Aprendizado de Algoritmos usando o Portugol Studio

APRESENTAÇÂO

O Portugol Studio é um ambiente para aprender a programar, voltado para os iniciantes em programação. Possui uma sintaxe fácil, diversos exemplos e materiais de apoio à aprendizagem. Também possibilita a criação de jogos e outras aplicações.

Este material foi compilado a partir dos textos e imagens presentes no contexto da ajuda do Portugol Studio do qual nos esforçamos para mantermos o mais fiél a obra original. Objetivamos com isso apenas contribuir com o acesso a essas informações fora do ambiente do programa para os estudantes que usam esta excelente ferramenta.

Link para Download do Portugol Studio:

https://sourceforge.net/projects/portugolstudio/

SUMÁRIO

Linguagem Portugol

Uma linguagem de programação é um método padronizado para comunicar instruções para um computador. É um conjunto de regras sintáticas e semânticas usadas para definir um programa de computador. Permite que um programador especifique precisamente sobre quais dados um computador vai atuar, como estes dados serão armazenados ou transmitidos e quais ações devem ser tomadas sob várias circunstâncias.

O Portugol é uma representação que se assemelha bastante com a linguagem C, porém é escrito em português. A ideia é facilitar a construção e a leitura dos algoritmos usando uma linguagem mais fácil aos alunos.

Sintaxe e semântica do Portugol

O compilador auxilia a verificar se a sintaxe e a semântica de um programa está correta.

Durante os tópicos da ajuda, serão apresentadas as estruturas básicas da linguagem.

Exemplo da estrutura básica

```
//O comando programa é obrigatório
programa
{
  //Inclusões de bibliotecas
  // - Quando houver a necessidade de utilizar
  //   uma ou mais bibliotecas, as inclusões
  //   devem aparecer antes de qualquer declaração

  /*
   * Dentro do programa é permitido declarar
   * variáveis globais, constantes globais e
   * funções em qualquer ordem.
   */

  //Declarações de funções somente
  //são permitidas dentro do programa.
  funcao inicio()
  {
    /*
     * Declarações de variáveis locais,
     * constantes locais, estruturas de
     * controle e expressões.
     */
  }
}
```

Bibliotecas

Em todo o algoritmo que se possa elaborar, existe a possibilidade da utilização de um conjunto de funções e comandos já existentes. A estes conjuntos de funções e comandos, dá-se o nome de Bibliotecas.

As bibliotecas contém códigos e dados auxiliares, que provém serviços a programas independentes, o que permite o compartilhamento e a alteração de código e dados de forma modular. Existem diversos tipos de bibliotecas, cada uma com funções para atender a determinados problemas.

Para se utilizar uma biblioteca é necessário primeiro importa-la para o seu programa.

No portugol para importar uma biblioteca usa-se as palavras reservadas **inclua biblioteca** seguido do nome da biblioteca que se deseja usar, e opcionalmente pode-se atribuir um apelido a ela usando do operador "-->" sem aspas seguido do apelido.

Para usar um recurso da biblioteca deve-se escrever o nome da biblioteca (ou apelido), seguido por um ponto e o nome do recurso a ser chamado como demonstrado abaixo.

```
inclua biblioteca Mouse
inclua biblioteca Graficos --> g
Mouse.ocultar_cursor()
g.iniciar_modo_grafico(verdadeiro)
```

No portugol, existem as seguintes bibliotecas:

- Arquivos
- Graficos
- Matematica
- Mouse
- Sons
- Teclado
- Texto
- Tipos
- Util

```
programa
{
    inclua biblioteca Matematica
    inclua biblioteca Texto --> t
    funcao inicio()
    {
        real resultado
        resultado = Matematica.arredondar(Matematica.PI,5)
        escreva(resultado)
        escreva(t.caracteres_maiusculos("texto"))

    }
}
```

Declarações

Sejam números ou letras, nossos programas tem que conseguir armazenar dados temporariamente para poderem fazer cálculos, exibir mensagens na tela, solicitar a digitação de valores e assim por diante.

Uma declaração especifica o identificador, tipo, e outros aspectos de elementos da linguagem, tais como variáveis, constantes e funções. Declarações são feitas para anunciar a existência do elemento para o compilador.

Para as variáveis, a declaração reserva uma área de memória para armazenar valores e ainda dependendo onde ela foi declarada pode ser considerada local (vai existir somente dentro de seu escopo) ou global (vai existir enquanto o programa estiver em execução).

Para as constantes a declaração funciona de forma parecida a de uma variável, porem sem a possibilidade de alterar seu valor no decorrer do programa.

Para as funções, declarações fornecem o corpo e assinatura da função.

Nesta seção, abordaremos os seguintes tipos de declarações:

- Declaração de Constante
- Declaração de Função
- Declaração de Matriz
- Declaração de Variáveis
- Declaração de vetor

Declaração de Constante

Existem algumas situações em que precisamos que um determinado parâmetro não tenha seu valor alterado durante a execução do programa. Para isso, existem as constantes. Constante é um identificador cujo valor associado não pode ser alterado pelo programa durante a sua execução.

Para declarar uma constante basta adicionar a palavra reservada **const** seguida do tipo de dado, pelo nome da constante e atribuir um valor a ela.

```
const inteiro NOME_DA_CONSTANTE = 3
const real NOME_DA_CONSTANTE2 = 45
```

Por uma questão de convenção, é aconselhável deixar o nome da sua constante em caixa alta (todas as letras em maiúsculo)

Para melhor compreensão deste conceito, confira o exemplo abaixo.

```
programa
{
  //Constante global do tipo de dado real
  const real aceleracao_gravidade = 9.78

  funcao inicio()
  {
    //Vetor constante local do tipo de dado caracter
    const caracter vogais[5] = {'a','e','i','o','u'}

    //Matriz constante local do tipo de dado inteiro
    const inteiro teclado_numerico[][] = {{1,2,3},{4,5,6},{7,8,9}}
  }
}
```

Declaração de Função

Se lhe fosse solicitado um algoritmo para preencher uma matriz, você o resolveria correto? Porém, se ao invés de uma matriz fossem solicitadas dez matrizes? Concordamos que o algoritmo ficaria muito cansativo e repetitivo. Mas, e se pudéssemos repetir o mesmo procedimento, quantas vezes necessárias, o escrevendo apenas uma vez? Nós podemos. Para isso, usamos uma função. Função consiste em uma porção de código que resolve um problema muito específico, parte de um problema maior.

Algumas das vantagens na utilização de funções durante a programação são:

- A redução de código duplicado num programa;
- A possibilidade de reutilizar o mesmo código sem grandes alterações em outros programas;
- A decomposição de problemas grandes em pequenas partes;
- Melhorar a interpretação visual de um programa;
- Esconder ou regular uma parte de um programa, mantendo o restante código alheio às questões internas resolvidas dentro dessa função;

Os componentes de uma função são:

- O seu protótipo, que inclui os parâmetros que são passados à função na altura da invocação;
- O corpo, que contém o bloco de código que resolve o problema proposto;
- Um possível valor de retorno, que poderá ser utilizado imediatamente a seguir à invocação da função.

A declaração de função no Portugol é realizada da seguinte forma: Deve-se utilizar a palavra reservada **funcao**, seguido do tipo de retorno. Quando o tipo de retorno é ocultado, o Portugol assume que o retorno é do tipo vazio. Então, deve-se definir o nome da função seguido de abre parênteses, uma lista de parâmetros pode ser incluída antes do fecha parênteses. Para concluir a declaração deve-se criar o corpo da função. O corpo da função consiste em estruturas dentro do abre e fecha chaves. Quando uma função possui um tipo de retorno diferente de vazio, é obrigatória a presença do comando retorne no corpo da função.

```
funcao real nome_da_funcao (inteiro parametro1,real parametro2)
{
   retorne parametro1 * parametro2
}
funcao inteiro nome_da_funcao2 ()
{
   retorne 1
}
funcao nome_da_funcao3 (cadeia &parametro)
{
   parametro = "Novo Valor"
}
```

A declaração dos parâmetros é similar a declaração de variável,vetor e matriz sem inicialização e devem ser separados por vírgula. Note que uma função do tipo vazio não tem retorno.

Para funções existem dois tipos de passagens de valores possíveis. São eles: por valor e por referência. A passagem de parâmetros por valor transfere para a função apenas o valor contido na variável, ou seja, a variável em si não terá seu conteúdo alterado. Já a passagem de parâmetro por referência transfere a variável como um todo, modificando a mesma de acordo com os comandos presentes no corpo da função.

Por padrão os parâmetros se comportam como passagem por valor, para o parâmetro se comportar como referência deve-se adicionar o símbolo & antes do nome do parâmetro.

Para melhor compreensão deste conceito, confira o exemplo abaixo.

```
programa
{
  //Função com retorno do tipo vazio sem parâmetro
  funcao vazio imprime_linha()
  {
    escreva("\n------------------------------\n")
  }

  //Função com retorno do tipo vazio e com um vetor como parâmetro
  funcao inicio(cadeia argumentos[])
  {
    //Imprime o retorno da função media
    escreva(media(4,9,8))

    imprime_linha()

    inteiro variavel = 123

    zera_valor(variavel)

    //Imprime 0
    escreva(variavel)

    imprime_linha()

    inteiro num=3

    //Verifica se o número 3 é par com uma resposta do tipo lógico
    escreva (num, " é par? ", verifica_par(num))
  }

  //Função com retorno do tipo real e três parâmetros do tipo inteiro
  funcao real media(inteiro m1, inteiro m2, inteiro m3)
  {
    retorne (m1 * 2 + m2 * 3 + m3 * 8) / 13.0
  }
```

Declaração de Matriz

Para a melhor compreensão do conceito de matriz, e interessante o entendimento de Vetores. Os vetores permitem solucionar uma série de problemas onde é necessário armazenamento de informações, porém ele possui a restrição de ser linear. Por exemplo, imagine que queremos armazenar três notas obtidas por quatro alunos diferentes. Neste caso, existe outra estrutura mais adequada para armazenar os dados. A matriz.

A matriz é definida como sendo um vetor com mais de uma dimensão (geralmente duas). Enquanto o vetor armazena as informações de forma linear, a matriz armazena de forma tabular (com linha e colunas).

```
//Função com retorno vazio e parâmetro por referência
funcao zera_valor(inteiro &valor)
{
   valor = 0
}

//Função com retorno do tipo lógico e parâmetro do tipo real
funcao  logico verifica_par(inteiro num)
{
      se (num % 2 != 0)
      {
         retorne falso
      }

      retorne verdadeiro
}
}
```

A tabela a seguir ilustra uma matriz que armazena três notas de quatro alunos:

Posições	0	1	2
0	10	9	6.7
1	6	8	10
2	8	7	4.5
3	5.2	3.3	0.3

Repare que cada linha da matriz representa um aluno que têm três notas (três colunas).

Assim como o vetor, a matriz também possui todos os elementos de um mesmo tipo. Na declaração de uma matriz temos sempre que indicar respectivamente o tipo de dado, nome da variável, número de linhas e colunas (nesta ordem) entre colchetes.

Para fazer acesso a um elemento da matriz, seja para preencher ou para consultar o valor, devemos indicar dois índices, uma para linha e outro para a coluna. O índice é um valor inteiro (pode ser constante ou uma variável) que aparece sempre entre colchetes "[ ]" após o nome do vetor.

```
//Declaração de uma matriz de numeros reais com 5 linhas e 3 colunas
real nome_da_variavel[5][3]
//Gravar um valor na matriz na posição 0 (primeira linha) e 1 (segunda coluna)
nome_da_variavel[0][1] = 2.5
```

Da mesma forma que o vetor, tentar acessar um índice fora do tamanho declarado irá gerar um erro de execução.

Para melhor compreensão deste conceito, confira o exemplo abaixo.

```
programa
{
  funcao inicio()
  {
    //Declaração de uma matriz de inteiros
    // de duas linhas e duas colunas já inicializado.
    inteiro matriz[2][2] = {{15,22},{10,11}}

    //Atribui -1 na primeira linha e segunda
    // coluna da matriz.
    matriz[0][1] = -1

    //Imprime o valor 15 correspondente
    // a primeira linha e primeira coluna da matriz.
    inteiro i = 0
    escreva(matriz[i][0])
    escreva("\n")

    //Imprime o valor 11 correspondente
    // a última linha e última coluna da matriz.
    escreva(matriz[1][1])

    //Declaração de uma matriz de reais de
    // duas linhas e quatro colunas.
    real outra_matriz[2][4]

    //Declaração de uma matriz de caracteres onde o tamanho
    // de linhas e colunas são definidos pela inicialização
    caracter jogo_velha[][] = {{'X','O','X'}
                              ,{'O','X','O'}
                              ,{' ',' ','X'}}

  }
}
```

Declaração de Variáveis

O computador armazena os dados que são utilizados nos programas e algoritmos na memória de trabalho ou memória RAM (Random Access Memory). A memória do computador é sequencial e dividida em posições. Cada posição de memória permite

armazenar uma palavra (conjunto de bytes) de informação e possui um número que indica o seu endereço.

Vamos supor que queremos fazer um programa que solicita para um usuário digitar a sua idade e exibe a ele quantos anos faltam para ele atingir 100 anos de idade. Precisaremos armazenar a idade do usuário para depois realizar o cálculo 100 - idade_usuario e depois armazenar também o resultado.

Para facilitar a nossa vida de programadores, foram criadas as variáveis. As variáveis podem ser entendidas como sendo apelidos para as posições de memória. É através das variáveis que os dados dos nossos programas serão armazenados. A sintaxe para se declarar uma variável é o tipo da variável, o nome da variável ou das variáveis (separadas por virgula cada uma) e opcionalmente pode ser atribuído a ela um valor de inicialização (exceto se for declarado mais de uma na mesma linha)

```
caracter nome_variavel
inteiro variavel_inicializada = 42
real nome_variavel2
logico nome_variavel3
// ou para declarar varias variáveis de um mesmo tipo:
cadeia var1,var2,var3,var4
logico var4,var5,var6
```

É importante ressaltar que o nome de cada variável deve ser explicativo, facilitando assim a compreensão do conteúdo que está armazenado nela.

Para melhor compreensão deste conceito, confira o exemplo abaixo.

```
programa
{
  //variável global do tipo inteiro
  inteiro variavel

  funcao inicio()
  {
    //variável local do tipo inteiro
    inteiro outra_variavel

    //variável local do tipo real já inicializada
    real altura = 1.79

    cadeia frase = "Isso é uma variável do tipo cadeia"

    caracter inicial = 'P'

    logico exemplo = verdadeiro

    //Imprime 1.79, valor obtido na variável altura
    escreva(altura)
  }
}
```

Declaração de Vetores

Armazenar a nota de um aluno é possível utilizando uma variável do tipo real. Mas para armazenar as notas de todos os alunos de uma turma? Seria necessário criar uma variável para cada aluno? E se cada turma tiver quantidade de alunos variáveis? E os nomes de cada um dos alunos? Poderíamos armazenar estes dados em variáveis, porém o controle de muitas variáveis em um programa não é uma solução prática. Ao invés disso, utiliza-se uma estrutura de dados que agrupa todos estes valores em um nome único. Esta estrutura chama-se vetor.

Um vetor pode ser visto como uma variável que possui diversas posições, e com isso armazena diversos valores, porém todos do mesmo tipo.

Assim como as variáveis, o vetor tem que ser declarado. Sua declaração é similar à declaração de variáveis, definindo primeiro o seu tipo, em seguida do seu nome e por fim a sua dimensão entre colchetes (opcional se for atribuir valores a ele na declaração)

```
inteiro vetor[5]
caracter vetor2[200]

//vetores inicializados
real vetor3[2] = {1.4,2.5}
logico vetor4[4] = {verdadeiro,falso,verdadeiro,verdadeiro}
cadeia vetor5[] = {"Questão","Fundamental"}

//Mudando o valor do vetor5 na posição 0 de "Questão" para "Pergunta"
vetor[0] = "Pergunta"
```

Elementos individuais são acessados por sua posição no vetor. Como um vetor tem mais de uma posição, deve-se indicar qual posição do vetor se quer fazer acesso. Para isso é necessário usarmos um índice.

O índice é um valor inteiro que aparece sempre entre colchetes "[ ]" após o nome do vetor. Adotamos que a primeira posição do vetor tem índice zero (similar a linguagem C) e a última depende do tamanho do vetor. Em um vetor de dez elementos tem-se as posições 0,1,2,3,4,5,6,7,8,9. Já um vetor de quatro elementos tem apenas os índices 0,1,2,3.

Para melhor compreensão deste conceito, confira o exemplo abaixo.

```
programa
{
    funcao inicio()
    {
        //Declaração de um vetor de inteiros
        // de cinco posições já inicializado.
        inteiro vetor[5] = {15,22,8,10,11}

        //Imprime o valor 15 correspondente
        // ao primeiro elemento do vetor.
        escreva(vetor[0])
        escreva("\n")

        //Imprime o segundo elemento do vetor
        escreva(vetor[1])
        escreva("\n")

        //Imprime o  valor 11 correspondente
        // ao último elemento do vetor
        escreva(vetor[4])

        //Declaração de um vetor de reais de dez posições
        real outro_vetor[10]

        //Declaração de um vetor de caracteres onde o tamanho
        // é definido pela quantidade de elementos da inicialização
        caracter nome[] = {'P','o','r','t','u','g','o','l'}
    }
}
```

Entrada e Saída

Entrada/saída é um termo utilizado quase que exclusivamente no ramo da computação (ou informática), indicando entrada(inserção) de dados por meio de algum código ou programa,para algum outro programa ou hardware, bem como a sua saída (obtenção de dados) ou retorno de dados, como resultado de alguma operação de algum programa,consequentemente resultado de alguma entrada.

A instrução de entrada de dados possibilita que o algoritmo capture dados provenientes do ambiente externo (fora da máquina) e armazene em variáveis. Assim um algoritmo consegue representar e realizar operações em informações que foram fornecidas por um usuário tais como: nome, idade,salário, sexo, etc. A forma mais comum de capturar dados é através do teclado do computador. Por meio dele o usuário pode digitar números, palavras, frases etc.

A instrução de saída de dados permite ao algoritmo exibir dados na tela do computador. Ela é utilizada para exibir mensagens, resultados de cálculos, informações contidas nas variáveis, etc.

Nesta seção, serão abordados os seguintes tópicos:

- Escreva
- Leia
- Limpa

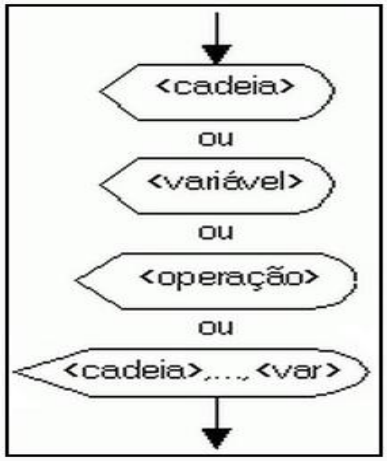
Escreva

Em determinadas situações precisamos mostrar ao usuário do programa alguma informação. Para isso, existe um comando na programação que exibe dados ao usuário. No portugol a instrução de saída de dados para a tela é chamada de “escreva", pois segue a ideia de que o algoritmo está escrevendo dados na tela do computador.

O comando escreva é utilizado quando deseja-se mostrar informações no console da IDE, ou seja, é um comando de saída de dados

.Para utilizar o comando escreva, você deverá escrever este comando e entre parênteses colocar a(s) variável(eis) ou texto que você quer mostrar no console. Lembrando que quando você utilizar textos, o texto deve estar entre aspas. A sintaxe para utilização deste comando está demonstrada a seguir:

O fluxograma abaixo ilustra as diversas formas de se exibir valores na tela com o comando escreva.



Note que quando queremos exibir o valor de alguma variável não utilizamos as aspas. Para exibição de várias mensagens em sequência, basta separá-las com vírgulas.

Existem duas ferramentas importantes que auxiliam a organização e visualização de textos exibidos na tela. São elas:o quebra-linha e a tabulação.

O quebra-linha é utilizado para inserir uma nova linha aos textos digitados. Sem ele, os textos seriam exibidos um ao lado do outro. Para utilizar este comando, basta inserir "\n". O comando de tabulação é utilizado para inserir espaços maiores entre os textos digitados. Para utilizar este comando, basta inserir "\t".

O exemplo a seguir ilustra em portugol o mesmo algoritmo do fluxograma acima, bem como a utilização do quebra-linha e da tabulação.

```
programa
{
  funcao inicio()
  {
    inteiro variavel=5

    //escreve no console um texto qualquer
    escreva ("Escreva um texto aqui.\n")

    //escreve no console o valor da variável "variavel"
    escreva (variavel, "\n")

    //escreve no console o resultado da operação
    escreva (variavel+variavel, "\n")

    //escreve no console o texto digitado, e o valor contido na variável
    escreva ("O valor da variável é: ", variavel)

     //escreve no console o texto com quebra de linha
     escreva ("Texto com\n", "quebra-linha")

     //escreve no console o texto com espaço de tabulação
     escreva ("Texto com\t tabulação")
  }
}
```

Leia

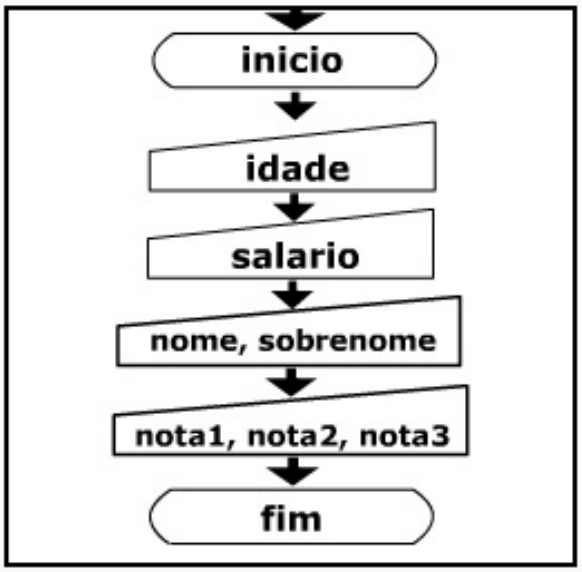
Em alguns problemas, precisamos que o usuário digite um valora ser armazenado. Por exemplo, se quisermos elaborar um algoritmo para calcular a média de nota dos alunos,precisaremos que o usuário informe ao algoritmo quais as suas notas. No portugol a instrução de entrada de dados via teclado é chamada de "leia", pois segue a ideia de que o algoritmo está lendo dados do ambiente externo(usuário) para poder utilizá-los. O Comando leia é utilizado quando se deseja obter informações do teclado do computador, ou seja, é um comando de entrada de dados. Esse comando aguarda um valor a ser digitado e o atribui diretamente na variável.

Para utilizar o comando leia, você deverá escrever este comando e entre parênteses colocar a(s) variável (eis) que você quer que recebam os valores a serem digitados. A sintaxe deste comando está exemplificada a seguir:

```
inteiro x
cadeia y
real z

//chamando o comando leia
leia(x)
leia(y,z)
```

Note que para armazenar um valor em uma variável, é necessário que a mesma já tenha sido declarada anteriormente. Assim como no comando escreva, se quisermos que o usuário entre com dados sucessivos, basta separar as variáveis dentro dos parênteses com vírgula.

O fluxograma abaixo ilustra um algoritmo que lê as variáveis: idade, salário, nome, sobrenome, nota1, nota2 e nota3.



O exemplo a seguir ilustra em portugol o mesmo algoritmo do fluxograma acima.

```
programa
{
  funcao inicio()
  {
     inteiro idade
     real salario, nota1, nota2, nota3
     cadeia nome, sobrenome

     escreva("Informe a sua idade: ")
     leia (idade)              //lê o valor digitado para "idade"

     escreva("Informe seu salario: ")
     leia (salario)            //lê o valor digitado para "salario"

     escreva("Informe o seu nome e sobrenome: ")
     leia (nome, sobrenome)    //lê o valor digitado para "nome" e "sobrenome"

     escreva("Informe as suas três notas: ")
     leia (nota1, nota2, nota3)  //lê o valor digitado para "nota1", "nota2" e "nota3"

     escreva("Seu nome é:"+nome+" "+sobrenome+"\n")
     escreva("Você tem "+idade+" anos e ganha de salario "+salario+"\n")
     escreva("Suas três notas foram:\n")
     escreva("Nota 1: "+nota1+"\n")
     escreva("Nota 2: "+nota2+"\n")
     escreva("Nota 3: "+nota3+"\n")
  }
}
```

Limpa

À medida que um algoritmo está sendo executado ele exibe mensagens e executa ações no console. Assim, em alguns casos o console fica poluído com informações desnecessárias, que atrapalham a compreensão e visualização do programa. Para isso, podemos usar o comando limpa. O comando limpa é responsável por limpar o console. Nãorequer nenhum parâmetro e não tem nenhuma saída. Sua sintaxe é simples, e está demonstrada a seguir:

```
limpa()
```

Para melhor compreensão deste conceito, confira o exemplo abaixo.

```
programa
{

  funcao inicio()
  {
    cadeia nome

    //imprime a frase "Qual é o seu nome?"
    escreva("Qual é o seu nome ?\n")

    //Detecta o que o usuario escreveu na tela
    leia(nome)

    //Limpa tudo que estava escrito no console
    limpa()

    //Escreve resposta
    escreva("Olá "+nome)
  }
}
```

Estruturas de Controle

Em determinadas situações é necessário executar ações de acordo com os dados fornecidos pelo programa. Em alguns casos, pode ser necessário que o programa execute uma determinada instrução repetidas vezes, por exemplo. Sendo assim, controlar e manipular a ordem com que instruções serão

executadas em função de dados fornecidos pelo programa é essencial, e é para isso que servem as estruturas de controle.

Estruturas de controle (ou fluxos de controle) referem-se à ordem em que instruções, expressões e chamadas de função são executadas. Sem o uso de estruturas de controle, o programa seria executado de cima para baixo, instrução por instrução, dificultando assim a resolução de diversos problemas.

Nesta seção, serão abordados os seguintes tópicos:

- Desvios Condicionais
- Laços de Repetição

Desvios Condicionais

Não é só na vida que fazemos escolhas. Nos algoritmos encontramos situações onde um conjunto de instruções deve ser executado caso uma condição seja verdadeira. Por exemplo: sua aprovação na disciplina de algoritmos depende da sua média final ser igual ou superior a 6. Podemos ainda pensar em outra situação: a seleção brasileira de futebol só participa de uma copa do mundo se for classificada nas eliminatórias, se isso não ocorrer ficaremos sem o hexacampeonato.

Estas e outras situações podem ser representadas nos algoritmos por meio de desvios condicionais.

Nesta seção, serão abordados os seguintes tópicos:

- se
- se-senao
- se-senao-se
- escolha-caso

Escolha Caso

Qual a melhor forma para programar um menu de, por exemplo, uma calculadora? Esta tarefa poderia ser executada através de desvios condicionais **se** e **senão**, porém esta solução seria complexa e demorada. Pode-se executar

esta tarefa de uma maneira melhor, através de outro tipo de desvio condicional: o **escolha** junto com o **caso**. Este comando é similar aos comandos **se** e **senão**, e reduz a complexidade do problema.

Apesar de suas similaridades com o **se**, ele possui algumas diferenças. Neste comando não é possível o uso de operadores lógicos, ele apenas trabalha com valores definidos, ou o valor é igual ou diferente. Além disto, o **escolha** e o **caso** tem alguns casos testes, e se a instrução **pare** não for colocada ao fim de*cada um destes testes, o comando executará todos casos*existentes.

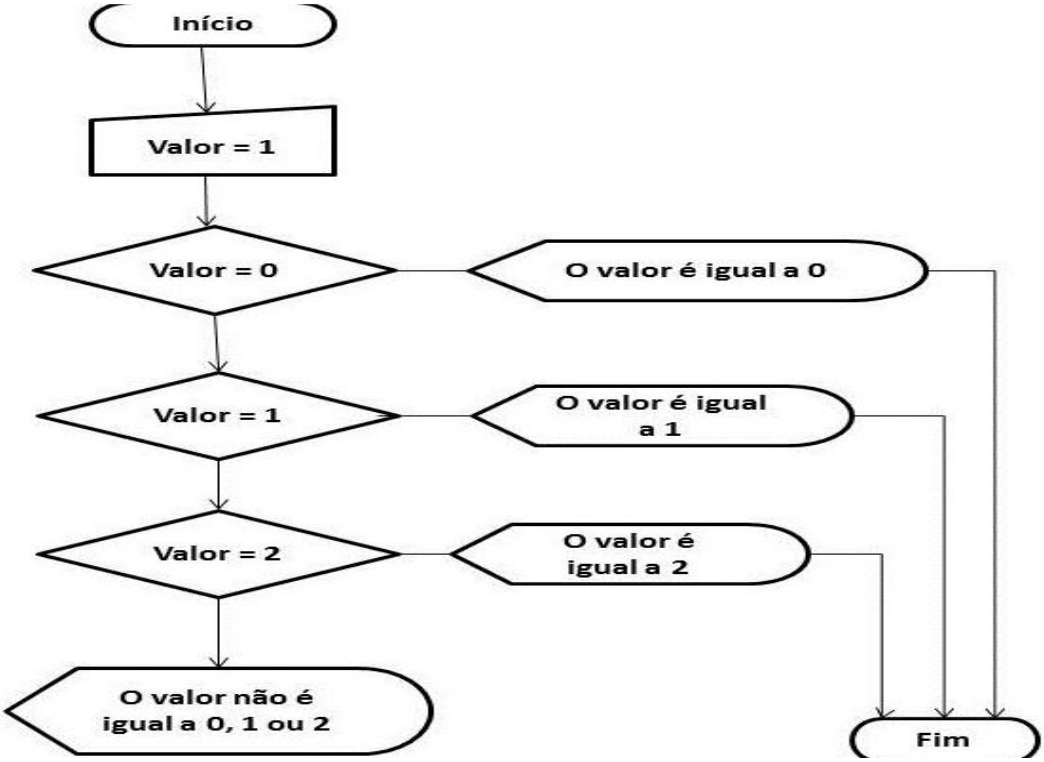
A sintaxe do **escolha** é respectivamente o comando **escolha** a condição a ser testada e entre chaves se coloca os casos.

A sintaxe para se criar um caso é a palavra reservada **caso**, o valor que a condição testada deve possuir dois pontos e suas instruções. Lembre-se de termina-las com o comando **pare.**

```
inteiro numero
escolha(numero)
{
   caso 1:
      //Instruções caso o numero for igual a 1
   pare

   caso 2:
      //Instruções caso o numero for igual a 2
   pare

   caso 50:
      //Instruções caso o numero for igual a 50
   pare

   caso contrario:
      //Instruções caso nenhum dos casos anteriores não seja verdadeiro
}

cadeia texto
escolha(texto)
{
   caso "sim":
      //Instruções caso o texto for igual a "sim"
   pare

   caso "nao":
      //Instruções caso o texto for igual a "nao"
}
```

O comando pare evita que os blocos de comando seguinte sejam executados por engano. O caso contrario será executado caso nenhuma das expressões anteriores sejam executadas.

A figura a seguir ilustra um algoritmo que verifica se o a variável valor é igual a 0, 1 ou 2.



O exemplo a seguir ilustra em portugol o mesmo algoritmo do fluxograma acima.

```
programa
{
    funcao inicio()
    {
    inteiro valor=1
        escolha (valor)
        {
        caso 0:        //testa se o valor é igual a 0
        escreva ("o valor é igual a 0")
        pare

        caso 1:        //testa se o valor é igual a 1
        escreva ("o valor é igual a 1")
        pare

        caso 2:        //testa se o valor é igual a 2
        escreva ("o valor é igual a 2")
        pare

        caso contrario:
        escreva ("o valor não é igual a 0, 1 ou 2")
        }
    }
}
```

Se

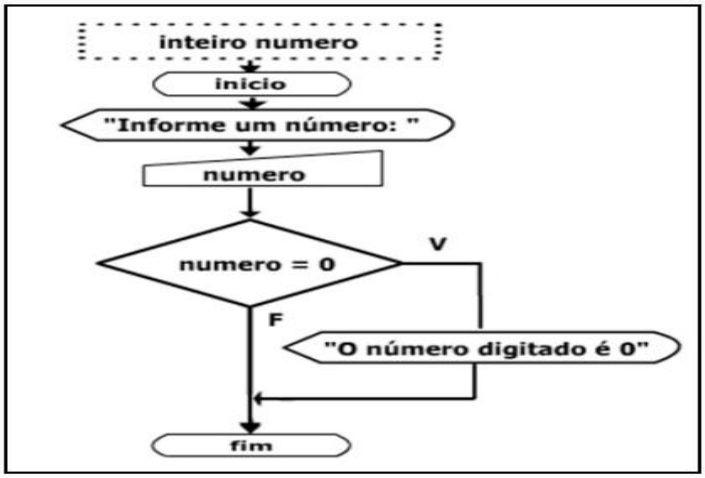
Aqui veremos como dizer a um algoritmo quando um conjunto de instruções deve ser executado. Esta determinação é estabelecida se uma condição for verdadeira. Mas o que seria esta condição? Ao executar um teste lógico teremos como resultado um valor verdadeiro ou falso. A condição descrita anteriormente nada mais é que um teste lógico.

Se este teste lógico resultar verdadeiro, as instruções definidas dentro do desvio condicional serão executadas. Se o teste for falso, o algoritmo pulará o trecho e continuará sua execução a partir do ponto onde o desvio condicional foi finalizado.

O desvio condicional que foi acima apresentado é considerado simples e conhecido como o comando **se**. A sintaxe é respectivamente a palavra reservada **se**, a condição a ser testada entre parênteses e as instruções que devem ser executadas entre chaves caso o desvio seja verdadeiro.

```
logico condicao = verdadeiro
se (condicao)
{
   //Instruções a serem executadas se o desvio for verdadeiro
}

inteiro x = 5
se (x > 3)
{
   //Instruções a serem executadas se o desvio for verdadeiro
}
```

A figura abaixo ilustra um algoritmo que verifica se o número digitado pelo usuário é zero. Ele faz isso usando um desvio condicional. Note que se o teste for verdadeiro exibirá uma mensagem, no caso falso nenhuma ação é realizada.



O exemplo a seguir ilustra em portugol o mesmo algoritmo do fluxograma acima.

```
programa
{
    funcao inicio()
    {

        inteiro num

        escreva ("Digite um número: ")
        leia (num)

        se (num==0)
        {
            escreva ("O número digitado é 0")
        }

     }
}
```

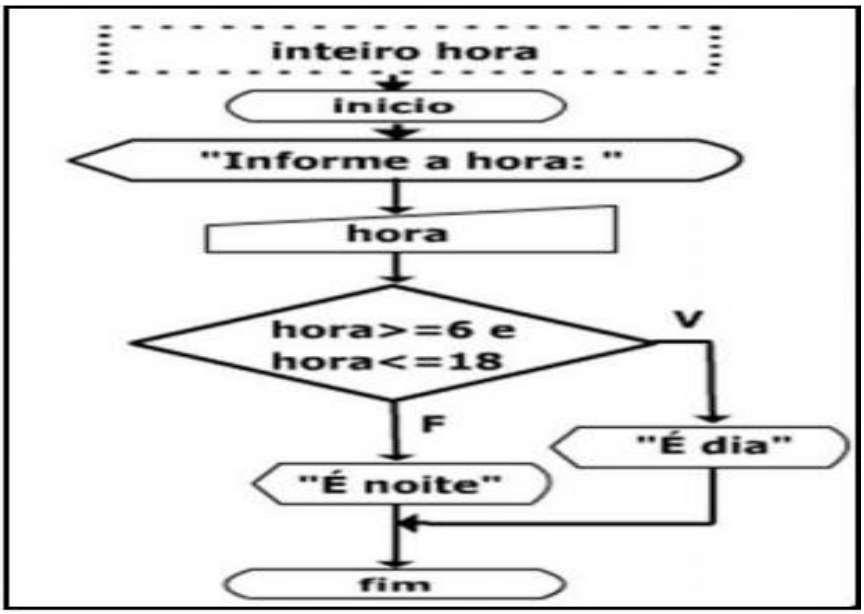
Se-senao

Agora vamos imaginar que se a condição for falsa um outro conjunto de comandos deve ser executado. Quando iremos encontrar esta situação?

Imagine um programa onde um aluno com média final igual ou maior a 6 é aprovado. Se quisermos construir um algoritmo onde após calculada a média, seja mostrada na tela uma mensagem indicando se o aluno foi aprovado ou reprovado. Como fazer isto? Utilizando o comando **se** junto com o **senao**.

Sua sintaxe é simples, basta no termino do comando se ao lado do*fechamento de chaves, colocar o comando **senao** e entre chaves colocar as instruções a serem executadas caso o comando se for falso.

```
logico condicao = falso
se (condicao)
{
   //Instruções a serem executadas se o desvio for verdadeiro
}
senao
{
   //Instruções a serem executadas se o desvio for falso
}
```



O exemplo a seguir ilustra em portugol o mesmo algoritmo do fluxograma acima.

```
programa
{
    funcao inicio()
    {

        inteiro hora

        escreva ("Digite a hora: ")
        leia (hora)

        se (hora >= 6 e hora <= 18)
        {
            escreva ("É dia")
        }
        senao
        {
            escreva ("É noite")
        }

    }
}
```

Se-senao se

Agora imagine que você precise verificar a nota da prova de um aluno e falar se ele foi muito bem, bem, razoável ou mau em uma prova como fazer isto ?

Quando você precisa verificar se uma condição é verdadeira, e se não for, precise verificar se outra condição é verdadeira uma das formas de se fazer*esta verificação é utilizando o **se** ... **senao se**;

A sua sintaxe é parecida com a do **senao**, mas usando o comando **se** imediatamente após escrever o comando **senao**.

```
logico condicao = falso
logico condicao2 = verdadeiro
se (condicao)
{
   //Instruções a serem executadas se o desvio for verdadeiro
}
senao se (condicao2)
{
   //Instruções a serem executadas se o desvio anterior for falso e este desvio for verdadeiro
}
```

Também pode-se colocar o comando **senao** no final do ultimo **senao se**, assim quando todos os testes falharem, ele irá executar as instruções dentro do **senão.**

```
se (12 < 5)
{
   //Instruções a serem executadas se o desvio for verdadeiro
}
senao se ("palavra" == "texto")
{
   //Instruções a serem executadas se o desvio anterior for falso e este desvio for verdadeiro
}
senao
{
   //Instruções a serem executadas se o desvio anterior for falso
}
```

O exemplo a seguir ilustra a resolução do em Portugol de avisar se o aluno foi muito bem, bem, razoável ou mau em uma prova.

```
programa
{
  funcao inicio()
  {
     real nota
     leia(nota)
     se(nota >= 9)
     {
        escreva("O aluno foi um desempenho muito bom na prova")
     }
       senao se (nota >= 7)
       {
          escreva("O aluno teve um desempenho bom na prova")
       }
       senao se (nota >= 6)
       {
          escreva("O aluno teve um desempenho razoável na prova")
       }
       senao
       {
          escreva("O aluno teve um desempenho mau na prova")
       }
    }
}
```

Laços de Repetição

Existem problemas que são repetitivos por natureza. Por exemplo, escrever um algoritmo para calcular a média de um aluno é algo fácil, mas se quisermos calcular a média da turma inteira? A solução mais simples seria executar o algoritmo tantas vezes que o cálculo fosse necessário, embora esta tarefa seja um tanto trabalhosa. Mas se ainda, nesta conta ao final o professor quisesse que fosse mostrada a média mais alta e mais baixa da turma?

Esse e outros problemas podem ser resolvidos com a utilização de laços de repetição. Um laço de repetição, como sugere o próprio nome, é um comando onde uma quantidades de comandos se repete até que uma determinada condição seja verdadeira.

O Portugol contém 3 tipos de laços de repetição: pré-testado, pós-testado e laço com variável de controle.

Nesta seção, serão abordados os seguintes tópicos:

- Enquanto
- Faça-Enquanto
- Para

Laço Enquanto (Pré-Testado)

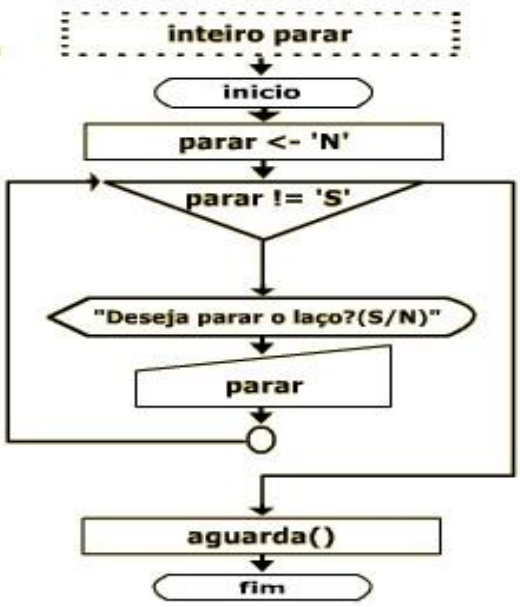
Se fosse necessário a elaboração de um jogo, como por exemplo um jogo da velha, e enquanto houvessem lugares disponíveis no tabuleiro, este jogo devesse continuar, como faríamos para que o algoritmo tivesse este comportamento? É simples. O comando **enquanto** poderia fazer esse teste lógico. A função do comando **enquanto** é: executar uma lista de comandos enquanto uma*determinada condição for verdadeira.

A sintaxe é respectivamente a palavra reservada **enquanto**, a condição a ser testada entre parênteses, e entre chaves a lista de instruções que se deseja executar.

```
logico condicao = verdadeiro
enquanto (condicao)
{
   //Executa a as instruções dentro do laço enquanto a
 condicao for verdadeira
}
```

A figura abaixo ilustra um algoritmo que verifica uma variável do tipo carácter. Enquanto a variável for diferente da letra ‘S’ o comando **enquanto** será executado, assim como as instruções dentro dele. No momento em que o

usuário atribuir ‘S’ a variável, o comando enquanto terminará e o programa chega ao seu final.



O exemplo a seguir ilustra em portugol o mesmo algoritmo do fluxograma acima.

```
programa
{
  funcao inicio()
  {
     caracter parar
     parar = 'N'

     enquanto (parar != 'S')
     {
        escreva ("deseja parar o laço? (S/N)")
        leia (parar)
     }
  }
}
```

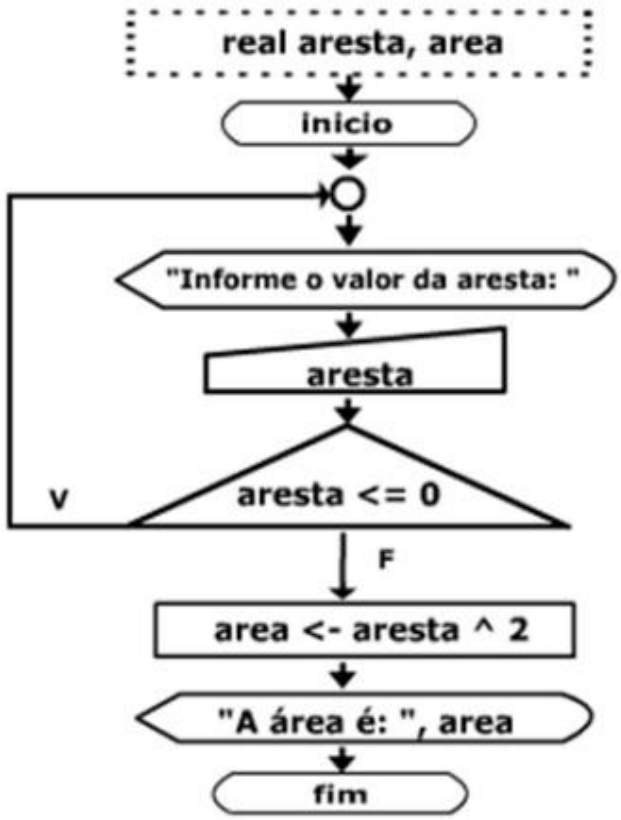
Laço Faça-Enquanto (Pós-Testado)

Em algumas situações, faz-se necessário verificar se uma condição é verdadeira ou não após uma entrada de dados do usuário. Para situações como essa, podemos usar o laço de repetição faca-enquanto. Este teste é bem parecido com o enquanto. A diferença está no fato de que o teste lógico é realizado no final, e com isso as instruções do laço sempre serão realizadas pelo menos uma vez. O teste verifica se elas devem ser repetidas ou não.

A sintaxe é respectivamente a palavra reservada **faca**, entre chaves as instruções a serem executadas, a palavra reservada **enquanto** e entre parênteses a condição a ser testada.

```
logico condicao = verdadeiro
faca
{
   //Executa os comandos pelo menos uma vez, e continua
 executando enquanto a condição for verdadeira
} enquanto (condicao)
```

A figura abaixo ilustra um algoritmo que calcula a área de um quadrado. Note que para o cálculo da área é necessário que o valor digitado pelo usuário para aresta seja maior que 0. Caso o usuário informe um valor menor ou igual a 0 para a aresta, o programa repete o comando pedindo para que o usuário entre novamente com um valor para a aresta. Caso seja um valor válido, o programa continua sua execução normalmente e ao fim exibe a área do quadrado.




```
programa
{
  funcao inicio()
  {
     real aresta, area

     faca
     {
        escreva ("Informe o valor da aresta: ")
        leia (aresta)
     } enquanto (aresta <= 0)

     area=aresta*aresta
     escreva("A área é: ", area)
  }
}
```

Laço Para (Com Variável de Controle)

E se houver um problema em que sejam necessárias um número determinado de repetições? Por exemplo, se quiséssemos pedir ao usuário que digitasse 10 valores. Poderíamos utilizar a instrução Leia repetidas vezes. Porém se ao invés de 10 valores precisássemos de 100, essa tarefa se tornaria muito

extensa. Para resolver problemas como esse, podemos usar um laço de*repetição com variável de controle. No portugol, ele é conhecido como **para.**

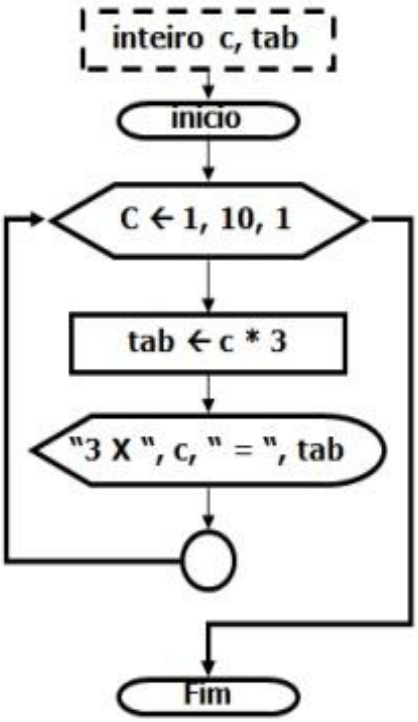
O laço de repetição com variável de controle facilita a construção de algoritmos com número definido de repetições, pois possui um contador (variável de controle) embutido no comando como incremento automático. Desta forma, um erro muito com um que se comete ao esquecer de fazer o incremento do contador é evitado. Toda vez que temos um problema cuja solução necessita de um número determinado de repetições utilizamos um contador. O contador deve ser inicializado antes do laço e deve ser incrementado dentro do laço.

O laço com variável de controle possui três partes. A inicialização da variável contadora, a definição do valor final do contador e adefinição do incremento. Estas três partes são escritas juntas, no início do laço.

A sintaxe é respectivamente a palavra reservada **para**, abre parenteses, a declaração de uma variável de controle, ponto e virgula, a condição a ser testada, ponto e virgula, uma alteração*na variável de controle a ser feita a cada iteração, fecha parenteses, e entre chaves as instruções do programa.

```
para (inteiro i = 0; i < 8; i++)
{
   //Codigo a ser executado enquanto a condição for satisfeita.
}
```

A figura abaixo ilustra um algoritmo que exibe na tela a tabuada de 3. Note que conforme a sintaxe mostrada anteriormente, a primeira instrução do laço é inicializar o contador c=1. O segundo comando especifica a condição para que o laço continue a ser executado, ou seja, enquanto o contador c for menor ou igual a10. Por último, a terceira instrução demonstra que o contador c será acrescentado em 1 em seu valor a cada iteração do comando. O laço será executado 10 vezes e mostrará a tabuada de 3.



O exemplo a seguir ilustra em portugol o mesmo algoritmo do fluxograma acima.

```
programa
{
  funcao inicio()
  {
     inteiro tab

     para (inteiro c=1; c<=10; c++)
     {
        tab=c*3
        escreva ("3 x ", c, " = ", tab, "\n")
     }
  }
}
```

Expressões

Uma expressão em uma linguagem de programação é uma combinação de valores explícitos, constantes, variáveis,operadores e funções que são interpretados de acordo com as regras específicas de precedência e de associação para uma linguagem de programação específica, que calcula e, em seguida,produz um outro valor. Este processo, tal como para as expressões

matemáticas, chama-se avaliação. O valor pode ser de vários tipos, tais como numérico, cadeia, e lógico.

Nesta seção, serão abordados os seguintes tópicos:

- Operação de Atribuição
- Operações Aritméticas
- Operações Bit a Bit
- Operações Lógicas
- Operações Relacionais

Atribuições

Quando criamos uma variável, simplesmente separamos um espaço de memória para um conteúdo. Para especificar esse conteúdo, precisamos de alguma forma determinar um valor para essa variável. Para isso, usamos a operação de atribuição. A instrução de atribuição serve para alterar o valor de uma variável.

Ao fazer isso dizemos que estamos atribuindo um novo valor a esta variável. A atribuição de valores pode ser feita de variadas formas. Pode-se atribuir valores através de constantes,de dados digitados pelo usuário (Leia) ou mesmo através de comparações e operações com outras variáveis já existentes.Neste último caso, após a execução da operação, a variável conterá o valor resultante da operação. O sinal de igual "=" é o símbolo da atribuição no Portugol. A variável a esquerda do sinal de igual recebe o valor das operações que estiverem à direita.

Veja a sintaxe:

```
variavel = 6
variavel = variavel2
variavel = 6 + 4 / variavel2
leia (variavel)
```

Note que uma variável só pode receber atribuições do mesmo tipo que ela. Ou seja, se a variável "b" é do tipo inteiro e a variável "a" é do tipo real, a atribuição não poderá ser realizada.

Existem alguns operandos no Portugol que podem ser utilizados para atribuição de valores. São eles:

```
variavel1 += variavel2    // Equivalente a: variavel1 = variavel1
 + variavel2;
variavel1 -= variavel2    // Equivalente a: variavel1 = variavel1 -
 variavel2;
variavel1 *= variavel2    // Equivalente a: variavel1 = variavel1
 * variavel2;
variavel1 /= variavel2    // Equivalente a: variavel1 = variavel1 /
 variavel2;
variavel1 %= variavel2    // Equivalente a: variavel1 = variavel1
 % variavel2;
variavel1 & variavel2    // Equivalente a: variavel1 = variavel1
 & variavel2;
variavel1 ^= variavel2    // Equivalente a: variavel1 = variavel1
 ^ variavel2;
variavel1 |= variavel2    // Equivalente a: variavel1 = variavel1
 | variavel2;
variavel1++              // Equivalente a: variavel1 = variavel1 +
 1;
variavel1--
```

Para melhor compreensão deste conceito, confira o exemplo abaixo.

```
programa
{
  funcao inicio()
  {
    //Atribuição de valores constantes a uma variável
    inteiro a
    a = 2

    //Atribuição através de entrada de dados, informado pelo
 usuário
    inteiro b
    leia(b)

    //Atribuição através de uma variável já informada pelo
 usuário
    inteiro c
    c = b
  }
}
```

Operações Aritméticas

As operações aritméticas são nossas velhas conhecidas da Matemática. Em algoritmos é muito comum usarmos operadores aritméticos para realizar cálculos.

Os símbolos que usamos para os operadores na Matemática mudam um pouquinho em algoritmos. A multiplicação, que na matemática é um xis 'x' ou um ponto "." torna-se um '*',justamente para não confundir com o xis que pode ser uma variável e com o ponto que pode ser a parte decimal de um número real. A tabela a seguir mostra quais são os operadores que o Portugol utiliza:

Operação	Símbolo	Prioridade
Adição	+	1
Subtração	-	1
Multiplicação	*	2
Divisão	/	2
Resto da divisão inteira	%	2

A prioridade indica qual operação deve ser realizada primeiro quando houverem várias juntas. Quanto maior a prioridade,antes a operação ocorre. Por exemplo:

6 + 7 * 9A multiplicação 7 * 9 é feita antes pois a operação de multiplicação tem prioridade maior que a soma. O resultado deste cálculo será 69.

O uso de parênteses permite modificar a ordem em que as operações são realizadas. Na Matemática existem os parênteses'(', os colchetes '[' e as chaves '{' para indicar as prioridades. Na computação, usa-se somente os parênteses, sendo que os mais internos serão realizados primeiro.Nesta seção, serão abordados os seguintes tópicos:

- Operação de Adição
- Operação de Divisão
- Operação de Módulo
- Operação de Multiplicação
- Operação de Subtração

Operação de Adição

Adição é uma das operações básicas da álgebra. Na sua forma mais simples, adição combina dois números (termos, somando sou parcelas), em um único número, a soma ou total. Adicionar mais números corresponde a repetir a operação.

A sintaxe é bem fácil, se coloca os operandos entre o sinal demais.

```
escreva(1 + 5) //Operação Aritmética 1 + 5 sendo escrita na
 tela

real numero = 50 + 30 //Operação Aritmética 50 + 30 sendo
 armazenada na variável  numero

se(20 + 40 < 70) //Operação Aritmética 20 + 40 dentro de uma
 estrutura de controle "se" em conjunto com uma operação
 relacional "<".
{
   //Comandos
}
```

Note que você poderá atribuir o resultado desta operação a uma variável, ou mesmo executar diretamente através do comando escreva.

Tabela de compatibilidade de tipos da operação de adição

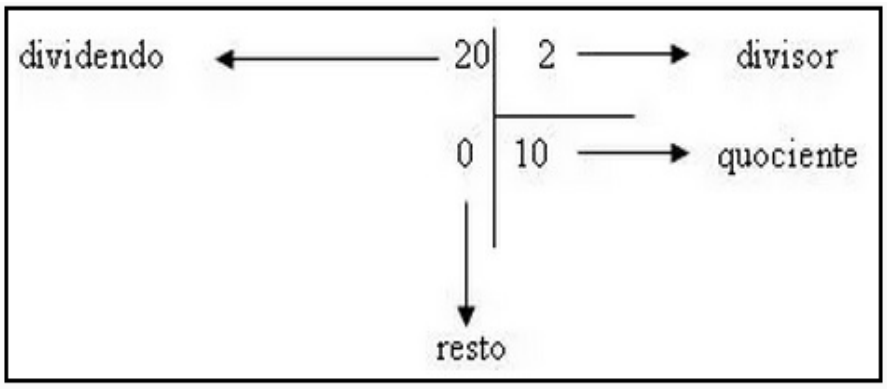
Operando Esquerdo	Operando Direito	Tipo Resultado	Exemplo	Resultado
cadeia	cadeia	cadeia	"Oi" + " mundo"	"Oi mundo"
cadeia	caracter	cadeia	"Banan" + 'a'	"Banana"
cadeia	inteiro	cadeia	"Faz um" + 21	"Faz um 21"
cadeia	real	cadeia	"Altura: " + 1.78	"Altura: 1.78"
cadeia	logico	cadeia	"Help bom =" + verdadeiro	"Help bom = verdadeiro"
caracter	cadeia	cadeia	'P' + "anqueca"	"Panqueca"
caracter	caracter	cadeia	'C' + 'a' + 'd' + 'e' + 'i' + 'a'	"Cadeia"
inteiro	cadeia	cadeia	22 + " de agosto"	"22 de agosto"
inteiro	inteiro	inteiro	12 + 34	46
inteiro	real	real	76 + 3.25	79.25
real	cadeia	cadeia	3.24 + " Kg"	"3.24 Kg"
real	inteiro	real	9.87 + 1	10.87
real	real	real	9.87 + 0.13	10.0
logico	cadeia	cadeia	verdadeiro + " amigo"	"verdadeiro amigo"

Para melhor compreensão deste conceito, confira o exemplo abaixo.

```
programa
{
   funcao inicio()
   {
    inteiro valor

    escreva (5+8, "\n")

    valor = 5+8

    escreva (valor)
   }
}
```

Operação de Divisão

Divisão é a operação matemática inversa da multiplicação. É utilizada para, como o próprio nome sugere, dividir, repartir, separar algum valor em partes iguais. Seus elementos estão demonstrados na figura a seguir:




```
escreva(15 / 5) //Operação Aritmética 1 * 5 sendo escrita na tela

real numero = 50 / 25.6 //Operação Aritmética 50 * 30 sendo armazenada na variável numero
```

Note que você poderá atribuir o resultado desta operação a uma variável, ou mesmo executar diretamente através do comando escreva.

Tabela de compatibilidade de tipos da operação de divisão

Operando Esquerdo	Operando Direito	Tipo Resultado	Exemplo	Resultado
inteiro	inteiro	inteiro	5 / 2	2
inteiro	real	real	125 / 4.5	27.777777
real	inteiro	real	785.4 / 3	261.8
real	real	real	40.351 / 3.12	12.9333333

Para melhor compreensão deste conceito, confira o exemplo abaixo.

```
programa
{
  funcao inicio()
  {
   inteiro valor

   escreva (20/10, "\n")

   valor = 20/10

   escreva (valor)
  }
}
```

Operação de Módulo

Em algumas situações faz-se necessário manipular o resto de algumas divisões. Por exemplo, se você quiser saber se um determinado valor é par ou ímpar, como faria? Para isso podemos utilizar o módulo. A operação módulo encontra o resto da divisão de um número por outro.

Dados dois números a (o dividendo) e b o divisor, a modulo b (a **%** b) é o resto da divisão de a por b. Por exemplo, **7 % 3** seria **1**, enquanto **9 % 3** seria **0.**

```
escreva(13 % 5) //Operação Aritmética 13 % 5 sendo escrita na
tela

real numero = 50 % 4 //Operação Aritmética 50 % 4 sendo
armazenada na variável numero
```

Note que você poderá atribuir o resultado desta operação a uma variável, ou mesmo executar diretamente através do comando escreva.

Tabela de compatibilidade de tipos da operação de módulo

Operando Esquerdo	Operando Direito	Tipo Resultado	Exemplo	Resultado
inteiro	inteiro	inteiro	45 % 7	3

Para melhor compreensão deste conceito, confira o exemplo abaixo.

```
programa
{
  funcao inicio()
  {
   inteiro valor

   escreva (7%3, "\n")

   valor = 7%3

   escreva (valor)
  }
}
```

Operação de Multiplicação

Na sua forma mais simples a multiplicação é uma forma de se adicionar uma quantidade finita de números iguais. O resultado da multiplicação de dois números é chamado produto. Os números sendo multiplicados são chamados de coeficientes ou operandos, e individualmente de multiplicando e multiplicador, conforme figura abaixo:

3	x	7	=	7+7+7	=	21
(multiplicador)		(multiplicando)		3 vezes		(produto)

```
escreva(1 * 5) //Operação Aritmética 1 * 5 sendo escrita na tela

real numero = 50 * 30 //Operação Aritmética 50 * 30 sendo armazenada na variável numero
```

Note que você poderá atribuir o resultado desta operação a uma variável, ou mesmo executar diretamente através do comando escreva.

Propriedades importantes

- **Comutatividade:** A ordem dos fatores não altera o resultado da operação. Assim, se x * y = z, logo y * x = z.

- **Associatividade**: O agrupamento dos fatores não altera o resultado. (Podemos juntar de dois em dois de modo que facilite o cálculo). Assim, se (x * y) * z = w, logo x *(y * z) =w.
- **Distributividade:** Um fator colocado em evidência numa soma dará como produto a soma do produto daquele fator com os demais fatores. Assim, x *(y + z) = (*x * y) +(x * z).
- **Elemento neutro:** O fator 1 (um) não altera o resultado dos demais fatores. O um é chamado "Elemento neutro" da multiplicação. Assim, se x * y = z, logo x * y * 1 = z.(obs:o 0 é o da soma.)
- **Elemento opositor:** O fator -1 (menos um) transforma o produto em seu simétrico. Assim, -1 * x = -x e -1 * y = -y, para y diferente de x.
- **Fechamento:** O produto de dois números reais será sempre um número do conjunto dos números reais.
- **Anulação:** O fator 0 (zero) anula o produto. Assim, x * 0 = 0, e y * 0 = 0, com x diferente de y.

Tabela de compatibilidade de tipos da operação de multiplicação

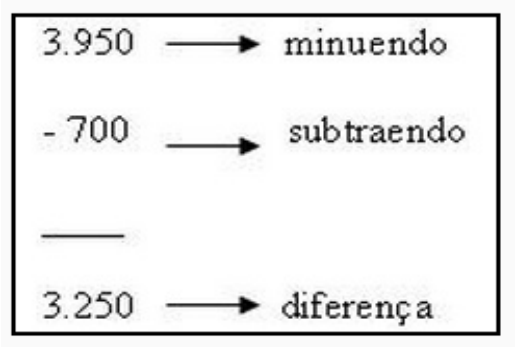
Operando Esquerdo	Operando Direito	Tipo Resultado	Exemplo	Resultado
inteiro	inteiro	inteiro	6 * 8	48
inteiro	real	real	4 * 1.11	4.44
real	inteiro	real	6.712 * 174	1167.888
real	real	real	207.65 * 1.23	255.4095

Para melhor compreensão deste conceito, confira o exemplo abaixo.

```
programa
{
  funcao inicio()
  {
   inteiro valor

   escreva (3*4, "\n")

   valor = 3*4

   escreva (valor)
  }
}
```

Operação de Subtração

Operação de Subtração é uma operação matemática que indica quanto é um valor numérico (minuendo) se dele for removido outro valor numérico (subtraendo).A subtração é o mesmo que a adição por um número de sinal inverso. É, portanto, a operação inversa da adição. Seus elementos estão demonstrados na figura a seguir:




```
escreva(1 - 5) //Operação Aritmética 1 - 5 sendo escrita na tela

real numero = 50 - 30 //Operação Aritmética 50 - 30 sendo armazenada na variável numero
```

Note que você poderá atribuir o resultado desta operação a uma variável, ou mesmo executar diretamente através do comando escreva.

Propriedades importantes

- **Fechamento:** A diferença de dois números reais será sempre um número real.
- **Elemento neutro:** Na subtração não existe um elemento neutro **n** tal que, qualquer que seja o real “a”, a - n = n - a = a.
- **Anulação:** Quando o minuendo é igual ao subtraendo, a diferença será 0 (zero).

Tabela de compatibilidade de tipos da operação de subtração

Operando Esquerdo	Operando Direito	Tipo Resultado	Exemplo	Resultado
inteiro	inteiro	inteiro	20 - 10	10
inteiro	real	real	90 - 0.5	89.5
real	inteiro	real	11.421 - 3	8.421
real	real	real	12.59 - 24.59	-12.0

Para melhor compreensão deste conceito, confira o exemplo abaixo.

```
programa
{
  funcao inicio()
  {
   inteiro valor

   escreva (10-3, "\n")

   valor = 10-3

   escreva (valor)
  }
}
```

Operações Lógicas

As operações lógicas são uma novidade para muitos, pois raramente são vistas na escola. Um operador lógico opera somente valores lógicos, ou seja, é necessário que o valor à esquerda e a direita do operador sejam valores lógicos (verdadeiro ou falso). É muito comum usar expressões relacionais (que dão resultado lógico) e combiná-las usando operadores lógicos. Por exemplo:

Operações	Resultado
5 > 3 **e** 2 < 1	falso
nao (8 < 4)	verdadeiro
1 > 3 **ou** 1 <= 1	verdadeiro

Assim como as operações aritméticas, as operações lógicas também possuem prioridades. Veja a tabela abaixo:

Operador	Prioridade
ou	1
e	2
nao	3

Ou seja, o **nao** tem maior prioridade que todos, e o **ou** tem a menor. Veja os exemplos a seguir:

Passo	Exemplo 1	Exemplo 2
Passo 1	**nao** verdadeiro **ou** falso	verdadeiro **e** falso **ou** verdadeiro
Passo 2	falso **ou** falso	falso **ou** verdadeiro
Passo 3	falso	verdadeiro

Nesta seção, serão abordados os seguintes tópicos:

- **E**
- **OU**
- **NAO**

E

Em algumas situações, necessitamos que alguma instrução só seja executada se outras condições forem verdadeiras. Por exemplo, se você quisesse testar se duas variáveis*distintas têm valor igual a 2, como faria? Para isso podemos utilizar o operador lógico **e**.

Quando usamos o operador **e** o resultado de uma operação lógica será verdadeiro somente quando AMBOS os operandos forem verdadeiros. Ou seja, basta que um deles seja falso e a resposta será falsa. A tabela a seguir é conhecida como tabela verdade e ilustra o comportamento do operador **e**.

Operação 1	Operação 2	Operação 1 e Operação 2
Verdadeiro	Verdadeiro	Verdadeiro
Verdadeiro	Falso	Falso
Falso	Verdadeiro	Falso
Falso	Falso	Falso

Em geral, os operadores lógicos são utilizados em conjunto com as Estruturas de Controle.

```
se (5 > 4 e 6 == 6) //Operação logica 'e' junto com operações relacionais.
{
   //comandos
}

enquanto(verdadeiro e 5 < 4) //Operação logica 'e' junto com operações relacionais e tipo logico.
{
   //comandos
}

logico saida = 5 > 3 e 4 < 5 e 6 < 7
```

Para melhor compreensão deste conceito, confira o exemplo abaixo.

```
programa
{
  funcao inicio()
  {
     //Teste utilizando o operador lógico "e" onde a deve ser igual a 2 e b deve ser igual a 2 também
     inteiro a = 2, b = 2
     se(a == 2 e b == 2)
      {
        escreva("Teste positivo")
     }

     //Neste caso c é igual a 2, entretanto d não é igual a 2, logo este teste não terá como resposta verdadeiro
     inteiro c = 2, d = 3
     se(c == 2 e d == 2)
      {
        escreva("Teste positivo")
     }

     //Neste caso de teste g é igual a 2 e f é diferente de 3, logo este teste terá como resposta verdadeiro
     inteiro g = 2, f = 2
     se(g == 2 e f != 3)
      {
        escreva("Teste positivo")
     }
  }
}
```

OU

Em algumas situações, necessitamos que alguma instrução seja executada se uma entre várias condições forem verdadeiras. Por exemplo, se você quisesse testar se pelo menos uma entre duas variáveis distintas têm valor igual a 2, como faria? Para isso podemos utilizar o operador lógico **ou**.

Quando usamos o operador *ou* o resultado de uma operação lógica será verdadeiro sempre que UM dos operandos for verdadeiro. A tabela verdade a seguir ilustra o*comportamento do operador **ou**.

Operação 1	Operação 2	Operação 1 ou Operação 2
Verdadeiro	Verdadeiro	Verdadeiro
Verdadeiro	Falso	Verdadeiro
Falso	Verdadeiro	Verdadeiro
Falso	Falso	Falso

Em geral, os operadores lógicos são utilizados em conjunto com as Estruturas de Controle.

```
se (5 > 4 ou 7 == 6) //Operação logica 'ou' junto com operações relacionais.
{
   //comandos
}

enquanto(falso ou 5 > 4) //Operação logica 'ou' junto com operações relacionais e tipo logico.
{
   //comandos
}

logico saida = 5 > 8 ou 4 < 12 ou 34 < 7
```

Para melhor compreensão deste conceito, confira o exemplo abaixo.

```
programa
{
  funcao inicio()
  {
    //Teste utilizando o operador lógico "ou" onde a deve ser igual a 2 ou pelo menos b deve ser igual a 2, qualquer um destes satisfaz o teste oferecendo-lhe verdadeiro como
 resposta
    inteiro a = 2, b = 2
    se(a == 2 ou b == 2)
     {
       escreva("Teste positivo")
    }

    //Neste caso c é igual a 2, entretanto d não é igual a 2, mas qualquer uma das condições oferece ao teste como resposta: verdadeiro
    inteiro c = 2, d = 3
    se(c == 2 ou d == 2)
     {
       escreva("Teste positivo")
    }

  }
}
```

NAO

Em algumas situações necessitamos verificar se o contrário de uma sentença é verdadeiro ou não. Por exemplo, se você tem uma variável com um valor falso, e quer fazer um teste que será positivo sempre que essa variável for falsa, como faria? Para isso podemos utilizar o operador lógico **nao**.

O operador **nao** funciona de forma diferente, pois necessita apenas de um operando. Quando usamos o operador **nao**, o valor lógico do operando é invertido, ou seja, o valor*falso torna-se verdadeiro e o verdadeiro torna-se falso. Em geral, os operadores lógicos são utilizados em conjunto com as Estruturas de Controle.

```
se (nao falso) //Operação logica 'nao' junto com operações relacionais.
{
   //comandos
}

enquanto(nao 5 < 4) //Operação logica 'nao' junto com operações relacionais e tipo logico.
{
   //comandos
}

logico saida = nao (5 > 3 e 4 < 5) e 6 < 7
```

Para melhor compreensão deste conceito, confira o exemplo abaixo.

```
programa
{
  funcao inicio()
  {
    //Neste caso de teste a variável teste foi inicializada como falso, e foi verificado se teste não é verdadeiro
    logico teste = falso
    se(nao(teste))
     {
       escreva("Teste positivo")
    }

    //Neste caso teste a soma das variáveis a e b resulta em 5, e comparado se a mesma é maior que 7, entretanto o operador nao, verifica se a+b não são maiores que 7
    inteiro a = 2, b = 3
    se(nao(a+b > 7))
     {
       escreva("Teste positivo")
    }
  }
}
```

Operações Relacionais

Vamos imaginar que você precise verificar se um número digitado pelo usuário é positivo ou negativo. Como poderíamos verificar isto? Através de uma operação relacional. As operações relacionais também são nossas conhecidas da Matemática. Em algoritmos, os operadores relacionais são importantes, pois permitem realizar comparações que terão como resultado um valor lógico (verdadeiro ou falso).

Os símbolos que usamos para os operadores também mudam um pouco em relação ao que usamos no papel. Os símbolos para diferente, maior ou igual e menor ou igual mudam pois não existem nos teclados convencionais. A tabela a seguir mostra todas as operações relacionais e os símbolos que o Portugol utiliza.

Operação	Símbolo
Maior	>
Menor	<
Maior ou igual	>=
Menor ou igual	<=
Igual	==
Diferente	!=

A tabela a seguir apresenta a estrutura de algumas dessas operações.

Operação	Resultado
3 > 4	Falso
7 != 7	Falso
9 == 10 - 1	Verdadeiro
33 <= 100	Verdadeiro
6 >= 5 + 1	Verdadeiro

Nos dois últimos exemplos, temos operadores aritméticos e relacionais juntos. Nestes casos, realiza-se primeiro a operação aritmética e depois a relacional. Em geral, as operações relacionais são utilizadas em conjunto com as Estruturas de Controle. Veja a sintaxe:

```
se (5 > 3) // Estrutura de controle: "se (...)", Operação relacional: "5 > 3"
{
   //conjunto de comandos se for verdadeiro
}

para (i = 0; i < 5; i++) // Estrutura de controle: "para(...)", Operação relacional: "i < 5"
{
   //conjunto de comandos a serem repetidos até a Expressão se tornar falsa
}

faca // Estrutura de controle "faca{}enquanto(...)", Operação relacional: "6 < 2"
{
   //conjunto de comandos a serem repetidos enquanto a condicao for verdadeira após a primeira execução.
} enquanto ( 6 < 2 );
```

Para melhor compreensão deste conceito, confira o exemplo abaixo.

```
programa
{
  funcao inicio()
  {
    //Comparação entre valor A e B utilizando o operador maior que
    inteiro a = 5, b = 3
    se(a > b){
       escreva("A é maior que B")
    }

    //Comparação entre A e B utilizando o operador igual a
    se(a == b){
       escreva("A é igual a B")
    }

    //Comparação entre A e B utilizando o operador maior ou igual a
    se(a >= b){
       escreva("A é maior ou igual a B")
    }

    //Nos testes acima somente o primeiro teste A > B é verdadeiro, deste modo somente esta mensagem será exibida
  }
}
```

Tipos

Quais são os tipos de dados que o computador pode armazenar?

Se pararmos para pensar que tudo que o computador compreende é representado através de Zeros e Uns. Então a resposta é Zero e Um. Certo? Certo! Mas como então o computador pode exibir mensagens na tela, apresentar ambientes gráficos cheios de janelas, compreender o significado das teclas do teclado ou dos cliques do mouse.

Bom tudo começa com a definição de uma série de códigos. Por exemplo. A letra "a" do teclado é representada pela seguinte sequência de zeros e uns "01000001". O número 22 é representado por "00010110". E assim todos os dados que são armazenados pelo computador podem ser representados em zeros e ums.

Sendo assim, existem alguns tipos básicos de dados nos quais valores podem ser armazenados no computador. O Portugol exige que o tipo de dado de um valor seja do mesmo tipo da variável ao qual este valor será atribuído.Nesta seção, serão abordados os seguintes tópicos:

- **Tipo Cadeia**
- **Tipo Caracter**
- **Tipo Inteiro**
- **Tipo Lógico**
- **Tipo Real**
- **Tipo Vazio**

Tipo Cadeia

Em algumas situações precisa-se armazenar em uma variável, um texto ou uma quantidade grande de caracteres. Para armazenar este tipo de conteúdo, utiliza-se uma variável do tipo **cadeia**. Cadeia é uma sequência ordenada de caracteres (símbolos) escolhidos a partir de um conjunto pré-determinado. A sintaxe é a palavra reservada **cadeia** seguida do nome da variável.

```
cadeia nome_da_variavel
```

O valor que essa variável assumirá poderá ser especificado pelo programador, ou solicitado ao usuário (ver Operação de Atribuição). Caso seja especificado pelo programador, o conteúdo deve estar acompanhado de aspas duplas.

Para melhor compreensão deste conceito, confira o exemplo abaixo.

```
programa
{
  funcao inicio()
  {
     cadeia nome1, nome2

     nome1 = "Variável declarada através de atribuição"       //variável declarada através de atribuição do programador

     escreva ("Digite seu nome: ")
     leia (nome2)                                  //variável declarada através de entrada do usuário
     escreva ("\nOlá ", nome2)
  }
}
```

Tipo Caracter

Em determinadas situações faz-se necessário o uso de símbolos, letras ou outro tipo de conteúdo. Por exemplo, em um jogo da velha, seriam necessárias variáveis que tivessem*conteúdos de 'X' e '0'. Para este tipo de situação, existe a variável do tipo **caracter**. A variável do tipo caracter é aquela que contém uma informação composta de apenas UM caracter alfanumérico ou especial. Exemplos de caracteres são letras, números, pontuações e etc.

A sintaxe é a palavra reservada **caracter** e em seguida um nome para variável.

```
caracter nome_da_variavel
```

O valor que essa variável assumirá poderá ser especificado pelo programador ou solicitado ao usuário (ver Operação de Atribuição). Caso seja especificado pelo programador, o conteúdo deve estar acompanhado de aspas simples.

Para melhor compreensão deste conceito, confira o exemplo abaixo.

```
programa
{
  funcao inicio()
  {
     caracter vogal, consoante
     vogal = 'a'                          //variável declarada através de atribuição do programador

     escreva ("Digite uma consoante: ")
     leia (consoante)                      //variável declarada através de entrada do usuário

     escreva ("Vogal: ", vogal, "\n", "Consoante: ", consoante)
  }
}
```

Tipo Inteiro

Em determinadas situações faz-se necessário a utilização de valores inteiros em um algoritmo. Como faríamos, por exemplo, uma simples soma entre dois números pertencentes*ao conjunto dos números inteiros? Simples. Utilizando variáveis do tipo **inteiro**. Uma variável do tipo inteiro pode ser entendida como uma variável que contém qualquer número*que pertença ao conjunto dos números inteiros. Podem ser positivos, negativos ou nulos.

A declaração de uma variável do tipo **inteiro** é simples. A sintaxe é a palavra reservada *inteiro* e em seguida um nome para variável.

```
inteiro nome_da_variavel
```

O valor que essa variável assumirá poderá ser especificado pelo programador ou solicitado ao usuário (ver Operação de Atribuição). Para melhor compreensão deste conceito, confira o exemplo abaixo.

```
programa
{
  funcao inicio()
  {
     inteiro num1, num2
     num1 = 5
     num2 = 3
     escreva (num1 + num2)
  }
}
```

Tipo Lógico

Em determinadas situações faz-se necessário trabalhar com informações do tipo verdadeiro e falso. Este tipo de necessidade aparece muito em operações relacionais para exibir*se determinada condição é verdadeira ou falsa. Por exemplo: como poderíamos verificar se um número digitado pelo usuário é maior que zero? Através de uma variável do tipo **logico**. Uma variável do tipo **logico** é aquela que contém um tipo de dado, usado em operações lógicas, que possui somente dois valores, que são consideradas pelo Portugol*como verdadeiro e falso.

A declaração de uma variável do tipo **logico** é simples. A sintaxe é a palavra reservada **logico** seguida do nome da variável.

```
logico nome_da_variavel
```

O valor que essa variável assumirá poderá ser especificado pelo programador ou solicitado ao usuário (ver Operação de Atribuição). Lembrando que em ambos os casos a variável só assume valores verdadeiro ou falso.

Para melhor compreensão deste conceito, confira o exemplo abaixo.

```
programa
{
  funcao inicio()
  {
     logico teste
     inteiro num

     escreva ("Digite um valor para ser comparado :")
     leia (num)

     teste = (num>0)

     escreva ("O número digitado é maior que zero? ", teste)
  }
}
```

Tipo Real

Em algumas situações é necessário armazenar valores que não pertencem aos números inteiros. Por exemplo, se quiséssemos armazenar o valor da divisão de 8 por 3, como faríamos? Este problema pode ser solucionado com uma variável do tipo **real**. Uma variável do tipo **real** armazena um número real como uma fração decimal possivelmente*infinita, como o número PI 3.1415926535. Os valores do tipo de dado *real* são números separados por pontos e não por vírgulas.

A sintaxe para a declaração é a palavra reservada **real** junto com o nome da variável.

```
real nome_da_variavel
```

O valor que essa variável assumirá poderá ser especificado pelo programador ou solicitado ao usuário (ver Operação de Atribuição). Para melhor compreensão deste conceito, confira o exemplo abaixo.

```
programa
{
  funcao inicio()
  {
    real div

    div = 8.0/3.0

    escreva (div)
  }
}
```

Tipo Vazio

Vazio é usado para o resultado de uma função que retorna normalmente, mas não fornece um valor de resultado ao seu chamado. Normalmente, essas funções de tipo **vazio** são chamadas por seus efeitos colaterais, como a realização de alguma tarefa ou escrevendo os seus parâmetros na saída de dados. A função com o tipo **vazio** termina ou por atingir o final da função ou executando um comando retorne sem valor retornado.

```
programa
{
  funcao inicio()
  {
    imprime_linha()
    informacoes("Portugol",2.0,"UNIVALI")
    imprime_linha()
    informacoes("Java",1.7,"Oracle")
    imprime_linha()
    informacoes("Ruby",2.0,"ruby-lang.org")
    imprime_linha()
     informacoes("Visual Basic",6.0,"Microsoft")
    imprime_linha()

  }

  //Função de retorno vazio que dezenha uma linha no console
  funcao vazio imprime_linha()
  {
    escreva("\n----------------------------------------------------------------------------------------")
  }

  //Função de retorno vazio que formata uma saida com base em seus parâmetros
  funcao vazio informacoes(cadeia nome, real versao, cadeia fornecedor)
  {
    se (nome == "Visual Basic")
    {
      retorne
    }
    escreva("\n")
    escreva("A linguagem ")
    escreva(nome)
    escreva(" encontra-se em sua versão ")
    escreva(versao)
    escreva(" e é fornecida pelo(a) ")
    escreva(fornecedor)
  }
}
```