Modularização

Com o avanço do estudo sobre algoritmos os problemas a serem solucionado aumentam em complexidade.

Um método de resolução de problemas complexos é visualizá-los como compostos por problemas menores e tratar um a um os sub-problemas identificados.

Podemos, desta forma, construir um algoritmo composto por subalgoritmos denominados módulos.

Esta metodologia é denominada top-down (de cima para baixo), ou melhor, do genérico para o específico.

UNIVASF

Modularização

Desta forma ao nos depararmos com um problema complexo devemos buscar visualizá-lo como um conjunto de problemas mais simples.

Trataremos agora de como construir um algoritmo composto por módulos.

Assim como um algoritmo, em geral, possui um conjunto de entradas, efetua um processamento e gera um conjunto de saídas. Um módulo também funciona da mesma forma.

UNIVASF

Modularização

Assim como foi definida uma estrutura para representação de algoritmos em pseudocódigo, também é especificada uma estrutura para construção de módulos.

Trabalharemos com dois tipos de módulos:

- a função;
- e o procedimento.

Iniciaremos nossas análise pelo conceito de função.

Uma função é um módulo que possui **ou não** um conjunto de entradas, efetua a execução de um conjunto de instruções e **sempre** gera um valor como saída, também denominado retorno.

Modularização

A estrutura de uma função é a seguinte:

funcao <*nome_da_função*> ([<*sequência-de-declarações-de-parâmetros*>]) : <*tipo_de_dado*>

// Seção de Declarações Internas

inicio

// Seção de Comandos

fimfuncao

Obs.: A declaração de uma função deve estar entre o final da declaração de variáveis e a linha contendo **inicio** no algoritmo principal.

Modularização

O *tipo_de_dado* é o tipo do valor que a função vai retornar.

A *sequência-de-declarações-de-parâmetros* é uma lista com a seguinte forma geral:

identificador1: tipo_de_dado; identificador2: tipo_de_dado; ...; identificadorN: tipo_de_dado

Repare que o *tipo_de_dado* deve ser especificado para cada uma das N variáveis definidas como parâmetros independente de mais de uma variáveis ser do mesmo *tipo_de_dado*. É na declaração de parâmetros que informamos quais serão as entradas da função (assim como informamos a saída no *tipo_de_dado* associado à função).

Modularização

Em um módulo, assim como em um algoritmo, podemos declarar variáveis que serão utilizadas nas manipulações efetuadas. Para esta finalidade existe a **Seção de Declarações Internas**.

As regras para construção do *nome_da_função* são as mesmas aplicadas na definição dos identificadores de variáveis.

Por fim, é na **Seção de Comandos** também denominado corpo da função que as entradas são processadas, saídas são geradas e/ou outras operações são feitas.

UNIVASF

Modularização

- Comando retorne

Forma geral:

retorne valor_de_retorno

Quando se executa um comando **retorne** a função é encerrada imediatamente e o valor de retorno é retornado pela função. É importante lembrar que o valor de retorno fornecido tem que ser compatível com o tipo de retorno declarado para a função.

Modularização

Exemplo de função:

```
algoritmo "exemplo1 função"
var     num:inteiro
        funcao quadrado (a: inteiro): inteiro
        inicio
                retorne (a*a)
        fimfuncao
inicio
  escreva ("Entre com um número inteiro: ")
  leia (num)
  num <- quadrado(num)
  escreva ("O seu quadrado vale: ",num)
fimalgoritmo
```

Modularização

Exemplo de função:

```
algoritmo "exemplo2 função"
var     num:inteiro
        funcao quadrado (a: inteiro): inteiro
        inicio
                retorne (a*a)
        fimfuncao
inicio
  escreva ("Entre com um número: ")
  leia (num)
  escreva ("O quadrado de ",num)
  escreva (" é ", quadrado(num))
fimalgoritmo
```

Modularização

Exercício 32:

Construa uma função capaz de receber um número inteiro como parâmetro e retornar se este é ou não um número impar.

Resposta:

```
funcao eh_impar (a: inteiro): logico
inicio
  retorne (a%2<>0)
fimfuncao
```

Exercício 32:

Resposta alternativa:

```
funcao eh_impar (a: inteiro): caractere
inicio
   se (a%2<>0) entao
      retorne ("O número é impar.")
   senao
      retorne ("O número é par.")
   fimse
fimfuncao
```

Modularização

No caso em que o problema anterior se apresentasse da seguinte forma: Construa uma função que receba um número inteiro e retorne se este é ou não um número impar. Algo mudaria com relação à interpretação?

Sim. Neste caso não estaria especificado se a função receberia o número através de um parâmetro ou se este seria lido através da entrada padrão. Logo, a seguinte solução também seria correta:

```
funcao eh_impar (): logico
var n:inteiro
inicio
   escreva ("Entre com um valor inteiro: ")
   leia (n)
   retorne (n%2<>0)
fimfuncao   //a chamada da função eh_impar deve ser
            //feita da seguinte forma: eh_impar()
```

Exercício 33

Utilizando-se do conceito de modularização construa um algoritmo que resolva o problema de obter as raízes reais de uma equação do segundo grau, caso existam raízes reais.

```
algoritmo "Calcular Raízes"
var
   a, b, c, d: real
   funcao calcular_delta(a:real; b:real; c:real):real
   inicio
      retorne (b^2-4*a*c)
   fimfuncao
inicio
   escreva("Algoritmo que calcula as raízes reais ")
   escreval ("de uma equação do tipo: ax^2+bx+c")
   repita
      escreva ("Entre com o valor de a: ")
      leia (a)
   ate (a<>0)
   escreva ("Entre com o valor de b: ")
   leia (b)
   escreva ("Entre com o valor de c: ")
   leia (c)
   d<-calcular_delta(a,b,c)
   se (d<0) entao
      escreva ("A equação não possui raízes reais.")
   senao
      escreval ("x1 =",(-b+d^0.5)/(2*a))
      escreval ("x2 =",(-b-d^0.5)/(2*a))
   fimse
fimalgoritmo
```

Exercício 34

Construa uma função que receba, como parâmetro, um número inteiro positivo, o qual representa a posição de um determinado termo na série de Fibonacci, a função deve retornar o valor do termo correspondente à posição recebida.

```
funcao fibonacci (p: inteiro):inteiro
var a, b: inteiro
inicio
   a<-0
   b<-1
   enquanto (p>1) faca
        b<-a+b
        a<-b-a
        p<-p-1
   fimenquanto
   retorne a
fimfuncao
```

Modularização

Exercício 35

Construa um algoritmo que seja capaz de efetuar uma multiplicação entre valores naturais quaisquer e também seja capaz de calcular o fatorial de um número natural qualquer. Tanto no cálculo da multiplicação quanto no cálculo do fatorial os únicos operadores aritméticos que podem ser utilizados são os de soma e subtração.

O algoritmo em questão deve possibilitar ao usuário fazer a seleção de qual operação será realizada. As entradas devem ser validadas e o conceito de modularização deve ser aplicado.

UNIVASF