Lógica de Programação

Módulo 05

Estruturas de Dados, Procedimentos e Funções

Introdução

Vamos iniciar este módulo com um conceito trazido da Wikipédia:

Na ciência da computação, uma ED (Estrutura de Dados) é um modo particular de armazenamento e organização de dados em um computador de modo que possam ser usados eficientemente, facilitando sua busca e modificação. EDs e algoritmos são temas fundamentais da ciência da computação, sendo utilizados nas mais diversas áreas do conhecimento e com os mais diferentes propósitos de aplicação. Sabe-se que algoritmos manipulam dados. Quando estes dados estão organizados (dispostos) de forma coerente, caracterizam uma forma, uma estrutura de dados. A organização e os métodos para manipular essa estrutura é que lhe confere singularidade (e vantagens estratégicas, como a minimização do espaço ocupado na memória RAM), além (potencialmente) de tornar o código-fonte mais enxuto e simples.

Do conceito acima podemos compreender que as Estruturas de Dados vieram para, dentre outras coisas, minimizar o espaço ocupado pela memória do computador e deixar o código do programa mais enxuto e simples. Para compreender como isso ocorre, vamos a um exemplo.

Imagine o seguinte problema: Você precisa criar um algoritmo que lê o nome e as 3 notas de 100 alunos, calcular a média de cada aluno e informar quais foram aprovados e quais foram reprovados. Quantas variáveis você vai precisar para fazer este algoritmo? Vamos iniciar o desenvolvimento dele.

```
Algoritmo notas_100_alunos
Início
  var aluno1, aluno2, aluno3, aluno4, aluno5: literal
  var nota1Aluno1, nota1Aluno2, nota1Aluno3, nota1Aluno4, nota1Aluno5: real
  var nota2Aluno1, nota2Aluno2, nota2Aluno3, nota2Aluno4, nota2Aluno5: real
```

Melhor parar por aqui né. Se para armazenar os nomes de 5 alunos e suas 2 notas

precisamos de 15 variáveis, imagina quantas precisaríamos para armazenar os nomes, as notas e as médias de 100 alunos! E olha que essa é uma escola pequena. E se fossem 5.000 alunos?

Por conta de problemas como esses, surgiu a necessidade de criar uma variável composta. Composta? Isso mesmo. Observe novamente o algoritmo.

```
Algoritmo notas_100_alunos
Início
  var alunos[]: literal
  var notas1[]: real
  var notas2[]: real
  var notas3[]: real
  var medias[]: real
```

Vamos imaginar agora que dá pra guardar todos os nomes dos alunos na variável alunos[]. Como? Digamos que a variável **alunos[]** funciona como um trem, onde cada vagão pode "carregar" um valor.

E qual o tamanho desse trem? O tamanho que definirmos no momento de sua criação. Daí se nós fôssemos começar a guardar os nomes dos alunos, faríamos algo mais ou menos assim:

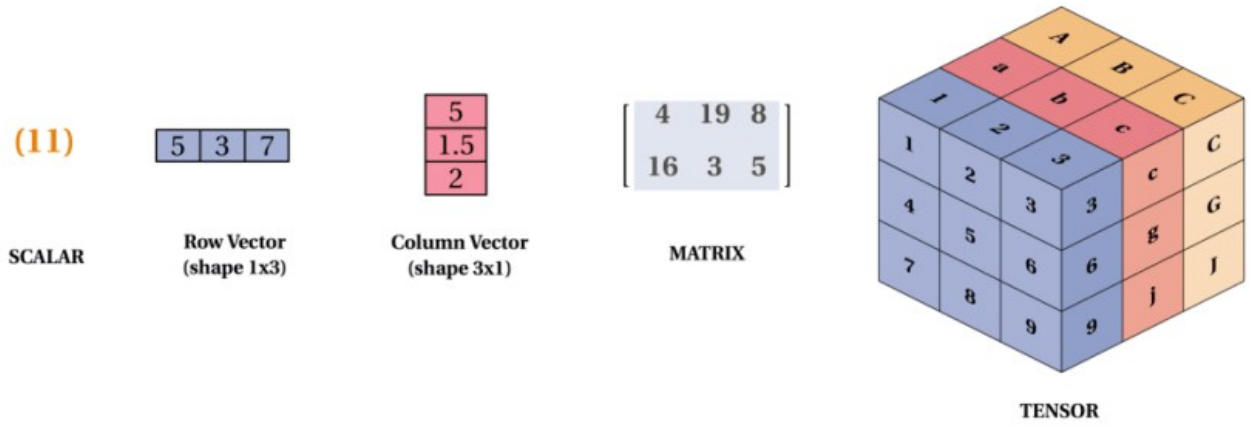
```
  alunos[0] ← "albert"
  alunos[1] ← "joão"
  alunos[2] ← "pedro"
```

Ou seja, em cada posição do nosso "trem" **alunos[]** podemos guardar um valor diferente. Legal né?

Vetores e Matrizes

Este é um dos assuntos mais importantes em termos de lógica de programação. É fundamental que você domine a criação e utilização de vetores e matrizes.

No inglês nós chamamos vetores e matrizes de arrays. Quer dizer que tanto o vetor como a matriz são chamados de arrays? Sim. No entanto, o vetor é um array unidimensional e a matriz é um array com duas dimensões. Observe a imagem abaixo.



- Scalar: um número simples.
- Vetor: um array com uma dimensão.
- Matriz: um array 2D. Pense numa planilha do Excel.
- Tensor: um array ND – várias dimensões.

Tais conceitos vêm da Álgebra Linear e são utilizados no universo da computação para várias finalidades. Para o nosso estudo aqui vamos nos concentrar nos arrays com uma e duas dimensões: vetores e matrizes.

Características Gerais

- Normalmente essas estruturas armazenam variáveis do mesmo tipo. Mas, dependendo da linguagem utilizada, isso pode mudar.
- As posições do vetor são chamadas de índices.
- Normalmente a primeira posição do vetor é acessada pelo índice ZERO. Isso também pode mudar dependendo da linguagem.
- Normalmente você declara um vetor e já informa o seu tamanho, ou seja, o vetor ou matriz devem ter um tamanho estático. Algumas linguagens possuem recursos

para a criação de vetores dinâmicos, onde não é preciso informar o tamanho do vetor no momento de sua declaração.

Vetores na Prática

Vamos para a prática para compreender a utilização dos vetores.

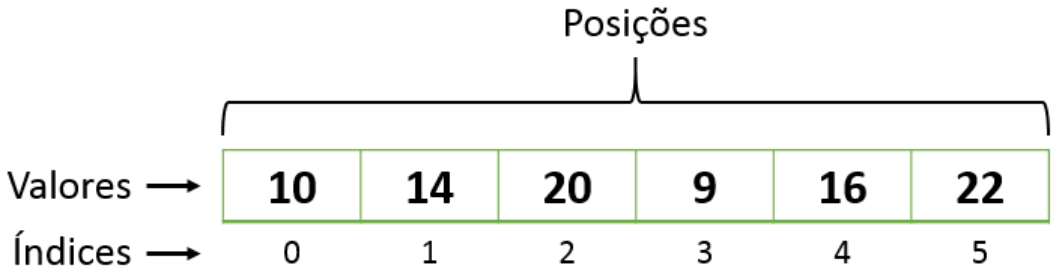
```
algoritmo alunos_na_sala
var
  n: inteiro
  j: inteiro
  valor: inteiro
  idades: vetor[10] de inteiro
inicio
  Escreva("Informe as idades dos 10 alunos")
  n ← idades.tamanho - 1
  para i de 0 até n faca
    leia(valor)
    aluno[i] ← valor
  fim-para
fim-algoritmo
```

Observe atentamente o código. No momento de declarar o vetor, nós informamos o tamanho dele dentro dos colchetes. Em termos de vetores e matrizes, os colchetes têm dois objetivos:

1. No momento da declaração do vetor, o colchete é utilizado para informar o tamanho dele.
2. No momento da utilização do vetor, o colchete é utilizado para acessar um índice do vetor, tanto para ler o valor dentro do índice, quanto para atribuir um valor para ele.

Note ainda que nós informamos o tipo de dado do vetor. No nosso caso, este é um vetor de inteiros, ou seja, ele só pode armazenar valores inteiros em seus índices.

Para preencher o vetor nós utilizamos um laço PARA-FAÇA. Esse laço começou em ZERO e terminou em **n**, que é uma variável que armazena o tamanho do vetor menos 1. Mas porque a variável **n** diminui o tamanho de vetor de 1? Vejamos.



Observe a imagem acima. Temos um vetor com 6 índices. Qual o tamanho desse vetor? SEIS. Mas como eu faço para pegar os valores nos índices?

Índice	Valor
0	10
1	14
2	20
3	9
4	16
5	22

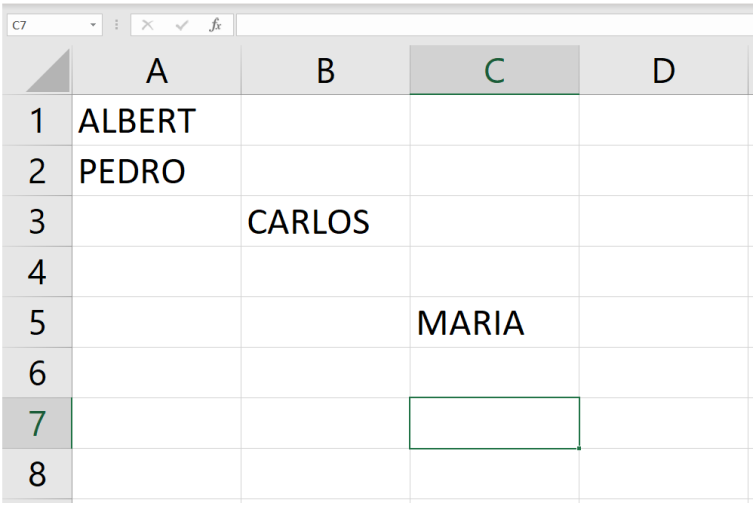
O tamanho do vetor é SEIS, mas para eu pegar o último índice eu preciso utilizar o número CINCO, pois o primeiro índice começa em ZERO. Vamos imaginar que eu quero imprimir esses valores e que, no meu algoritmo, o nome desse vetor é "vetor_imagem". Observe o algoritmo abaixo.

```
inicio
  Escreva("Vamos imprimir os dados do vetor que está na imagem")
  n ← vetor_imagem.tamanho - 1 // preciso diminuir o valor de 1
  para i de 0 até n faca
    escreva(vetor_imagem[i])
  fim-para
fim-algoritmo
```

Para que o laço funcione, eu preciso começar a imprimir os dados do índice ZERO. Eu também preciso informar ao laço até onde ele deve seguir com essa impressão. Como eu quero imprimir todos os elementos do vetor, eu vou usar o tamanho dele como um parâmetro. No entanto, eu não posso terminar o laço usando o tamanho do vetor, pois ele tem SEIS elementos, mas o último elemento está na posição CINCO. É por isso que eu diminuo o tamanho dele em 1.

Matrizes na Prática

Vamos para a prática para compreender a utilização das matrizes. Para facilitar sua visualização mental da matriz, pense nela como uma planilha do Excel.



Quando usamos o Excel, somos apresentados aos conceitos de linhas, colunas e células. Na imagem acima podemos ver que as linhas são acessadas através de números e as colunas através de letras e que a célula é o encontro de uma linha com uma coluna.

Observe que o cursor está na célula C7. O nome CARLOS está na célula B3. É bem simples de se trabalhar dessa maneira.

Quando formos criar uma matriz em nosso algoritmo, porém, sabemos que as linhas vão começar da posição ZERO e que as colunas, no lugar de letras, também serão números iniciados por ZERO. Sabendo dessas informações, consegue dizer em que posição está o nome CARLOS? Ele se encontra na posição (2,1) – linha DOIS, coluna UM. Vamos numerar as células para ficar mais fácil de você visualizar.

	A	B	C	D
1	0,0 [ALBERT]	0,1	0,2	0,3
2	1,0 [PEDRO]	1,1	1,2	1,3
3	2,0	2,1 [CARLOS]	2,2	2,3
4	3,0	3,1	3,2	3,3
5	4,0	4,1	4,2 [MARIA]	4,3
6	5,0	5,1	5,2	5,3

Para não criar confusão na sua cabeça, comece contando as linhas e as colunas a partir do ZERO e tudo ficará claro. Vamos a um exemplo.

```
algoritmo matriz
var
  linhas: inteiro
  colunas: inteiro
  i: inteiro
  j: inteiro
  valor: literal
  matriz: vetor[3, 3] de literal
inicio
  // primeiro preenchemos a matriz
  para i de 0 ate 2 faca
    para j de 1 ate 2 faca
      escreva("Informe o valor para a celula ", i, j)
      leia(valor)
      matriz[i,j] ← valor
    fim-para
  fim-para

  // agora imprimimos os dados da matriz
  para i de 1 ate 2 faca
    para j de 1 ate 2 faca
      escreva(matriz[i,j])
    fim-para
  fim-para
fim-algoritmo
```

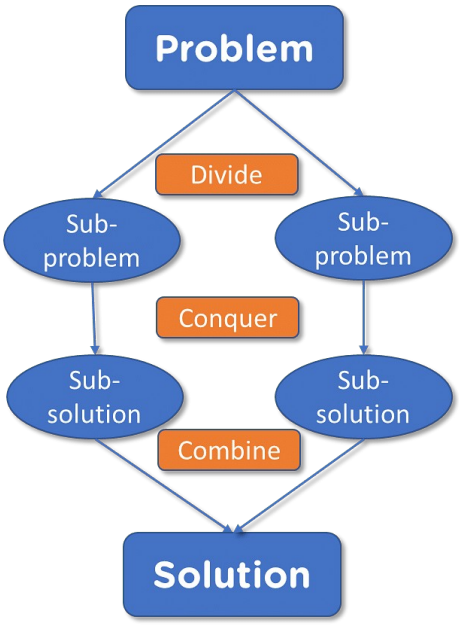
Como fica essa matriz depois de preenchida? Vamos inserir os dados na planilha do Excel para termos uma noção. Para isso, vamos supor que o usuário foi inserindo as letras do alfabeto quando foi solicitado a informar o valor para cada célula.

	A	B	C
1	0,0 [A]	0,1 [B]	0,2 [C]
2	1,0 [D]	1,1 [E]	1,2 [F]
3	2,0 [G]	2,1 [H]	2,2 [I]

Observe que não é tão difícil compreende como os arrays (vetores e matrizes) funcionam. Com bastante prática você dominará o assunto sem muitos problemas.

Procedimentos e Funções

Já ouviu falar do ditado "Dividir para Conquistar"? Você vai utilizar essa estratégia muitas vezes para criar seus programas.



A ideia é que você divida seu problema em problemas menores até que o problema original seja solucionado. E o que isso tem a ver com procedimentos e funções? Vamos explicar com um exemplo. Observe o algoritmo abaixo.

```
algoritmo calculadora
var
  valor_a: real
  valor_b: real
  operação: literal
inicio
  escreva("Informe o primeiro valor: ")
  leia(valor_a)
  escreva("Informe o segundo valor: ")
  leia(valor_b)

  escreva("Operação: [A] [S] [M] [D]") //Adição, Subtração, Multiplicação, Divisão
  leia(operação)

  Escolha (operação);
  Caso "A"
    Escreva "Soma dos valores = (valor_a + valor_b)";
    Pare;
  Caso "S"
    Escreva "Subtração dos valores = (valor_a - valor_b)";
```

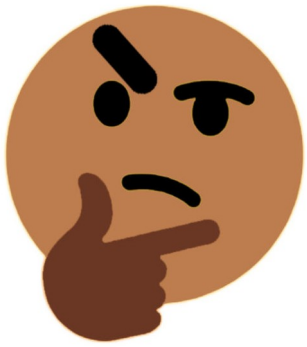
```
    Pare;
  Caso "M"
    Escreva "Multiplicação dos valores = (valor_a * valor_b)";
    Pare;
  Caso "D"
    Escreva "Divisão dos valores = (valor_a / valor_b)";
    Pare;
  Fim-Escolha

fim-algoritmo
```

Este programa realiza as quatro operações básicas para dois números dados. Ele funciona bem. Mas vamos supor que nós queremos, em determinados momentos, apenas somar dois números ou apenas multiplicar dois números. Deveríamos criar um programa só para somar e outro só para multiplicar? E se fosse possível dividir esse nosso algoritmo em outras partes?

Você já percebeu que nós chamamos várias vezes as instruções **escreva("alguma coisa")** e **leia(uma_variavel)**? Como o algoritmo sabe o que deve fazer quando chamamos essas duas intruções?

Quando começamos a dividir nosso programa em outras partes, nós chamamos essas "partes" de procedimentos e funções.



Sim, imaginamos que você faria essa cara. Para facilitar, vamos pegar o mesmo algoritmo acima e dividi-lo, criando um procedimento para a soma.

```
algoritmo calculadora
var
  valor_a: real
  valor_b: real
  operação: literal
inicio
  escreva("Informe o primeiro valor: ")
  leia(valor_a)
```

```
  escreva("Informe o segundo valor: ")
  leia(valor_b)

  escreva("Operação: [A] [S] [M] [D]") //Adição, Subtração, Multiplicação, Divisão
  leia(operação)

  Escolha (operação);
  Caso "A"
    Somar(valor_a, valor_b)
    Pare;
  Caso "S"
    Escreva "Subtração dos valores = (valor_a - valor_b)";
    Pare;
  Caso "M"
    Escreva "Multiplicação dos valores = (valor_a * valor_b)";
    Pare;
  Caso "D"
    Escreva "Divisão dos valores = (valor_a / valor_b)";
    Pare;
  Fim-Escolha

  Procedimento Somar(v1, v2)
  Var
    valor_soma: real
  Inicio
    valor_soma ← v1 + v2
    Escreva "Soma dos Valores = (valor_soma)"
  Fim-Procedimento

fim-algoritmo
```

Observe que dentro do algoritmo nós criamos um procedimento chamado Somar. Esse procedimento (porção de código) espera receber dois valores que são chamados de parâmetros. Dentro do procedimento você pode definir variáveis, que serão utilizadas apenas pelo procedimento. No nosso caso, definimos a variável **valor_soma**. Finalmente, no corpo do procedimento nós realizamos a operação desejada.

Então no corpo do algoritmo nós chamamos esse procedimento através do nome dele e passamos para ele os dois números que o usuário informou.

Procedimentos

Procedimentos são rotinas (trechos ou módulos) de programas, capazes de executar uma tarefa definida pelo programador. Os programas desenvolvidos com procedimentos são ditos 'modulares' e, normalmente, são mais legíveis e melhor estruturados.

Observe a sintaxe de um procedimento.

```
procedimento <nome-de-procedimento> [(<parâmetros>)]
var
  // declaração de variáveis
inicio
  // operações
fim-procedimento
```

Nós inserimos ali uma seção separada para a declaração de variáveis. Esse é um formato parecido com o Pascal/Delphi. Linguagens como Java, C#, JavaScript e outras não precisam de uma seção específica para isso.

Note que um procedimento sempre tem **NOME** e pode ou não ter **PARÂMETROS**. Não existe limite para a quantidade de parâmetros que você pode passar para um procedimento. No entanto, se você observar que está passando parâmetros demais, provavelmente existe um erro de lógica ou então você poderia dividir o procedimento em outros procedimentos.

Escopo

É importante que falemos de escopo neste momento. O que é escopo? É um contexto delimitante aos quais valores e expressões estão associados.

```
algoritmo calculadora
var
  valor_a: real
  valor_b: real
  operação: literal
inicio
  escreva("Informe o primeiro valor: ")
  leia(valor_a)
  escreva("Informe o segundo valor: ")
  leia(valor_b)

  escreva("Operação: [A] [S] [M] [D]") //Adição, Subtração, Multiplicação, Divisão
```

```
  leia(operação)

  Escolha (operação);
  Caso "A"
    Somar(valor_a, valor_b)
    Pare;
  Fim-Escolha

  Procedimento Somar(v1, v2)
  Var
    valor_soma: real
  Inicio
    valor_soma ← v1 + v2
    Escreva "Soma dos Valores = (valor_soma)"
  Fim-Procedimento

fim-algoritmo
```

No programa acima nós temos as variáveis **valor_a** e **valor_b.** Essas variáveis podem ser acessadas de qualquer parte do programa. Dizemos que o escopo delas é **GLOBAL**. Já a variável **valor_soma** que está dentro do procedimento **Somar** não pode ser acessada de qualquer lugar do programa. **Ela só pode ser acessada dentro do procedimento**. Dizemos que o escopo dela é **LOCAL**.

Portugol Webstudio

Veja como fica o nosso algoritmo no Portugol Webstudio:

```
programa
{
  funcao inicio()
  {
    real valor_a
    real valor_b
    cadeia operacao

    escreva("\n")
    escreva("Informe o primeiro valor: ")
    escreva("\n")
    leia(valor_a)
    escreva("\n")
    escreva("Informe o segundo valor: ")
    escreva("\n")
    leia(valor_b)
```

```
    escreva("\n")
    escreva("Operação: [A] [S] [M] [D]: ")
    escreva("\n")
    leia(operacao)

    escolha (operacao)
    {
      caso "A":
        somar(valor_a, valor_b)
        pare
      caso contrario:
        escreva ("Opção Inválida !")
    }
  }

  funcao vazio somar (real v1, real v2)
  {
    real soma
    soma = v1 + v2

    escreva("\n====================")
    escreva("\n Valor da Soma = ")
    escreva(soma)
    escreva("\n====================")
  }

}
```

Observe o resultado no console.

```
Informe o primeiro valor:
45.23

Informe o segundo valor:
23.56

Operação: [A] [S] [M] [D]:
A

====================
 Valor da Soma = 68.78999999999999
====================
Programa finalizado. Tempo de execução: 9346 ms
```

Muito bem, falamos bastante sobre procedimentos e vimos como eles funcionam. Mas o que dizer de funções? Vejamos.

Funções

Funções são rotinas similares aos procedimentos, só que retornam um valor após cada chamada. Uma função não deverá simplesmente ser chamada, como no caso dos procedimentos, mas deverá ou poderá ser atribuída a alguma Variável, visto que a função retornará um valor.

Observe a sintaxe de uma função.

```
função <nome-da-função> [(<parâmetros>)]: <tipo_dado_retorno>
var
  // declaração de variáveis
inicio
  // operações
fim-função
```

A sintaxe é quase a mesma. A diferença está no fato de que, após a lista de parâmetros, deve-se informar o tipo de dado de retorno da função. Vamos pegar o nosso mesmo algoritmo das operações matemáticas e, caso o usuário resolva utilizar a subtração, vamos chamar uma função.

```
algoritmo calculadora
var
  valor_a: real
  valor_b: real
  valor_retorno: real
  operação: literal
inicio
  escreva("Informe o primeiro valor: ")
  leia(valor_a)
  escreva("Informe o segundo valor: ")
  leia(valor_b)

  escreva("Operação: [A] [S] [M] [D]") //Adição, Subtração, Multiplicação, Divisão
  leia(operação)

  Escolha (operação);
  Caso "A"
    Somar(valor_a, valor_b)
    Pare;
  Caso "S"
    valor_retorno = Subtrair(valor_a, valor_b)
    Escreva "Subtração dos valores = (valor_retorno)";
    Pare;
```

```
  Caso "M"
    Escreva "Multiplicação dos valores = (valor_a * valor_b)";
    Pare;
  Caso "D"
    Escreva "Divisão dos valores = (valor_a / valor_b)";
    Pare;
  Fim-Escolha

  Procedimento Somar(v1, v2)
  Var
    valor_soma: real
  Inicio
    valor_soma ← v1 + v2
    Escreva "Soma dos Valores = (valor_soma)"
  Fim-Procedimento

  Função Subtrair(v1, v2): real
  Var
    valor_subtracao: real
  Inicio
    valor_subtracao ← v1 - v2
    retorne valor_subtracao
  Fim-Função

fim-algoritmo
```

No caso do procedimento **Somar**, nós realizamos a operação de soma dentro do procedimento e já imprimimos tudo ali. Já na função **Subtrair** nós realizamos a operação de subtração mas não imprimimos nada, apenas retornamos o valor da subtração. Este valor é atribuído para a variável **valor_retorno** no local onde a função foi chamada. Não é responsabilidade da função Subtrair imprimir o resultado da operação, mas apenas realizá-la e retornar o seu valor.

Portugol Webstudio

Veja como fica o nosso algoritmo no Portugol Webstudio, agora atualizado com a função subtrair:

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

```
  real valor_b
  real valor_retorno
  cadeia operacao

  escreva("\n")
  escreva("Informe o primeiro valor: ")
  escreva("\n")
  leia(valor_a)
  escreva("\n")
  escreva("Informe o segundo valor: ")
  escreva("\n")
  leia(valor_b)

  escreva("\n")
  escreva("Operação: [A] [S] [M] [D]: ")
  escreva("\n")
  leia(operacao)

  escolha (operacao)
  {
    caso "A":
      somar(valor_a, valor_b)
      pare
    caso "S":
      valor_retorno = subtrair(valor_a, valor_b)
      escreva("\n====================")
      escreva("\n Valor da Subtração = ")
      escreva(valor_retorno)
      escreva("\n====================")
      pare
    caso contrario:
      escreva ("Opção Inválida !")
  }
}

funcao vazio somar (real v1, real v2)
{
  real soma
  soma = v1 + v2

  escreva("\n====================")
  escreva("\n Valor da Soma = ")
  escreva(soma)
  escreva("\n====================")
}

funcao real subtrair (real v1, real v2)
{
  real subtracao
  subtracao = v1 - v2
```

```
    retorne subtracao
  }
}
```

Observe o resultado no console.

```
Informe o primeiro valor:
40

Informe o segundo valor:
34

Operação: [A] [S] [M] [D]:
S

==================
Valor da Subtração = 6
==================
Programa finalizado. Tempo de execução: 9760 ms
```

Tudo certo. Você observou que o retorno do tipo de dado da função não é colocado após os parâmetros, como vimos na sintaxe apresentada anteriormente? O tipo de dado de retorno é colocado antes do nome da função.

Além disso, no Portugol Webstudio não existe o termo procedimento. Pois é, tudo lá é função. Para que uma função se comporte como um procedimento, ou seja, para que ela não retorne um valor, usamos o termo **vazio** como retorno, ou seja, uma função que retorna vazio é um procedimento para o Portugol Webstudio.

E assim mesmo vai acontecer quando você for utilizar uma linguagem de programação. Se for usar o Delphi, o comportamento será mais parecido com o que apresentamos na sintaxe. Se for usar o Java o comportamento será mais parecido com o que vimos no Portugol Webstudio.

Dominando a Lógica de Programação você estará apto para implementar seus algoritmos em qualquer linguagem, bastando para isso estudar a sintaxe e particularidades daquela linguagem.