* ao invés de *float* está na lista de argumento. Exemplos:

Código	Formato
<code>%c</code>	Um único caracter (char)

%d	Um número decimal (int)
%i	Um número inteiro
%hi	Um short int
%li	Um long int
%e	Um ponto flutuante
%f	Um ponto flutuante
%lf	Um double
%h	Inteiro curto
%o	Número octal
%s	String
%x	Número hexadecimal
%p	Ponteiro

### Comandos

**ABS** – Pega o valor absoluto de um número decimal <Math.h>.

**ATOI** – Converte uma string para integer (inteiro) <Stdlib.h>.

**CALLOC** – Aloca parte da memória principal (usado com arquivo) <Stdlib.h>.

**CEIL** – Arredonda o número para cima <Math.h>.

**CLRSCR** – Limpa a tela <Stdio.h>.

**DELAY** – Espera um tempo determinado <Stdio.h>.

**EOF** – Verifica se é fim de arquivo <Io.h>.

**EXIT** – Termina o programa <Stdlib.h>.

**FCLOSE** – Fecha um arquivo <Stdio.h>.

**FLOOR** – Arredonda o número para baixo <Math.h>.

**FOPEN** – Abre um arquivo <Stdio.h>.

**FREE** – Libera um espaço de memória <Stdlib.h>.

**FSCANF** – Guarda dados formatados em um arquivo <Stdio.h>.

**FSEEK** – Posiciona o ponteiro em algum lugar do seu arquivo (registro) <Stdio.h>.

**GETCHAR** – Lê um caracter (dado) formatado <Stdio.h>

**GETCHE** – Lê um caracter e mostra ele na tela <Conio.h>.

**GETCH** – Lê um caracter e não mostra ele na tela <Conio.h>.

**GETS** – Lê uma string <Stdio.h>.

**GOTOXY** – Posiciona o cursor na tela <Conio.h>.

**ISALPHA** – Verifica e retorna verdadeiro se o caracter é um símbolo alfanumérico <Ctype.h>.

**ISASCII** – Verifica se o caracter pertence a tabela Ascii <Ctype.h>.

**ISDIGIT** – Verifica e retorna verdadeiro se o caracter for um dígito entre zero e nove <Ctype.h>.

**ISLOWER** – Verifica se o caracter é minúsculo <Ctype.h>.

**ISUPPER** – Verifica e retorna verdadeiro se o caracter é maiúsculo <Ctype.h>.

**ISSPACE** – Verifica e retorna falso se o caracter for em branco, return, tab ao final de uma linha <Ctype.h>.

**ITOA** – Converte um número decimal inteiro em string <Math.h>.

**KBHIT** – Verifica se foi pressionada alguma tecla <Conio.h>.

**KEEP** – Sai para o DOS mas mantém a volta na memória (DOS SHELL) <Dos.h>.

**MALLOC** – Aloca a memória principal (usado com arquivo) <Alloc.h>.

**MIN** – Compara dois valores para verificar qual é o menor <Stdio.h>, <Stdlib.h>.

**OPEN** – Abre um arquivo <Io.h>.

**OUTTEXT** – Escreve na tela no modo gráfico <Graphics>.

**POW** – Retorna  $x$  elevado a potência  $y$  <Math.h>.

**PRINTF** – Permite a saída de dados via monitor <Stdio.h>.

**PUTCH** – Escreve um caracter na tela <Stdio.h>.

**PUTC** – Escreve um caracter Ascii na tela <Stdio.h>.

**PUTCHAR** – Escreve um caracter (dado) formatado <Stdio.h>.

**PUTS** – Exibe o conteúdo de uma variável e uma string entre aspas (=printf) <Stdio.h>.

**RAND** – Gerador de números randômicos (número aleatório) <Stdlib.h>.

**RANDOMIZE** – Inicializa o gerador de números randômicos (aleatórios) <Stdlib.h>.

**REALLOC** – Aloca a memória principal <Alloc.h>.

**SCANF** – Permite a entrada de dados via teclado <Stdio.h>.

**SETCOLOR** – Ajusta a cor do desenho. Muda a cor do texto na unidade gráfica <Graphics>.

**SLEEP** – Suspende a execução de um programa por um tempo determinado <Dos.h>.

**SOUND** – Emite som com uma frequência especificada <Dos.h>.

**STRCAT** – Concatena (junta) duas strings <String.h>.

**STRCHR** – Posiciona o ponteiro na primeira posição da string <String.h>.

**STRCMP** – Compara duas strings <String.h>.

**STRCOPY** – Copia uma string para uma variável <String.h>.

**STRLEN** – Retorna o comprimento da string <String.h>.

**SYSTEM** – Executa um comando do MS-DOS <Stdlib.h>.

**TEEL** – Pega a posição atual do ponteiro de arquivo <Io.h>.

**TEXTBACKGROUND** – Seleciona uma nova cor de fundo para o texto <Graphics>.

**TEXTCOLOR** – Seleciona uma nova cor para os caracteres no modo texto <Conio.h>.

**TEXTHEIGHT** – Retorna a altura de uma string em pixels <Graphics>.

**TEXTMODE** – Coloca a tela no modo texto <Conio.h>.

**TIME** – Pega do sistema a hora do dia <Time.h>.

**TMPFILE** – Abre um arquivo binário “temporário” <Stdio.h>.

**TOASCII** – Traduz caracteres para o formato Ascii <Ctype.h>.

**TOLOWER** – Converte os caracteres em minúsculo <Ctype.h>.

**TOUPPER** – Converte os caracteres em maiúsculo <Ctype.h>.

**WHEREX** – Fornece a posição horizontal do cursor dentro da janela <Conio.h>.

**WHEREY** – Fornece a posição vertical do cursor dentro da janela <Conio.h>.

**WINDOW** – Define a janela ativa no modo texto <Conio.h>.

**WRITE** – Escreve num arquivo <Io.h>.

## EXERCÍCIOS DE FIXAÇÃO

### Comando if...else

- 1) Faça um programa em “C” que leia dois números e exiba qual é o maior.
- 2) Faça um programa em “C” que leia a altura de duas pessoas (A e B) e verifique qual é a maior e exiba com uma mensagem qualquer qual é o maior. Ex: “A é maior que B”
- 3) Faça um programa em “C” que pergunte em que ano você nasceu e exiba quantos anos você tem. Exiba também se você é mais velho que 18 anos inclusive, se está entre 15 e 18 anos, ou se tem menos que 15 anos.
- 4) Faça um programa em “C” que permita o usuário digitar o número do mês escolhido e exiba o mês por extenso.
- 5) Faça um programa em “C” que permita a entrada de um número qualquer e exiba se este número é par ou ímpar. Se for par exiba também a raiz quadrada do mesmo; se for ímpar exiba o número elevado ao quadrado.
- 6) Faça um programa em “C” que leia os lados de um triângulo retângulo e exiba sua hipotenusa. Se a hipotenusa for maior que 100 escreva, hipotenusa muito grande, caso contrário exiba hipotenusa pequena.

### Comando for/ while/ do...while

- 7.) Faça um programa em “C” que exiba na tela os valores de 20 a 50 usando um for, um while e um do...while no mesmo programa, separados por um getch() cada um
- 8.) Faça um programa em “C” que exiba os números ímpares de 1 até 50.
- 9.) Faça um programa em “C” que exiba os números pares de 1 até um determinado número definido pelo usuário.
- 10.) Faça um programa em “C” que monte uma tabela de 1 a 20 contendo o número, o quadrado e a raiz quadrada do número.
- 11) Desenvolva um programa em “C” que permita o usuário digitar um número qualquer e exiba na tela todos os números antecedentes até chegar em 1
- 12) Apresentar todos os números divisíveis por 4 que sejam menores que 200.
- 13) Apresente ao usuário todos os números de 1 a 300 e no final exiba quantos são múltiplos de 7 e múltiplos de 11.
- 14) Apresentar o total da soma obtido dos cem primeiros números inteiros:  $1+2+3+4+\dots+98+99+100$ .

- 15) Apresente o fatorial de um número qualquer:  $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$ .
- 16) Apresentar as potências de 3, variando de 0 a 15. Deve ser considerado que qualquer número elevado a zero é 1, e elevado a 1 é ele próprio
- 17) Faça um programa em “C” que exiba a seqüência lógica abaixo(Fibonacci): (os 30 primeiros números)
- 1    1    2    3    5    8    13 ....
- 18) Faça um programa em “C” que exiba a seqüência lógica abaixo(Bergamash): (os 30 primeiros números)
- 1    1    1    3    5    9    17    31 ....
- 19) Tendo como entrada X e N, calcular  $X^n$ . Lembre-se:  $2^3 = 2 \times 2 \times 2 = 8$ . Para isto faça um programa em “C” para calcular.
- 20) Faça um programa em “C” que permita o usuário entrar com o valor de “n” resistores. Feito isto calcule a resistência equivalente caso estes resistores estiverem em série e em paralelo.
- 21) Faça um programa em “C” que calcule a somatória dos 20 primeiros números desta seqüência:
- $1 - \frac{1}{2} + \frac{1}{4} - \frac{1}{8} + \frac{1}{16} - \frac{1}{32} + \frac{1}{64} - \dots$
- 22) Faça um programa em “C” que verifique se um determinado número digitado pelo usuário é primo ou não. Número primo é aquele que é divisível somente por um e por ele mesmo.
- 23) Faça um programa em “C” que solicite as variáveis hora,minuto e segundo atuais e ainda um tempo qualquer em segundos. Após todas as informações disponíveis, calcular e mostrar o novo horário que será após a passagem dos segundos informados.
- $12:50:35 + 10 \text{ segundos} \rightarrow 12:50:45$   
 $10:20:50 + 5000 \text{ segundos} \rightarrow 11:44:10$
- 24) Considere a progressão geométrica (PG) 1,2,4,8,16,32.... e um número inteiro positivo “n”.
- a. Deseja-se exibir os “n” primeiros números
  - b. calcular e exibir a soma dos “n” primeiros termos da PG

## Vetores Unidimensionais

- 25) Faça um programa em “C” que preencha um vetor chamado A de 15 posições com o valor 1. Exiba este vetor na tela.
- 26) Faça um programa em “C” que permita a entrada de dados em 2 vetores de 5 posições cada, realize a soma e guarde em um terceiro, a multiplicação em um quarto. No final exiba todos os vetores.
- 27) Faça um programa em “C” que preencha um vetor de 10 posições e exiba no final somente os números pares deste vetor, juntamente com a posição que se encontra no vetor.
- 28) Faça um programa em “C” que preencha randomicamente um vetor de 100 posições e no final exiba quantos números são pares, quantos ímpares, quantos múltiplos de 5 e quantos múltiplos de 7.
- 29) Faça um programa em “C” que preencha um vetor randomicamente um vetor de 20 posições e guarde em um vetor chamado par os números pares e em um vetor chamado impar os números ímpares. No final exiba os vetores.

- 30) Armazenar o salário de 5 pessoas. Calcular e armazenar o novo salário em um novo vetor sabendo-se que o reajuste foi de 8%. Imprima no final, o salário anterior e o novo salário das 5 pessoas. No final exiba também a somatória dos salários.
- 31) Preencha um novo vetor de 100 posições e exiba no final o maior e o menor valor.
- 32) Entrar com números reais para dois vetores A e B de dez elementos cada. Gerar e imprimir o vetor diferença.
- 33) Permita o usuário digitar 10 valores e exiba este vetor em ordem crescente e decrescente.
- 34) Faça um programa em “C” que leia dois vetores A e B, contendo cada um 10 valores. Intercale esses dois conjuntos (a[1] b[1] a[2] b[2] ....) formando um vetor V de 20 elementos. Ordene de forma decrescente. Imprima o vetor V.
- 35) Preencher randomicamente um vetor VET do tipo inteiro com 50 posições, onde podem existir vários elementos repetidos. Gere o vetor VET1 que também será ordenado e terá somente os elementos do vetor VET que não são repetidos.
- 36) Dado um número qualquer em decimal, converta-o para binário.
- 37) Preencha um vetor de 100 posições randomicamente; verifique qual o número que mais aparece e mostre no final o número e a quantidade de vezes que aparece.

## Matrizes

- 38) Dada uma matriz de ordem “N”, calcular e exibir a soma de todos os seus elementos.
- 39) Dada uma matriz de ordem “N”, calcular e exibir a soma de todos os elementos da diagonal principal.
- 40) Dada uma matriz de ordem “N”, calcular e exibir a soma de todos os elementos da diagonal secundária.
- 41) Dada uma matriz de ordem “N”, calcular e exibir a soma de todos os elementos que estão acima da diagonal principal.
- 42) Dada uma matriz de ordem “N”, calcular e exibir a soma de todos os elementos que estão abaixo da diagonal principal.
- 43) Dada uma matriz de ordem “N”, calcular e exibir a soma de todos os elementos que estão acima da diagonal secundária.
- 44) Dada uma matriz de ordem “N”, calcular e exibir a soma de todos os elementos que estão abaixo da diagonal secundária.
- 45) Dada uma matriz A de ordem “N”, calcular e exibir uma matriz B, cujos valores são os mesmos da matriz A multiplicados por 5.
- 46) Dada uma matriz A de ordem “N”, calcular e exibir uma matriz B, cujos valores são os mesmos da matriz A multiplicados por 3.

- 47) Dada uma matriz A e uma B de ordem “N”, gere e exiba uma matriz C cujos valores são dados pela soma dos elementos de  $A + B$ .
- 48) Dada uma matriz A e uma B de ordem “N”, gere e exiba uma matriz C cujos valores são dados pela subtração dos elementos de  $A - B$ .
- 49) Dada uma matriz A de ordem “N”, calcular e exibir uma matriz B, cuja matriz B é transposta de A
- 50) Dadas duas matrizes A e B, realizar uma multiplicação de matrizes.

## Colunas

**Linhas**

<b>0,0</b>	<b>0,1</b>	<b>0,2</b>
<b>1,0</b>	<b>1,1</b>	<b>1,2</b>
<b>2,0</b>	<b>2,1</b>	<b>2,2</b>

**Boa Leitura!!!**  
**Bom Estudo!!!**

[gale@unisal.com.br](mailto:gale@unisal.com.br)  
[tizzei@unisal.com.br](mailto:tizzei@unisal.com.br)

Nome: \_\_\_\_\_ RA \_\_\_\_\_

Complete os quadrinhos de acordo com a tabela abaixo:

- 1. Converte um caracter maiúsculo para minúsculo
- 2. Seleciona modo de exibição do texto
- 3. Sai do programa e o mantém residente
- 4. Exibe um caracter na tela
- 5. Sai para o sistema
- 6. Lê um caracter, mas não exibe
- 7. Checa se alguma tecla foi pressionada
- 8. Aciona pesquisa aleatória
- 9. Verifica o tamanho da string
- 10. Verifica a posição vertical do cursor na tela
- 11. Copia uma string para outra.
- 12. Seleciona cor da letra
- 13. Seleciona cor da letra no modo gráfico
- 14. Converte uma variável caracter para inteiro
- 15. Verifica se um caracter é maiúsculo
- 16. Permite a entrada de dados formatados
- 17. Eleva um número ao outro
- 18. Libera memória
- 19. Arredonda um número para baixo
- 20. Arredonda um número para cima

