

**Apostila da disciplina de**  
**PROGRAMAÇÃO I**  
**- Linguagem Pascal -**

**Profa. Flávia Pereira de Carvalho**

**Agosto de 2006**

## Sumário

---

|   | <i>Página</i> |
|---|---------------|
| <b>1 INTRODUÇÃO .....</b>                               | <b>3</b>      |
| <b>2 PROGRAMAS E PROGRAMAÇÃO .....</b>                  | <b>3</b>      |
| <b>3 LINGUAGEM DO COMPUTADOR.....</b>                   | <b>4</b>      |
| <b>4 LINGUAGENS DE ALTO NÍVEL .....</b>                 | <b>5</b>      |
| 4.1 LINGUAGEM PASCAL .....                              | 6             |
| 4.2 TIPOS DE INSTRUÇÕES DE ALTO NÍVEL.....              | 6             |
| <b>5 CONVERSÃO DO PROGRAMA-FONTE.....</b>               | <b>8</b>      |
| <b>6 SOFTWARE QUE VOCÊ PRECISA TER OU CONHECER.....</b> | <b>9</b>      |
| <b>7 TIPOS DE PROGRAMAS .....</b>                       | <b>10</b>     |
| <b>8 ESTRUTURA DE UM PROGRAMA .....</b>                 | <b>11</b>     |
| <b>9 TIPOS DE DADOS.....</b>                            | <b>12</b>     |
| <b>10 OPERADORES ARITMÉTICOS.....</b>                   | <b>14</b>     |
| <b>11 EXPRESSÕES ARITMÉTICAS .....</b>                  | <b>14</b>     |
| 11.1 FUNÇÕES NUMÉRICAS PREDEFINIDAS .....               | 15            |
| <b>12 FUNÇÕES DE FORMATAÇÃO DA TELA.....</b>            | <b>16</b>     |
| <b>13 EXPRESSÕES LÓGICAS.....</b>                       | <b>16</b>     |
| 13.1 RELAÇÕES .....                                     | 16            |
| 13.2 OPERADORES LÓGICOS.....                            | 17            |
| 13.3 PRIORIDADES .....                                  | 19            |
| <b>14 ESTRUTURA CONDICIONAL (SELEÇÃO).....</b>          | <b>20</b>     |
| 14.1 IF-THEN-ELSE .....                                 | 20            |
| 14.2 CASE.....  | 23            |
| <b>15 ESTRUTURAS DE REPETIÇÃO .....</b>                 | <b>24</b>     |
| 15.1 WHILE-DO.....                                      | 24            |
| 15.2 REPEAT-UNTIL.....                                  | 25            |
| 15.3 FOR-TO-DO .....                                    | 26            |
| <b>16 EXPRESSÕES LITERAIS .....</b>                     | <b>28</b>     |
| <b>17 FUNÇÕES PREDEFINIDAS .....</b>                    | <b>29</b>     |
| <b>18 DEFINIÇÃO DE CONSTANTES .....</b>                 | <b>31</b>     |
| <b>19 DEFINIÇÃO DE TIPOS .....</b>                      | <b>32</b>     |
| <b>20 ESTRUTURAS DE DADOS.....</b>                      | <b>33</b>     |
| 20.1 ARRAY .....  | 33            |
| 20.1.1 Como declarar e acessar VETORES em Pascal.....   | 33            |
| 20.1.2 Como declarar e acessar MATRIZES em Pascal.....  | 34            |
| 20.2 REGISTRO (RECORD) .....                            | 37            |
| 20.2.1 Como declarar e acessar REGISTROS em Pascal..... | 37            |
| 20.2.2 Exemplos de Registros .....                      | 39            |
| <b>21 COMANDO WITH.....</b>                             | <b>42</b>     |
| <b>22 MODULARIZAÇÃO .....</b>                           | <b>43</b>     |
| 22.1 PROCEDIMENTOS (PROCEDURE) .....                    | 43            |
| 22.1.1 Passagem de parâmetros POR VALOR.....            | 44            |
| 22.2 FUNÇÕES (FUNCTION) .....                           | 46            |
| 22.2.2 Passagem de parâmetros POR REFERÊNCIA.....       | 47            |

## 1 Introdução

---

Um computador é uma máquina que, para realizar algo, precisa que alguém lhe indique o que fazer, de uma forma que ele entenda. Para que você possa fazer isso, é preciso que:

- Conheça o computador e os recursos disponíveis;
- Saiba o que quer que o computador faça;
- Instrua o computador, através de um programa escrito em uma linguagem de programação.

Nesta introdução veremos informações básicas para se aprender uma linguagem de programação de computadores.

## 2 Programas e Programação

---

O *hardware* do computador, constituído de placas e dispositivos mecânicos e eletrônicos, precisa do *software* para lhe dar vida: programas, com finalidades bem determinadas, que façam o que os usuários querem ou precisam. Há programas para editar um texto, para fazer cálculos, jogos e literalmente milhares de outras finalidades. Alguns programas maiores, como processadores de texto, planilhas eletrônicas e navegadores da Internet, são de fato agrupamentos de dezenas de programas relacionados entre si.

Programas são constituídos de instruções e comandos que o processador do computador entende do tipo: faça isso, faça aquilo. Esses comandos devem estar representados em uma linguagem. Você talvez não esteja ciente de que está familiarizado com vários tipos de linguagem. Além do Português, há linguagens para inúmeras finalidades: sinais, faixas e placas de trânsito, gestos com a mão e com a cabeça, o Braille, a linguagem dos surdos-mudos etc. Até para falar com bebês temos formas específicas! Também há formas de linguagem mais simples para nos comunicarmos com máquinas, como a televisão, o videocassete, a calculadora. Ninguém "chama" verbalmente um elevador, nem "diz" à TV qual canal sintonizar; se você não fizer algo que os aparelhos entendam, não vai conseguir o que quer. Assim é o computador: você deve comandá-lo de uma forma que ele entenda.

Para que algo aconteça no computador, não basta um programa; os comandos do programa devem ser executados. Programas de computador são como filmes: uma coisa é a película em um rolo, contendo uma seqüência de imagens. Outra coisa é colocar o filme em um projetor e assisti-lo na telona. Os programas ficam guardados em arquivos no disco rígido ou CD, DVD, enfim, até que seja comandada (acionada) sua execução. São então carregados pelo Sistema Operacional para a memória e só então acontece (exatamente) o que foi programado, e você pode perceber o que o programa faz.

Uma diferença entre máquinas em geral e o computador é que este pode fazer muito mais coisas, portanto precisa de uma variedade maior de comandos. E outra diferença fundamental: o computador pode armazenar os comandos, agrupados em programas, para execução posterior.

Programar um computador é, portanto, produzir comandos agrupados em programas, em uma linguagem que o computador entenda e que, quando executados, façam o computador produzir algum resultado desejado. Um bom programador é treinado em algumas habilidades, sendo o nosso objetivo desenvolver na prática essas habilidades, para isso usando uma linguagem de programação de computadores chamada Pascal.

### 3 Linguagem do Computador

A atividade básica de um computador consiste em executar instruções, através de um microprocessador ou simplesmente processador, às vezes também chamado de CPU (*Central Processing Unit* – Unidade Central de Processamento). O processador, em última análise, recebe instruções na forma de impulsos elétricos: em um determinado circuito, pode estar ou não fluindo corrente. Representamos cada impulso por 1 ou 0, conforme passe corrente ou não. Esta é a menor unidade de informação que pode ser representada em um computador, e é chamada bit. A CPU recebe instruções e dados na forma de bits agrupados de 8 em 8; cada conjunto de 8 bits é chamado byte. De uma forma simplificada, um byte, ou seja, um conjunto de 8 impulsos elétricos (ou sua ausência), constitui uma instrução ou um dado (na verdade, uma instrução pode ocupar mais de um byte). Representaremos essa unidade de informação pelo número correspondente no sistema decimal. Dessa forma, ao byte 00000001 associamos o número 1, ao byte 00000011 associamos o número 3, ao 00000100 o número 4 e assim por diante. Um byte pode armazenar, portanto, um número de 0 a 255 (11111111).

A memória RAM, ou simplesmente memória, de um computador é constituída de uma seqüência de milhares ou milhões de bytes, cada um identificado por um número que constitui o seu endereço (veja a Tabela 1). O processador tem a capacidade de buscar o conteúdo da memória e executar instruções ali armazenadas. A CPU também contém algumas unidades de memória, chamadas registradores, identificados por nomes como AX, CS e IP, que também armazenam números e servem a várias funções.

|    |    |   |     |     |    |     |  |  |  |  |  |  |  |  |
|----|----|---|-----|-----|----|-----|--|--|--|--|--|--|--|--|
| 1  | 2  | 3 | 4   | 5   | 6  | ... |  |  |  |  |  |  |  |  |
| 56 | 23 | 0 | 247 | 154 | 87 | ... |  |  |  |  |  |  |  |  |

**Tabela 1:** Esquema simplificado da memória RAM: endereços e respectivos valores.

Os números armazenados na memória podem representar dados ou instruções. Quando representando instruções, têm significados específicos para a CPU. Por exemplo, a CPU em geral recebe comandos do tipo (mas não na forma):

*“Armazene 9 no registrador DS”*

*“Armazene 204 no endereço de memória 1.234.244”*

*“Some 5 ao conteúdo do registrador AL”*

*“Se a última instrução deu resultado 0, passe a executar as instruções a partir do endereço de memória 457.552”*

```
PUSH DS
CLD
MOV CX,0FFFFH
XOR AL,AL
NOT CX
LES DI,Dest
REP MOVSB
MOV AX,DI
MOV DX,ES
DEC AX
POP DS
```

Este tipo de linguagem é chamada *linguagem de máquina* ou assembly (veja ao lado um trecho de programa). As instruções são executadas na seqüência em que estiverem na memória, a menos que o fluxo de execução seja redirecionado por alguma instrução apropriada. Nesta linguagem, mesmo para mostrar algo na tela é necessário colocar números em certos endereços de memória, reservados para isso. Não há indicação explícita do tipo número da linha ou da coluna da tela; devemos calcular esses dados.

#### 4 Linguagens de Alto Nível

---

Escrever programas de computador em linguagem de máquina, apesar de os programas produzidos serem extremamente rápidos, pode ser muito difícil, trabalhoso e de alto custo, além de exigir conhecimentos profundos sobre o computador.

Para não termos que programá-lo nessa linguagem difícil, foram desenvolvidas as **linguagens de alto nível**. Estas nos permitem descrever o que queremos que o computador faça utilizando instruções mais próximas da nossa linguagem. Além de facilitarem as descrições dos processos a serem executados, as linguagens de alto nível simplificam a utilização da memória do computador, diminuindo a quantidade de detalhes com os quais deve ocupar-se o programador. Assim, ao invés de lidarmos com bits, bytes, endereços de memória e uma infinidade de outros detalhes, podemos pensar em "limpar a tela", "imprimir uma linha de texto", somar e subtrair variáveis como na matemática e tomar uma decisão na forma "se...então".

Veja um exemplo específico - as instruções abaixo, na linguagem Pascal, determinam que a tela seja limpa, uma soma seja efetuada e o resultado desta mostrado na tela:

```
ClrScr;
Write (513 + 450 + 1200);
```

Uma linguagem de programação de alto nível possui várias características em comum com a nossa linguagem. Elas possuem um *alfabeto* (letras, números e outros símbolos) e *palavras*. Há palavras predefinidas na linguagem, mas também podemos formar as nossas próprias. *Frases* podem ser construídas com as palavras, respeitando-se certas regras. O conjunto de regras de construção de palavras e frases numa linguagem de alto nível, assim como nas linguagens comuns, chama-se **sintaxe**.

## 4.1 Linguagem Pascal

O PASCAL (não fale "pascoal"! ) foi desenvolvido por Niklaus Wirth (<http://www.cs.inf.ethz.ch/~wirth/>) para ser a primeira linguagem de programação a ser aprendida. É bem fácil de ensinar e aprender, e sua estrutura é um pouco mais rígida do que a de outras linguagens, visando estimular maior organização e disciplina no programador. Isso se justifica devido ao fato de que as linguagens, ao mesmo tempo em que permitem a elaboração de grandes soluções, possibilitam que se cometa grandes erros, principalmente em programas maiores. Um programa pode estar sintaticamente correto e, no entanto conter erros como seqüência inadequada das instruções, valores incorretos ou também a sua execução pode nunca terminar normalmente, porque as condições de terminação nunca são atingidas (esses erros são chamados "bugs" ou erros de lógica).

Para a construção de programas grandes e com estrutura muitas vezes complexa demais para nossos limites atuais, foram (e ainda continuam sendo) criadas várias técnicas e metodologias de desenvolvimento e estruturação de programas mais fáceis de serem testados, alterados ou corrigidos. Mas não se iluda: qualquer que seja a linguagem ou a metodologia, é possível produzir programas desorganizados, incompreensíveis e repletos de erros ou que não fazem o que deveriam fazer. A qualidade final dos produtos dependerá predominantemente do programador.

## 4.2 Tipos de Instruções de Alto Nível

O que você pode fazer dentro de um programa, utilizando uma linguagem de alto nível? Vamos ver algumas analogias com outras máquinas.

Um televisor possui vários recursos: a tela, o alto-falante, o controle remoto, opções de configuração. Quando você aperta ou gira um botão de controle, está de certa forma instruindo o televisor a fazer algo; é uma forma rudimentar de linguagem. Em qualquer caso, você está limitado a atuar sobre os recursos do seu televisor; não há meios de instruí-lo a gravar algo numa fita de vídeo. Da mesma forma, numa linguagem de programação você não vai encontrar (a princípio) instruções para regular contraste ou temperatura. Uma categoria de instruções que você pode esperar que existam são aquelas que vão lhe permitir **interagir com os recursos disponíveis**: teclado e mouse (entrada), tela e impressora (saída), discos (entrada e saída), memória. Assim, há instruções para se ler algo do teclado e mostrar algo na tela. Sendo o disco rígido estruturado em pastas (diretórios) e arquivos, há instruções para se ler e gravar arquivos no disco rígido, e para se saber seu conteúdo. Também há instruções que declaram que você quer usar a memória, e instruções para armazenar e consultar dados nela armazenados.

Para poder executar uma gravação programada, seu videocassete precisa tomar pelo menos duas decisões: quando iniciá-la e quando interrompê-la. Assim é num programa: em várias situações será necessário fazer verificações e **tomar decisões** de quando executar ou não um conjunto de instruções, ou executar um ou outro conjunto. Por exemplo, um programa que lê uma nota que um aluno tirou em uma disciplina e deve informar se ele foi aprovado ou não. Ou o programa informa que o aluno passou ou que foi reprovado, mas nunca as duas coisas. Portanto, numa linguagem de alto nível, também você pode esperar que existam instruções para tomada de decisões.

Para explorar a enorme capacidade de processamento de um computador, que tal deixarmos para ele fazer coisas que exigem a repetição de um procedimento uma certa quantidade de vezes, sejam duas ou milhões de vezes? Você também encontrará numa linguagem instruções para programar **repetição condicional** de um grupo de instruções. Suponha que você quer escrever um programa que escreva seu nome em cada linha da tela. Ao invés de escrever 25 instruções, uma para cada linha, você pode usar uma instrução para repetir 25 vezes a instrução que mostra seu nome na tela. Em outros casos você terá um padrão a ser seguido, como no caso do cálculo da média aritmética: lê o número, calcula a soma parcial, incrementa um contador, lê outro número, calcula a soma parcial, incrementa um contador, e assim por diante; no final da leitura, divide a soma pelo contador. Uma vez identificado o padrão, você escreve apenas uma vez as instruções que se repetem e inclui um comando que controle a repetição.

Para facilitar a vida do programador, as linguagens de alto nível também contêm vários outros tipos de instruções: todas contêm **operações matemáticas** básicas (soma, subtração, multiplicação e divisão) e também funções matemáticas como seno, co-seno, logaritmo e exponencial. Algumas linguagens disponibilizam instruções gráficas, para desenhar retas, círculos e outros objetos, colorir objetos e muitas outras.

E o melhor dos mundos: você também poderá **criar suas próprias instruções e funções**, que depois de testadas e corretas, poderá usar como se fizessem parte do repertório da linguagem. Por exemplo, você poderá escrever uma instrução chamada RepeteMeuNome, que mostra seu nome na tela 25 vezes. Uma vez testada, correta e gravada, basta inserir o nome da instrução na posição apropriada do programa ("chamar" a instrução) para mostrar seu nome 25 vezes. E se quiser mostrar seu nome mais 25 vezes, basta "chamar" de novo a instrução. Para melhorar ainda mais, você pode criar a instrução de forma que a quantidade de repetições seja definida somente quando o programa for executado.

## 5 Conversão do Programa-Fonte

Como o computador não entende as instruções de um programa-fonte (ou código-fonte), para que este possa ser executado, ele precisa ser convertido para a linguagem que o computador reconhece, a **linguagem de máquina**. Uma instrução em uma linguagem de alto nível pode corresponder a centenas ou até milhares de instruções em linguagem de máquina. A conversão é feita por programas apropriados, e pode ser feita **antes** ou **durante** a execução do programa.

Quando o **programa-fonte** é todo convertido em linguagem de máquina *antes* da execução, esse processo é chamado de **compilação**, e os programas que o fazem são chamados **compiladores**. O programa resultante é chamado **programa-objeto** (ou código-objeto), que contém instruções em linguagem de máquina, mas ainda não pode ser executado; para que isso ocorra, é necessária outra etapa chamada "linkedição" ou **ligação**, efetuada também por um programa apropriado chamado "linkeditor" ou **ligador**. Na linkedição, são juntados ao código-objeto do programa outros códigos-objeto necessários à sua execução. Após esta etapa, a conversão está completa e produz finalmente um **programa executável**, observe a Figura 1.



**Figura 1:** Etapas para Construção de um Programa Executável pelo Computador

Quando a conversão é feita *durante* a execução, o processo é chamado de **interpretação**, e o programa conversor, **interpretador**. Neste caso, o interpretador permanece na memória, junto com o programa-fonte, e converte cada instrução e a executa, antes de converter a próxima. É evidente, portanto, que a execução de um programa interpretado é mais lenta que a de um compilado.

Note que não é a linguagem em si que é compilada e interpretada. Essa característica depende da *disponibilidade* de um **compilador** ou um **interpretador**; pode haver ambos para uma linguagem, ou até nenhum.



## 6 Software que você precisa ter ou conhecer

---

Alguns programas são necessários às atividades de um programador:

**Sistema Operacional:** como programador, você precisa ter um mínimo de informações sobre o sistema ou ambiente operacional em que vai trabalhar, como por exemplo:

- Como executar programas
- Os nomes que você pode dar aos arquivos
- Como organizar seus arquivos em pastas ou subdiretórios
- Como listar, excluir, renomear e copiar arquivos

Se o computador em que você trabalha está conectado a uma rede, você vai precisar de um nome registrado, associado a uma senha, para poder acessar arquivos e programas da rede.

**Editor de Textos:** um programa Pascal é um texto simples, sem caracteres de controle do tipo negrito, tamanho de fonte, paginação etc. Você vai precisar de um editor para digitar seu programa no computador. Editores que produzem tais textos são o Edit, que vem com o MS-DOS, o Bloco de Notas (NotePad), que vem com o Windows, e vários outros. Textos *formatados* pelo Word ou WordPad, por exemplo, não servem. Os compiladores comerciais normalmente trazem um editor; usá-los será normalmente mais prático.

**Compilador:** uma vez que você digitou o texto do programa, precisará convertê-lo para linguagem de máquina. Para isso você precisa de um compilador. Como compiladores comerciais da linguagem Pascal temos por exemplo as várias versões do **Turbo Pascal**, da Borland (<http://www.borland.com.br>), inclusive há versões para DOS e para Windows. Há vários outros, como o **Free Pascal**, que utilizaremos nesta disciplina e que você pode transferir via Internet gratuitamente através do endereço: <http://www.freepascal.org>



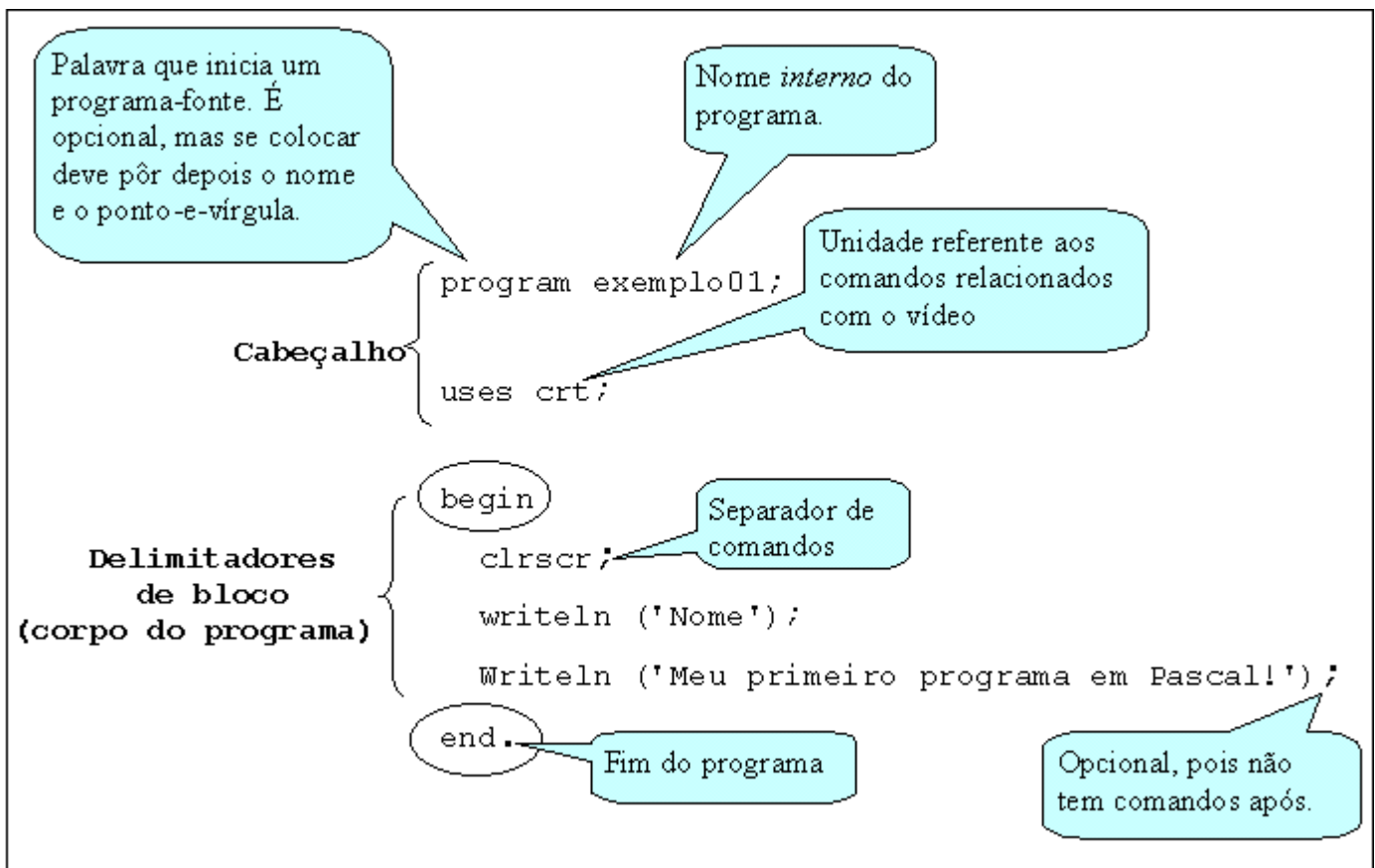
## 8 Estrutura de um Programa

Os programas são divididos em quatro áreas:

- 1a) Cabeçalho do programa
- 2a) Área de declaração de variáveis
- 3a) Área de definição de procedimentos e funções
- 4a) Corpo do programa (**algoritmo**; comandos)



Na linguagem Pascal não tem diferença entre maiúsculas e minúsculas. Mas é uma boa prática de programação utilizar a grande maioria dos caracteres em minúsculo. Utiliza-se maiúsculas somente em casos especiais. Um texto escrito com a maioria das letras em maiúsculo torna-se “pesado” visualmente para o leitor!



## Causa problemas não saber

\* **PROGRAM**, **begin** e **end** são *palavras reservadas*; têm um significado para o compilador, e você não pode usá-las para outra finalidade (como por exemplo, nomes de variáveis).

\* Ao final de todo comando ou declaração deve ter um ponto-e-vírgula, exceto antes de **end**.

\* O nome “interno” do programa é criado por você, no cabeçalho do programa, seguindo algumas regras:

- deve começar com uma letra
- a partir do segundo caractere, pode conter letras, dígitos ou o caractere sublinha ( \_ ), mas não pode conter espaços
- não são permitidas letras acentuadas

| Exemplos de <b>nomes válidos</b> de programas | Exemplos de <b>nomes inválidos</b> que impedem a compilação |
|---|---|
| CalculaFatorial                               | Calcula Fatorial  |
| Calc_Dobro_versao_2                           | 2_Calc_Dobro  |
| emite_som                                     | EmiteSom!   |

## 9 Tipos de Dados

---

**INTEIRO:** na linguagem Pascal há cinco tipos de inteiros pré-definidos. Cada tipo denota um subconjunto dos números inteiros de acordo com a Tabela 2.

| Tipo     | Faixa Numérica           | Número de Bytes |
|----------|--------------------------|-----------------|
| Shortint | -128 a 127               | 1               |
| Byte     | 0 a 255                  | 1               |
| Integer  | -32768 a 32767           | 2               |
| Word     | 0 a 65535                | 2               |
| Longint  | -2147483648 a 2147483647 | 4               |

**Tabela 2:** Tipos de Dados Inteiros

**REAL:** a Tabela 3 apresenta a faixa numérica, número de dígitos significativos e o tamanho do tipo Real.

| Tipo | Faixa Numérica                               | Número de Dígitos Significativos | Número de Bytes |
|------|--|----------------------------------|-----------------|
| Real | $-2,9 \cdot 10^{-39}$ a $+1,7 \cdot 10^{38}$ | 11 a 12                          | 6               |

**Tabela 3:** Tipo de Dado Real

**CARACTERE:** há duas maneiras de representar caracteres no Pascal. Veja a Tabela 4.

| Tipo   | Características   |
|--------|---|
| Char   | usado para armazenar caracteres ASCII <sup>1</sup>                              |
| String | armazena uma seqüência de caracteres com um comprimento variável entre 1 e 255. |

**Tabela 4:** Tipos de Dados Caractere

**Inteiro ou string?** Em alguns casos pode haver um dilema na representação de um dado que só contém dígitos: como string ou como inteiro? Esse é o caso de um número de telefone. Outros dados possuem dígitos e sinais, mas o armazenamento dos sinais é desnecessário, como em um CEP ou um CPF. Em princípio, você pode aplicar a regra: **se não serão feitas operações matemáticas com o dado, use o tipo *string***, a menos que tenha um bom motivo para não fazê-lo. Nessa questão (como em várias outras), a experiência será seu melhor professor ;-)

**LÓGICO:** variáveis lógicas podem assumir os valores *verdadeiro* ou *falso*. Veja suas características na Tabela 5.

| Tipo    | Características  |
|---------|--|
| Boolean | pode assumir o valor TRUE (verdadeiro) <b>ou</b> FALSE (falso) |

**Tabela 5:** Tipo de Dado Lógico

<sup>1</sup> **ASCII** (*American Standard Code for Information Interchange*) é um conjunto de códigos para o computador representar números, letras, pontuação e outros caracteres. Surgido em 1961, um dos seus inventores foi Robert W. Bemer. ASCII é uma padronização da indústria de computadores, onde cada caracter é manipulado na memória discos etc, sob forma de código binário. O código ASCII é formado por todas as combinações possíveis de 7 bits, sendo que existem várias extensões que abrangem 8 ou mais bits. Mais informações em: <http://pt.wikipedia.org/wiki/ASCII>

## 10 Operadores Aritméticos

A Tabela 6 apresenta os tipos de operadores aritméticos:

| Símbolo   | Significado      | Tipo dos Operandos         | Tipo do Resultado                                      | Prioridade     |
|---|------------------|----------------------------|--|----------------|
| *   | multiplicação    | inteiro * inteiro          | inteiro  | 1 <sup>a</sup> |
|   |                  | inteiro * real             | real   |                |
|   |                  | real * real                | real   |                |
| /   | divisão          | inteiro / inteiro          | real   | 1 <sup>a</sup> |
|   |                  | inteiro / real             | (sempre resulta real,<br>indiferente dos<br>operandos) |                |
|   |                  | real / real                |  |                |
| <b>div</b>  | divisão inteira  | inteiro <b>div</b> inteiro | inteiro  | 1 <sup>a</sup> |
| Exemplo: 11 div 4 = 2   |                  |                            |  |                |
| <b>mod</b>  | resto da divisão | inteiro <b>mod</b> inteiro | inteiro  | 1 <sup>a</sup> |
| Exemplo: 11 mod 4 = 3   |                  |                            |  |                |
| Observação1: <b>Soma</b> e <b>Subtração</b> são iguais a <i>Multiplicação!!!</i><br>Mas suas prioridades = 2 <sup>a</sup>   |                  |                            |  |                |
| Observação2: Não existe sinal para a <b>Potenciação</b> e <b>Radiciação</b> , devendo-se indicá-las com combinações de produtos ou funções predefinidas (ver cap 11.1). |                  |                            |  |                |

**Tabela 6:** Operadores Aritméticos

## 11 Expressões Aritméticas

As expressões aritméticas são escritas *linearmente*, usando-se a notação matemática. A Tabela 7 apresenta dois exemplos de representação de expressões aritméticas em Pascal:

| Expressão Aritmética                                    | Representação em Pascal                |
|---|--|
| $\sqrt{P \times (P - A) \times (P - B) \times (P - C)}$ | SqRt (P * (P - A) * (P - B) * (P - C)) |
| $A - B \times (C + \frac{D}{E - 1} - F) + G$            | A - B * (C + D / (E - 1) - F) + G      |

**Tabela 7:** Exemplo de Representação de Expressão Aritmética em Pascal  
(Horizontalização)

## 11.1 Funções Numéricas Predefinidas

Um programa, escrito em Pascal, pode fazer uso das seguintes funções numéricas, que são consideradas preexistentes na linguagem e têm prioridade sobre qualquer operação:

(EA = expressão aritmética)

| Nome        | Resultado                 | Tipo de Resultado |
|-------------|---------------------------|-------------------|
| Ln (EA)     | logaritmo neperiano de EA | real              |
| Exp (EA)    | número $e$ elevado a EA   | real              |
| Abs (EA)    | valor absoluto de EA      | real ou inteiro   |
| Trunc (EA)  | trunca o valor real de EA | inteiro           |
| Round (EA)  | arredonda o valor de EA   | inteiro           |
| Sqr (EA)    | quadrado de EA            | real ou inteiro   |
| SqRt (EA)   | raiz quadrada de EA       | real              |
| Sin (EA)    | seno de EA                | real              |
| Cos (EA)    | co-seno de EA             | real              |
| ArcTan (EA) | arco tangente de EA       | real              |

**Tabela 8:** Funções Numéricas Predefinidas

**Observação1:** As funções *Abs* e *Sqr* podem receber um argumento real ou inteiro e produzem, respectivamente, resultado real ou inteiro. As funções *Trunc* e *Round* recebem argumento real e produzem resultado inteiro. As demais recebem argumentos reais ou inteiros e produzem resultados reais.

**Observação2:** Não confundir  $Sqr(X)$  com  $SqRt(X)$ , que representam, respectivamente:

$$X^2 \quad \text{e} \quad \sqrt{X}$$

Note que:

*Sqr* = Square (quadrado)

*SqRt* = Square Root (raiz quadrada)

## 12 Funções de Formatação da Tela

---

**TextBackground:** seleciona a cor de fundo da tela. Valores válidos vão de 0 a 7.

Exemplo: `textbackground (1)`

Observação: Este comando deve ser colocado *antes* do comando de limpar a tela (`ClrScr`).

**TextColor:** seleciona a cor do texto. Valores válidos vão de 0 a 15.

Exemplo: `textcolor (14)`

Observação: Em vez de colocar o número correspondente a cor desejada, pode-se colocar o "nome" da cor em inglês. Exemplo: `textcolor (red)`

☛ A Tabela 9 abaixo apresenta os valores (números) correspondentes a cada cor:

|           |   |              |    |
|-----------|---|--------------|----|
| Black     | 0 | Darkgray     | 8  |
| Blue      | 1 | Lightblue    | 9  |
| Green     | 2 | Lightgreen   | 10 |
| Cyan      | 3 | Lightcyan    | 11 |
| Red       | 4 | Lightred     | 12 |
| Magenta   | 5 | Lightmagenta | 13 |
| Brown     | 6 | Yellow       | 14 |
| Lightgray | 7 | White        | 15 |

**Tabela 9:** Valores que Representam as Cores

## 13 Expressões Lógicas

---

Expressão lógica é uma expressão cujos operadores são os operadores lógicos e cujos operandos são *relações e/ou* variáveis do tipo lógico.

### 13.1 Relações

Uma *relação* é uma *comparação* realizada entre valores do mesmo tipo. Estes valores são representados na relação por constantes, variáveis ou expressões do tipo correspondente. A natureza da comparação é indicada por um dos operadores relacionais apresentados na tabela 10 a seguir:



| Símbolo da Operação  | Significado em Pascal |
|--|-----------------------|
| =  | igual                 |
| < >  | diferente             |
| < =  | menor ou igual        |
| <  | menor                 |
| >  | maior                 |
| > =  | maior ou igual        |
| in   | contido em (conjunto) |
| O resultado de uma relação é sempre um valor lógico, isto é, <b>true</b> ou <b>false</b> |                       |

**Tabela 10:** Operadores Relacionais

Exemplo:  $X + Y = Z$  é uma relação que será falsa ou verdadeira dependendo da expressão aritmética ter valor diferente ou igual ao valor da variável Z.

### 13.2 Operadores Lógicos

A Álgebra das Proposições define três conectivos usados na formação de novas proposições a partir de outras já conhecidas. Esses conectivos são os operadores nas expressões lógicas, veja na tabela abaixo:

| Operador | Função    |
|----------|-----------|
| and      | conjunção |
| or       | disjunção |
| not      | negação   |

A Tabela 11 abaixo apresenta um resumo do operador lógico **and**, onde o resultado final será *true* se todos os testes da expressão forem *true*.

| Teste           | Resultado |
|-----------------|-----------|
| true and true   | true      |
| true and false  | false     |
| false and true  | false     |
| false and false | false     |

**Tabela 11:** Resumo do Operador *and*

A Tabela 12 abaixo apresenta um resumo do operador lógico *or*, onde o resultado final será *false* se todos os testes da expressão forem *false*.

| Teste          | Resultado |
|----------------|-----------|
| true or true   | true      |
| true or false  | true      |
| false or true  | true      |
| false or false | false     |

**Tabela 12:** Resumo do Operador *or*

A Tabela 13 abaixo apresenta um resumo do operador lógico *not*.

| Teste     | Resultado |
|-----------|-----------|
| not true  | false     |
| not false | true      |

**Tabela 13:** Resumo do Operador *not*

### Combinações de Expressões Lógicas

Quando for necessário fazer uma ou mais verificações combinadas, como " $N1 = 0 \text{ e } N2 = 0$ ", " $N < 0$  ou  $N > 9$ ". podemos juntar duas ou mais expressões lógicas através dos conectivos lógicos **and** (e) e **or** (ou). Para o operador **and**, uma expressão será verdadeira somente quando *todas* as expressões que a compõem forem avaliadas como verdadeiras. Para o operador **or**, toda a expressão é verdadeira quando *qualquer* uma das expressões resultar verdadeira. Eles podem ser usados em expressões lógicas dos comandos **if**, **while** e qualquer outro que contenha expressões desse tipo. As expressões parciais *sempre* têm de estar entre parênteses. Por exemplo:

➤ Para saber se um número é de um dígito:

```
if (Numero >= 0) and (Numero <= 9) then { ... }
```

➤ No Pascal você *não* pode construir expressões lógicas do tipo " $0 \leq \text{Numero} \leq 9$ ", como na Matemática; você deve usar **and** ou **or**.

▶ Para verificar se uma variável do tipo caractere contém 'F' ou 'M':

```
if (TipoPessoa = 'F') or (TipoPessoa = 'M') then {...}
```

▶ Verificar se um número está entre duas faixas numéricas: 0 a 100 ou 200 a 300:

```
if ((Num>0) and (Num<100)) or (Num>200) and (Num<300)) then {...}
```

▶ Um operador muito interessante e conciso é o **in**, através do qual você verifica se um valor ou resultado de expressão pertence a um *conjunto*, definido entre colchetes e no qual podem constar valores ou faixas numéricas. Veja como os comandos acima ficam mais simples com esse operador:

```
if TipoPessoa in [ 'F', 'J' ] then ...
```

```
if Num in [ 0..100, 200..300 ] then ...
```

### 13.3 Prioridades

Pode-se ter mais de um operador lógico na mesma expressão, além dos operadores de relação e dos operadores aritméticos. Em Pascal, a prioridade das operações está dada na Tabela 14.

| Prioridade     | Operadores              |
|----------------|-------------------------|
| 1 <sup>a</sup> | not                     |
| 2 <sup>a</sup> | *, /, div, mod, and     |
| 3 <sup>a</sup> | +, -, or                |
| 4 <sup>a</sup> | =, <>, <, >, <=, >=, in |

**Tabela 14:** Prioridade das Operações

Vários níveis de parênteses também podem ser usados em expressões lógicas para fixar entre os operadores uma ordem de execução, diferente da indicada pela prioridade dada na Tabela 14.

**Sugestão:** use parênteses para definir a ordem de execução das operações e durma tranqüilo :-D

## 14 Estrutura Condicional (Seleção)

---

Na linguagem Pascal existem duas estruturas de seleção: *if-then-else* e *case*, sendo que a primeira estrutura pode-se apresentar de duas formas (simples ou composta).

### 14.1 IF-THEN-ELSE

#### a) Forma Simples (*sem Else*):

```
if condição then
begin
    seqüência A de comandos;
end
;
```

onde:

**if - then** - são palavras-chaves

condição - é uma expressão lógica (teste)

A “seqüência A de comandos” será executada **se** a condição for verdadeira; caso contrário, o comando a ser executado será o que vier logo após o **end;**

Ao ser mencionada "seqüência de comandos" está implícito que ela contém *um ou mais comandos* e pode conter uma ou mais estruturas.

Quando a seqüência A de comandos é constituída por *um único comando*, o **begin** e o **end** podem ser omitidos.

#### **Exemplo1:**

```
program Teste_If_Simples_1;
uses crt;
var A, B, C : real;
begin
    clrscr;
    write ('Digite um valor para A: ');
    readln (A);
    write ('Digite um valor para B: ');
    readln (B);
    write ('Digite um valor para C: ');
    readln (C);
    if (A + B < C) then
        writeln ('A soma de A com B é menor do que C!')
    ;
    writeln ('Tchau...');
end.
```

**Exemplo2:**

```
program Teste_If_Simples_2;

uses crt;

var A, B, C : real;

begin
  clrscr;
  write ('Digite um valor para A: ');
  readln (A);
  write ('Digite um valor para B: ');
  readln (B);
  write ('Digite um valor para C: ');
  readln (C);

  if (A + B < C) then
    begin
      writeln ('Eu calculei A + b < C e concluí que:');
      writeln ('A soma de A com B é menor do que C!');
    end
  ;
  writeln ('Tchau...');
end.
```

**b) Forma Composta (com Else):**

```
if condição then
  begin
    seqüência A de comandos;
  end
else
  begin
    seqüência B de comandos;
  end
;
```

onde:

**if - then - else** - são palavras-chaves

condição - é uma expressão lógica

Se a condição for verdadeira, a seqüência A de comandos é executada e, a seguir, a estrutura é abandonada, passando a execução para o comando que vier logo após o término da estrutura condicional composta e, neste caso, a seqüência B de comandos não será executada. Se a condição for falsa, a seqüência A de comandos será saltada (e não executada) e a seqüência B de comandos será processada.

Quando a seqüência A de comandos e/ou a seqüência B de comandos forem constituídas por *um único comando*, o **begin** e o **end**, que as contém, podem ser omitidos.

**Exemplo1:**

```

program Teste_If_Composto_1;

uses crt;

var A, B : real;

begin
  clrscr;
  write ('Digite um valor para A: ');
  readln (A);
  write ('Digite um valor para B: ');
  readln (B);
  if (A = B) then
    writeln ('A é igual a B')
  else
    writeln ('A não é igual a B')
  ;
  writeln ('Tchau...');
end.

```

---

**Exemplo2:**

```

program Teste_If_Composto_2;

uses crt;

var A, B, X, Y : real;

begin
  clrscr;
  write ('Digite um valor para A: ');
  readln (A);
  write ('Digite um valor para B: ');
  readln (B);
  if (A = B) then
    begin
      X := 1.5;
      Y := 2.5;
    end
  else
    begin
      X := -1.5;
      Y := -2.5;
    end
  ;
  writeln ('X= ', X:3:1, ' Y= ', Y:3:1); {Formato de saída
                                         3 = total de caracteres
                                         1 = caracteres depois do ponto}
end.

```

**Observação:** Observe que é o ponto-e-vírgula que encerra o if (é o equivalente ao *fim\_se* do português estruturado).

## 14.2 CASE

```
case variável of
  valordeCaso1 : seqüência A de comandos;
  valordeCaso2 : seqüência B de comandos;
  valordeCaso ... : seqüência C de comandos;
else
  seqüência D de comandos;
end;
```

onde:

**case - of - else - end** - são palavras-chaves

valordeCaso - são os possíveis valores que a variável pode assumir

Como a estrutura IF, o comando CASE divide uma seqüência de possíveis ações em seções de código individuais. Para a execução de um determinado comando CASE, somente uma dessas seções será selecionada para execução. A seleção está baseada numa série de testes de comparação, sendo todos executados sobre um valor desejado.

O comando ELSE é opcional, ou seja, assim como no IF, pode não haver a opção ELSE.

### **Exemplo:**

```
program Teste_Case;
uses crt;
var numero: integer;
begin
  clrscr;
  write ('Digite um número inteiro: ');
  readln (numero);
  case numero of
    1 : writeln ('O número digitado foi 1');
    2 :
      begin
        writeln ('Eu sei qual o número que você digitou');
        writeln ('O número digitado foi 2');
      end;
    3,4 : writeln ('O número digitado foi 3 ou 4 ');
    5..10 : writeln ('O número digitado está entre 5 e 10');
  else
    writeln ('O número digitado não está entre 1 e 10 ');
  end;
end.
```

### Observações da Estrutura Case

- 1) Na linguagem Pascal, o Case só testa igualdade ( = ), ou seja, não é possível utilizar um teste com >= por exemplo.
- 2) Os testes (as comparações) só podem ser feitas com constantes e não com variáveis.
- 3) A variável do Case deve ser do tipo: integer, char ou boolean.
- 4) A linguagem Pascal permite intervalos de números no Case, por exemplo:

```
case X of
    1..5 : writeln ('Azul');
    20..30,45 : writeln ('Vermelho');
    10,5 : writeln ('Amarelo');
```

## 15 Estruturas de Repetição

---

Na linguagem Pascal, existem três estruturas de repetição: *while-do*, *repeat-until* e *for-to-do*, sendo que esta terceira estrutura pode-se apresentar de duas formas. Veja abaixo as características e sintaxe das três estruturas:

### 15.1 WHILE-DO

```
while condição do
begin
    seqüência A de comandos;
end
;
```

onde:

**while - do** - são palavras-chaves

condição - é uma expressão lógica

Nesta estrutura, a seqüência A de comandos será repetida **enquanto** a **condição for verdadeira**. Quando isto não mais ocorrer, a repetição é interrompida e a seqüência de comandos, que estiver logo após o **end**; da estrutura, passa a ser executada.



**Exemplo:**

```
program Historia;  
  
uses crt;  
  
var Resp : char;  
  
begin  
  clrscr;  
  write ('Você quer ler uma história? [S]im ou [N]ão ');  
  readln (Resp);  
  while (Resp = 'S') or (Resp = 's') do  
    begin  
      writeln ('Era uma vez um bolo inglês...');  
      writeln ('Quer que eu conte outra vez? [S]im ou [N]ão ');  
      readln (Resp);  
    end  
  ;  
  writeln ('Tchau...');  
end.
```

**15.2 REPEAT-UNTIL**

|   |
|---|
| <b>repeat</b><br>seqüência A de comandos;<br><b>until</b> condição; |
|---|

onde:

**repeat - until** - são palavras-chaves

condição - é uma expressão lógica

Nesta estrutura, a seqüência A de comandos será **repetida até** que a **condição se torne verdadeira**. Quando isto ocorrer, a repetição é interrompida e a seqüência de comandos, que estiver logo após a estrutura, passa a ser executada.

Como o **until** é um delimitador, *não é necessário* colocar-se um ponto-e-vírgula (;) após o comando que o precede.

**Exemplo:**

```
program Historia_2;

uses crt;

var Resp : char;

begin
  clrscr;
  repeat
    writeln ('Era uma vez um bolo inglês...');
    writeln ('Quer que eu conte outra vez? [S]im ou [N]ão ');
    readln (Resp);
  until (Resp = 'N') or (Resp = 'n');
  writeln ('Tchau...');
end.
```

**15.3 FOR-TO-DO**

```
for variável := valor-inicial to valor-final do
begin
  seqüência A de comandos;
end
;
```

onde:

**for - to - do** - são palavras-chaves

valor-inicial - é o primeiro valor que a variável assume

valor-final - é o valor máximo que a variável pode assumir

Nesta estrutura, a variável recebe o valor-inicial; verifica se ele ultrapassa o valor-final; se não ultrapassa, a seqüência A de comandos é executada; a seguir, **a variável recebe o valor sucessor** (o laço é **crescente**); verifica se ele ultrapassa o valor-final; se não ultrapassa, a seqüência A de comandos é executada; e assim sucessivamente.

▶ Outro modo de usar o **for** é:

```
for variável := valor-inicial downTo valor-final do
  begin
    seqüência A de comandos;
  end
;
```

onde:

**for - downTo - do** - são palavras-chaves

valor-inicial - é o primeiro valor que a variável-de-controle assume

valor-final - é o valor máximo que a variável-de-controle pode assumir

Nesta estrutura, a variável recebe o valor-inicial; verifica se ele ultrapassa o valor-final; se não ultrapassa, a seqüência A de comandos é executada; a seguir, **a variável recebe o valor predecessor** (o laço é **decrescente**); verifica se ele ultrapassa o valor-final; se não ultrapassa, a seqüência A de comandos é executada; e assim sucessivamente.

**Exemplo1:**

```
program Teste_For_To;
uses crt;

var i : integer;

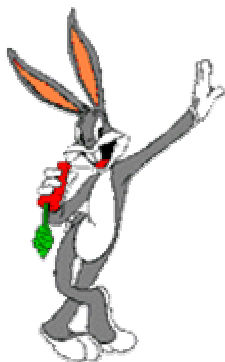
begin
  clrscr;
  for i := 1 to 10 do
    writeln (i)
  ;
end.
```

**Exemplo2:**

```
program Teste_For_downTo;
uses crt;

var x : integer;

begin
  clrscr;
  for x := 10 downTo 1 do
    writeln (x)
  ;
end.
```

**Dica:**

Até que tenha sido armazenado um valor pela primeira vez em uma variável, seu conteúdo é **indefinido**, e pode ser qualquer coisa; você é que deve cuidar disso inicializando (zerando) as variáveis antes de utilizá-las, ok? ;-)

## 16 Expressões Literais

---

As expressões do tipo **char** são formadas por uma constante, uma variável ou a ativação de uma função do tipo char. Existem, porém, funções predefinidas, tais como as apresentadas abaixo, na Tabela 15:

| Nome     | Resultado Fornecido   |
|----------|---|
| Succ (X) | o sucessor de X, no conjunto de caracteres considerado, se existir<br>Exemplo: Succ ('A') = 'B' <i>ou</i> Succ (1) = 2  |
| Pred (X) | O predecessor de X, no conjunto de caracteres considerado, se existir<br>Exemplo: Pred ('D') = 'C' <i>ou</i> Pred (100) = 99  |
| Ord X)   | É inteiro e indica a ordem de X (variável ou constante do tipo Char), no conjunto de caracteres usado na implementação<br>Exemplo: Ord ('C') = 67    ( <i>código ASCII</i> )          |
| Chr (EA) | Caractere que corresponde à ordem, dada pelo valor da expressão aritmética EA, se existir, no conjunto de caracteres utilizados<br>Exemplo: Chr (65) = 'A'    ( <i>código ASCII</i> ) |

**Tabela 15:** Funções Literais Predefinidas

As funções `Ord (X)`, `Succ (X)` e `Pred (X)` aplicam-se não apenas às variáveis ou constantes do tipo **char**, mas de qualquer tipo predefinido, **exceto real**.

## 17 Funções Predefinidas

---

**UpCase (X)** - recebe um argumento X do tipo char e retorna uma versão em letras maiúsculas da variável X.

### Exemplo:

```
program teste_upcase;
uses crt;

var letra : char;

begin

    letra := 'a';
    writeln (letra);
    writeln (upcase(letra));
    readln;
end.
```

>> Os dois comandos *writeln* deste programa produzem as seguintes linhas na tela de saída:

```
a
A
```

**ReadKey** - é uma outra importante função localizada na unidade CRT. Esta função espera que o usuário pressione uma tecla e depois retorna com um char da mesma.

### Exemplo:

```
program MostraTecla;
uses crt;

var tecla : char;

begin
    writeln ('Pressione uma tecla');
    tecla := ReadKey;
    writeln ('Você pressionou a tecla ', tecla);
    writeln ('Pressione <Enter> para terminar');
    readln;
end.
```

ReadKey é uma função que retorna o caractere correspondente à tecla pressionada. Uma vantagem de usar ReadKey para ler um caractere é que o usuário não precisa teclar Enter, como no ReadLn. Sendo uma função, podemos chamar ReadKey dentro de um Write ou atribuir seu resultado a uma variável.

**Observação:** O programa acima funcionará para teclas de caracteres “normais”, como alfabéticos e numéricos. Para teclas de função ou normais alteradas por uma tecla de controle, como Shift, Alt e Control, ReadKey retornará zero e será necessário chamá-lo novamente para obter um segundo código. Para tratar essa situação, são necessárias outras instruções.

**IMPORTANTE:** *As funções UpCase e ReadKey, assim como outras existentes na linguagem Pascal, funcionam somente em alguns compiladores, ou seja, para que elas funcionem adequadamente, verifique se o compilador que você está utilizando suporta tais funções.*

### Armazenando Caracteres na Memória

→ Se em um byte de memória podemos apenas representar números de 0 a 255, como então representar letras e outros caracteres, que constituem os valores dos tipos de dados char e string?

Letras e outros símbolos não numéricos são armazenados na memória também na forma de números, sendo a correspondência feita através de tabelas que associam letras a códigos numéricos. Nos computadores padrão PC é usada a tabela ASCII (*American Standard Code for Information Interchange*), que vai de 1 a 255 (1 byte, portanto). Nela, além de letras, números e sinais de pontuação, existem códigos numéricos para caracteres de controle (por exemplo, o 13 corresponde ao ENTER e o 27 ao ESC) e desenhos de linhas simples e duplas (retas horizontais, verticais e cantos). Para letras, há códigos distintos para maiúsculas e minúsculas: o código 66 corresponde a 'B', e o 98 corresponde a 'b'. Há um código também para o caractere 'espaço', que é o 32.

Assim, o número 66 armazenado num byte de memória é interpretado de forma diferente, conforme o tipo de dado com que foi definido a variável ali armazenada. Se o dado for, por exemplo, de um tipo numérico que ocupa um byte, é o número 66 mesmo. Se o byte for parte da representação de uma cadeia de caracteres, é interpretado como o caractere 'B'. Para o compilador, letras não se misturam com números. No caso de um dado do tipo cadeia de caracteres, mesmo se o número armazenado for 50, é interpretado como o caractere '2', e não o número 2. Para representar a seqüência de caracteres '65', são necessários 2 bytes: um para o '6' e outro para o '5'.

## 18 Definição de Constantes

---

Na seção **Const**, você define identificadores que representam **valores fixos** de dados por toda a execução de um programa. Esses identificadores podem ser usados como **sinônimos** de constantes, desde que tenham sido previamente definidos como tal.

A forma geral para a definição de constantes é apresentada a seguir:

```
const nomedaconst = valordaconst; nomedaconst2 = valordaconst2 ...
```

onde:

**const** - é uma palavra-chave que inicia uma parte do programa para a definição de constantes

nomedaconst - é qualquer identificador permitido pela linguagem seguindo as mesmas regras para nomes de variáveis

valordaconst - é um número (com ou sem sinal), um valor lógico, uma string etc.

O uso de identificadores como sinônimos de constantes é recomendável na medida em que *aumenta a legibilidade do programa e ajuda na sua documentação*. Além disso, os programas que manipulam valores ganham maior flexibilidade, uma vez que estes valores, se agrupados no início do programa, podem ser facilmente notados e/ou modificados (atualizados).

### Exemplo:

```
program salario;

uses crt;

const salario_min = 180.00; {não precisa declarar o tipo da
constante, pois é um "sinônimo" p/ 180.00}

var qtos_sal : integer;
    sal_total : real;

begin
    write ('Quantos salários mínimos você recebe? ');
    readln (qtos_sal);
    sal_total := salario_min * qtos_sal;
    writeln ('Salário total: ', sal_total:8:2);
    {"8:2" é a formatação de saída do número}
end.
```

## 19 Definição de Tipos

---

Embora a linguagem Pascal ofereça uma variedade de tipos de dados predefinidos (integer, real, boolean, char, etc.), nem sempre os tipos de dados disponíveis na linguagem atendem perfeitamente às necessidades do programa. Uma característica da linguagem Pascal é a possibilidade que o programador tem de *declarar seus próprios tipos de dados*. Uma vez criado o novo tipo, pode-se declarar quantas variáveis quiser do tipo criado. Usa-se a palavra reservada **Type** para declarar um tipo de dado **definido pelo programador**.

A seção Type é declarada *antes* da seção Var, pois na seção Var pode-se declarar as variáveis que pertencem aos tipos padrão e aos tipos definidos pelo usuário. Veja os exemplos a seguir:

### Exemplo1:

```
program exemplo_type;

uses crt;

type valor = real;

var salario : valor;

begin
    salario := 200.00;
    ...
end;
```

### Exemplo2:

```
...
type tIdade = 1..120; {declaração de um tipo chamado tIdade que
representa um range (conjunto) de 1 a 120}
    tMaiusc = 'A'..'Z';

var Idade      : tIdade;
    Ch1, Ch2   : tMaiusc;

...
end;
```

### Exemplo3:

```
...
type vet = array [1..10] of integer; {declaração de um tipo
chamado vet que representa um vetor de 10 elementos inteiros}

var x : vet;

...
end;
```



## 20 Estruturas de Dados

---

Existem dois tipos importantes de estruturas de dados estudados nesta disciplina, os **Arrays** e os **Registros**. Um array é uma estrutura que representa listas (*vetores*) ou tabelas (*matrizes*) de valores sob um único nome de variável. Um registro (*record*) é uma estrutura que representa vários tipos de dados distintos.

### 20.1 Array

Variáveis do tipo array são **variáveis compostas homogêneas**, ou seja, correspondem a posições de memória, identificadas por um único nome, individualizadas por *índices* e seu conteúdo é de um mesmo tipo. Pode ser um **Vetor** ou uma **Matriz**.

O nome de uma variável composta segue as mesmas regras das variáveis simples. O nome refere-se, coletivamente, a todos os elementos da variável composta. Para referência de um elemento, é necessário colocar o nome da variável, seguido de um ou mais *índices*, entre colchetes.

#### 20.1.1 Como declarar e acessar VETORES em Pascal

>> *Como declarar um VETOR:*

```
program exemplo_vetor;

uses crt;

var nota : array [1..10] of real; {declaração de um array com 10 elementos do
                                  tipo real, que representa um vetor, pois é
                                  unidimensional}
```

>> *Como acessar um VETOR:*

```
x := nota [3]; {variável x recebe o conteúdo armazenado na 3a posição do array nota}

i := 4; {variável i recebe o valor 4}

writeln (nota [8]); {escreve o conteúdo armazenado na 8a posição do array nota}

writeln (nota [i]); {escreve o conteúdo armazenado na 4a posição do array nota}
```

Tenha em mente que cada elemento de um vetor é uma variável semelhante às outras, podendo sofrer atribuição, ser passado como parâmetro ou participar como operando em expressões, desde que você indique qual é o elemento, através do **índice**.

A forma de referenciar elementos de vetores é idêntica à maneira de referenciar caracteres específicos de uma string; na verdade, **strings são vetores do tipo char !**

### 20.1.2 Como declarar e acessar MATRIZES em Pascal

>> *Como declarar uma MATRIZ:*

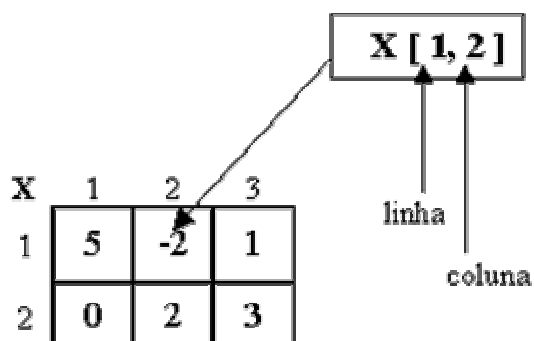
```
program exemplo_matriz;
```

```
uses crt;
```

```
var x : array [1..2, 1..3] of integer;
```

{declaração de um array com 2 linhas e 3 colunas do tipo integer, que representa uma **matriz**, pois é multidimensional; neste exemplo é bidimensional (linhas e colunas)}

▶ Exemplo de uma matriz X de 2 linhas e 3 colunas:



>> **Como acessar uma MATRIZ:**

```
a := x [2,1]; {variável A recebe o conteúdo armazenado na 2a linha e 1a coluna do array X}
```

```
i := 2; {variável I recebe o valor 2}
```

```
writeln (x [1,3]); {escreve o conteúdo armazenado na 1a linha e 3a coluna do array X}
```

```
j := 3; {variável J recebe o valor 3}
```

```
writeln (x [i,j]); {escreve o conteúdo armazenado na 2a linha e 3a coluna do array X}
```

**Observação:** Veja abaixo um exemplo de leitura e escrita de uma matriz M de 2 linhas e 3 colunas em Pascal

#### 📌 *Leitura da Matriz M [2,3]*

```
for i := 1 to 2 do
  for j := 1 to 3 do
    begin
      writeln ('Digite um valor inteiro para a matriz: ');
      readln (M [i,j]);
    end
  ;
;
```

#### 📌 *Escrita da Matriz M [2,3]*

```
for i := 1 to 2 do
  for j := 1 to 3 do
    begin
      writeln (M [i,j]);
    end
  ;
;
```

**ATENÇÃO:** *As diferenças básicas entre Vetores e Matrizes são:*

- Na declaração de matrizes, devemos informar mais de um conjunto de índices, ou seja, uma faixa numérica ("tamanho") para cada dimensão da matriz:

*Exemplo:*

```
var M:array [1..10, 1..20, 1..5] of byte; {declaração de uma matriz com 3 dimensões}
```

- Ao acessar um elemento de uma matriz, devemos informar um índice para cada dimensão:

*Exemplo:*

```
Matriz [1, 11, 2] := 77;  
Readln (Matriz [9, 7, 1]);  
Writeln (Matriz [2, 5, 5] );
```

## 20.2 Registro (Record)

Um record é um conjunto de valores individuais de dados chamados **campos**. Uma variável record é, portanto, um identificador único que representa todos os valores dos campos de um determinado record. A característica mais importante de um record talvez seja a de que os diversos campos nomeados possam ter *tipos de dados diferentes*, por isso, variáveis do tipo record são **variáveis compostas heterogêneas**.

Existem duas maneiras de se definir uma variável do tipo record:

➤ Definir a estrutura record em uma declaração TYPE e, depois declarar a própria variável em um comando VAR

ou

➤ Definir a estrutura e a variável record de uma só vez em um comando VAR

A primeira destas técnicas é, provavelmente, a mais clara e mais utilizada, mas vamos explorar as duas maneiras nesta disciplina.

### 20.2.1 Como declarar e acessar REGISTROS em Pascal

>> *Como declarar um REGISTRO sem o uso do Type :*

```
program exemplo_registro;
uses crt;
var Cliente : record
    Nome, Ender : string [35];
    Fone : string [20];
    Sexo : char;
    Idade : byte;
    Salario : real;
end;
...
```

No exemplo acima, é declarada uma variável do tipo registro de nome Cliente que possui seis campos, onde os campos Nome, Ender e Fone são do tipo string, o campo Sexo é do tipo char, o campo Idade é do tipo byte e o campo Salario é do tipo real.

>> **Como acessar o REGISTRO** *Cliente*, declarado acima:

```
write ('Nome: '); {pede que o usuário entre com um nome}

readln (cliente.nome); {armazena o nome lido no campo nome do registro cliente}

cliente.sexo := 'F'; {o campo sexo do registro cliente recebe a letra F}

writeln (cliente.sexo); {escreve o conteúdo do campo sexo do registro cliente}
```

>> **Como declarar um REGISTRO** *com o uso do Type* :

```
program exemplo_registro;
uses crt;
type Pessoa = record
    Nome, Ender : string [35];
    Fone : string [20];
    Sexo : char;
    Idade : byte;
    Salario : real;
end;
var funcionario, aluno : Pessoa;
...
```

No exemplo acima, é declarado um tipo chamado Pessoa que representa um registro com seis campos, onde os campos Nome, Ender e Fone são do tipo string, o campo Sexo é do tipo char, o campo Idade é do tipo byte e o campo Salario é do tipo real. Após, são declaradas duas variáveis: funcionário e aluno, do tipo Pessoa definido antes pelo programador.

>> **Como acessar o REGISTRO** *Pessoa*, declarado acima:

```
...
write ('Nome: '); {pede que o usuário entre com um nome}
readln (funcionario.nome); {armazena o nome lido no campo nome da variável
                           funcionário que é do tipo Pessoa (que é um registro)}
aluno.sexo := 'F'; {o campo sexo da variável aluno recebe a letra F}
writeln (aluno.sexo); {escreve o conteúdo do campo sexo da variável aluno}
...
```

## 20.2.2 Exemplos de Registros

### ↳ Vetor de Registros

```
...  
  
type pessoa = record  
    nome : string [30];  
    idade : byte;  
    nota : real;  
end;  
turma = array [1..10] of pessoa;  
  
var aluno : turma;  
...
```

→ Esse exemplo acima seria um cadastro mais ou menos assim:

|     | Nome | Idade | Nota |
|-----|------|-------|------|
| 1   |      |       |      |
| 2   |      |       |      |
| ... |      |       |      |
| 10  |      |       |      |

→ Atribuindo valores, por exemplo:

```
aluno[1].nome := 'João';
```

```
aluno[1].idade := 20;
```

```
aluno[1].nota := 8.5;
```

## ➔ Registro dentro de Registro

```

...

type regdata = record
    dia, mes, ano : word;
end;
regpessoa = record
    nome : string [30];
    data_nasc : regdata;
    salario : real;
end;

var aluno : regpessoa;

...

```

➔ Esse exemplo acima seria um cadastro mais ou menos assim:

|     | Nome | Data_Nasc |     |     | Salário |
|-----|------|-----------|-----|-----|---------|
|     |      | Dia       | Mês | Ano |         |
| 1   |      |           |     |     |         |
| 2   |      |           |     |     |         |
| ... |      |           |     |     |         |
| 10  |      |           |     |     |         |

➔ Atribuindo valores, por exemplo:

```

aluno.data_nasc.dia := 10;
aluno.data_nasc.mes := 02;
aluno.data_nasc.ano := 1981;

```



## ▀ Vetor dentro de Registro

```
...
type regdata = record
    dia, mes, ano : word;
    end;

    regnotas = array [1..10] of real;

    regpessoa = record
        nome : string [30];
        data_nasc : regdata;
        salario : real;
        notas : regnotas;
    end;

    regturma = array [1..10] of regpessoa;

var aluno : regturma;

begin
...

    aluno[1].notas[1] := 8.0;
    aluno[1].data_nasc.dia := 12;

...

```

## 21 Comando WITH

---

É um comando que permite abreviar os identificadores de campos usados dentro de um determinado bloco de comandos, ou seja, permite referências abreviadas a campos de uma variável do tipo registro.

Por exemplo, digamos que se tenha declarado um registro da seguinte maneira:

```
...  
var endereco : record  
    rua : string [30];  
    cidade : string [25];  
    estado : string [2];  
end;  
...
```

>> *Acessando os campos deste registro declarado acima:*

```
...  
  
write ('Digite a rua: ');  
readln (endereco.rua);  
write ('Digite a cidade: ');  
readln (endereco.cidade);  
write ('Digite o estado: ');  
readln (endereco.estado);  
  
...
```

>> *Agora, acessando os campos do registro "endereco", com o uso do comando WITH:*

```
with endereco do  
    begin  
        write ('Digite a rua: ');  
        readln (rua);  
        write ('Digite a cidade: ');  
        readln (cidade);  
        write ('Digite o estado: ');  
        readln (estado);  
    end  
;
```

**Observação:** Este comando WITH (assim como vários outros) não funciona em todos os compiladores, somente em alguns :-)

## 22 Modularização

---

Resolver um problema complexo é mais fácil se não precisarmos considerar todos os aspectos do problema simultaneamente, ou seja, podemos decompor o problema em subproblemas (módulos).

A linguagem Pascal nos oferece duas maneiras de criarmos programas modulares e estruturados que são: **PROCEDIMENTOS** e **FUNÇÕES**. São unidades (trechos) de código de programa autônomas projetadas para cumprir uma tarefa particular. Para serem executados devem ser ativados por um programa principal, por outro procedimento ou por outra função. A comunicação entre programa principal, procedimentos e funções ocorre através de "chamadas". Veja alguns objetivos (algumas vantagens) do uso de procedimentos e funções:

- Eliminar segmentos de programas usados várias vezes
- Melhorar a clareza dos programas
- Facilitar a modificação (atualização) dos programas

### 22.1 Procedimentos (Procedure)

**Exemplo:** Escreva um procedimento que imprima uma linha de vinte asteriscos na tela. Utilizar a posição corrente do cursor. Após a impressão dos asteriscos o cursor deve ficar posicionado no início da linha seguinte. (Procedimento *sem* passagem de parâmetros).

```

                                program exemplo;
                                uses crt;
                                procedure linha;
                                var i: integer; {variável local (da procedure)}
                                begin
                                    for i := 1 to 20 do
                                        write ('*');
                                    writeln;
                                end;
                                begin
                                    clrscr;
                                    linha; {chamada do procedimento (ativação)}
                                end.
  
```

definição do procedimento

programa principal

### 🚩 Como melhorar este procedimento?

- Permitindo que seja impresso um número *variável* de asteriscos.

### 🚩 Como passar dados (informações) para o procedimento?

- Através de passagem de parâmetros.

Dentre os modos de transferência de parâmetros, pode-se destacar a **passagem por valor** e a **passagem por referência**.

## 22.1.1 Passagem de parâmetros POR VALOR

### Exemplo:

```
program exemplo;
uses crt;

var j: integer; {variável global: acessível por todos os módulos do programa}

procedure linha (n: integer); { n é um parâmetro formal }
var i: integer;
begin
    for i := 1 to n do
        write ('*')
    ;
    writeln;
end;

begin
    clrscr;
    linha (40);
    linha (20); {40 e 20 são parâmetros reais }
    for j := 1 to 10 do
        linha (j)
    ;
end.
```

### ➤ O procedimento deve ser definido ANTES do local onde é chamado!!!

>> **Parâmetros:** são canais pelos quais se estabelece uma comunicação bidirecional entre um procedimento e o programa chamador (procedimento ou programa principal).

**Parâmetros Formais:** são os nomes simbólicos introduzidos no cabeçalho dos procedimentos, usados na definição dos parâmetros dos mesmos.

**Parâmetros Reais:** são aqueles que substituem os parâmetros formais na chamada de um procedimento.

### ➤ Utilizando mais de um parâmetro

```
program exemplo;
uses crt;

var num: integer;
    car: char;

procedure linha (n: integer; simb: char);
var i: integer;
begin
    for i := 1 to n do
        write (simb)
    ;
    writeln;
end;

begin
    clrscr;
    write ('Digite um número: ');
    readln (num);
    write ('Digite um caractere: ');
    readln (car);
    linha (num, car);
end.
```

**ATENÇÃO:** Os parâmetros usados na codificação do procedimento e os argumentos descritos na chamada do subprograma devem coincidir em *número, ordem e tipo*.

>> Como o módulo pode retornar um valor para o local onde foi chamado? Utilizando uma função :-D

## 22.2 Funções (Function)

### Características das funções:

- Calcular um resultado
- **Retorna um valor ao ponto de sua chamada**, sendo que este será associado ao próprio nome que identifica o módulo.

### Exemplo: Função Soma

```
program exemplo_funcao;  
uses crt;  
  
var a, b, c: integer;  
  
function soma (x, y: integer): integer;  
var r: integer;  
begin  
    r := x + y;  
    soma := r; {retorno do valor}  
end;  
  
begin  
    clrscr;  
    write ('Digite um valor: ');  
    readln (a);  
    write ('Digite outro valor: ');  
    readln (b);  
    c := soma (a, b);  
    writeln ('Resultado: ', c);  
end.
```

Tipo de Saída



**Observação1:** \* O valor é *retornado* pelo nome da função  
\* Dessa forma só é possível retornar *um único* valor

**Observação2:** \* Qualquer modificação feita, em parâmetros passados por valor, *dentro* do procedimento, NÃO surtirá efeito *fora* dele.

>> **Como permitir que a modificação de uma variável dentro do procedimento tenha efeito em uma variável fora do procedimento?** Utilizando passagem de parâmetros por *referência* : -D

### 22.2.2 Passagem de parâmetros POR REFERÊNCIA

#### Exemplos de utilização da passagem por referência:

Passando um valor válido, permitindo a sua alteração dentro do módulo

```
program exemplo;

uses crt;

var x, y: integer;

procedure troca (var a, b: integer);
var aux: integer;
begin
    aux := a;
    a := b;
    b := aux;
end;

begin
    x := 10;
    y := 20;
    troca (x, y);
    writeln ('X= ', x, 'Y= ', y);
end.
```

### Para retornar mais de um valor:

>> Passamos uma variável com um valor indeterminado ("lixo") e retornamos o resultado desejado por ela. Veja exemplo a seguir.

**Exemplo:** Função *divide* que retorne o resultado da divisão e um código que indique se ela foi ou não bem sucedida. (0 = divisão OK, 1 = erro: divisão por zero).

```

program exemplo;

uses crt;


var x, y, cod: integer;
    r: real;

function divisao (a, b: integer; var erro: integer): real;
begin
    if b = 0 then
        erro := 1
    else
        begin
            erro := 0;
            divisao := a/b;
        end
    ;
end;

begin
    clrscr;
    write ('Digite um número: ');
    readln (x);
    write ('Digite outro número: ');
    readln (y);
    r := divisao (x,y,cod);
    if cod = 1 then
        writeln ('Divisão por zero')
    else
        writeln ('Resultado: ', r)
    ;
end.

```

Indica passagem por referência



**Observação1:** Passagem por referência pode ser usada tanto em funções quanto em procedimentos.

**Observação2:** Um procedimento só poderá chamar outro que tenha sido declarado *antes* dele.



## Como fazer um programa que faça uma divisão, utilizando modularização?

➤ Veja os seguintes módulos:

| Obtém_Resposta           | Calcular_Divisão |
|--------------------------|------------------|
| Entrada: nenhuma         | Entrada: nenhuma |
| Saída: resposta validada | Saída: nenhuma   |

| Obtém_um_Valor        | Resolve_Problema |
|-----------------------|------------------|
| Entrada: nenhuma      | Entrada: nenhuma |
| Saída: valor validado | Saída: nenhuma   |

| Obtém_todos_Valores             |
|---------------------------------|
| Entrada: nenhuma                |
| Saída: leitura dos dois valores |

➤ Veja como ficaria a programação destes módulos:

```

program exemplo;
uses crt;

{-----Função para obter a resposta}

function obtem_resposta: char;
var resp: char;
begin
  repeat
    readln (resp);
  until (resp = 'n') or (resp = 's');
  obtem_resposta := resp;
end;

```

```
{-----Função para obter um valor}
```

```
function obtem_um_valor: real;  
var b: real;  
begin  
  repeat  
    write ('B=');  
    readln (b);  
    if b = 0 then  
      writeln ('Valor Inválido');  
  until b <> 0;  
  obtem_um_valor := b;  
end;
```

```
{-----Procedimento para obter todos valores}
```

```
procedure obtem_todos_valores (var a, b: real);  
begin  
  write ('A=');  
  readln (a);  
  b := obtem_um_valor;  
end;
```

```
{-----Procedimento para calcular a divisão}
```

```
procedure calcula_divisao;  
var a, b, r: real;  
  x: char;  
begin  
  repeat  
    obtem_todos_valores (a,b);  
    write ('Dados corretos (S/N)? ');  
    x := obtem_resposta;  
  until x = 's';  
  r := a/b;  
  writeln ('Resultado: ', r);  
end;
```

```
{-----Procedimento para resolver o problema}

procedure resolve_problema;
var c: integer;
    x: char;
begin
    c := 0;
    repeat
        calcula_divisao;
        c := c + 1;
        write ('Novo Cálculo (S/N)? ');
        x := obtem_resposta;
    until x = 'n';
    writeln ('Foram calculadas ', c, ' divisões');
end;

{-----Programa principal}

begin
    clrscr;
    resolve_problema;
end.
```