

**INSTITUTO FEDERAL**

Paraná

Campus Assis Chateaubriand

# Algoritmos e Lógica de Programação

## Modularização

### Funções e Procedimentos

# Modularização – Funções e Procedimentos

## Conteúdo

- Modularização de algoritmos;
- Tipos de módulos: funções e procedimentos;
- Escopo de visibilidade (variáveis locais e globais);
- Passagem de parâmetros
  - Passagem de parâmetro por valor;
  - Passagem de parâmetro por referência.

# Modularização – Funções e Procedimentos

## Objetivos de Aprendizagem

1. Compreender o que é modularização de código;
2. Identificar as situações em que a modularização é benéfica;
3. Demonstrar o efeito prático da modularização na redução da complexidade e, portanto, na manutenção do código;
4. Entender como a modularização promove a clareza e a reutilização de código;
5. Compreender o que é o escopo de uma variável e como evitar conflitos de visibilidade de variáveis;
6. Aumentar a generalidade dos módulos através da passagem de parâmetros.

# Modularização de Código – O que é (1)

Um problema complexo frequentemente é resolvido com mais facilidade se for dividido de alguma forma. Embora essas partes ainda constituam problemas que precisam ser resolvidos de alguma forma, certamente serão menores e mais simples de se compreender e resolver que o problema original. Essa é a essência da modularização: decompor um problema grande e complexo em problemas menores e mais simples.

## A diversidade é a mãe da complexidade!

Por razões didáticas, nós aprendemos a programar em ordem de dificuldade crescente e, portanto, os primeiros programas que criamos são relativamente simples: somar dois números, determinar se um número é par, calcular a área de um círculo e assim por diante. Problemas dessa natureza dificilmente são divisíveis em partes menores por serem muito singelos, mas na vida real os problemas tendem a ser mais **complexos**. Embora o conceito de “complexo” seja elástico, em muitos casos a complexidade de um programa comercial é reflexo de sua **diversidade**: quanto mais funcionalidades um programa implementa, maior ele se torna e, com isso, se torna mais *complexo*. Então, quanto maior um programa for (em linhas de código), mais complexo ele se torna.

**Modularizar consiste em dividir um programa em pequenos blocos de código de forma a aumentar a clareza e a legibilidade.** Esses blocos menores são chamados de **módulos**, **sub-rotinas** ou simplesmente **rotinas** e executam tarefas muito específicas dentro do programa, o que facilita a manutenção do código e promove seu reuso (um mesmo módulo pode ser utilizado facilmente em vários programas diferentes).

Um programa existe para resolver um problema. Nesse sentido, a modularização pode ser entendida como uma forma de **decomposição** que transforma *um problema complexo e geral* em *vários problemas simples e específicos*. Então, podemos dizer que **um programa é modular quando é constituído por vários módulos independentes**.

# Modularização de Código – O que é (2)

## Muito além das aulas de algoritmos: como *realmente* funciona uma folha de pagamento

Entre os vários exemplos que empregamos nos nossos estudos de algoritmos, utilizamos um programa para calcular o salário de um empregado. Nesse exemplo, o processamento consistia em multiplicar o número de horas trabalhadas pelo valor de cada hora. Na vida real, um sistema de folha de pagamento é mais complexo e implementa inúmeras funções.

Numa empresa, a folha de pagamento processa todos os fatos relacionados ao trabalho de um empregado, como mostrado ao lado. Isso envolve calcular salário bruto, descontos diversos, salário líquido, bases de cálculo de INSS, FGTS, IRRF, vencimentos, adicionais e comissões, entre muitas outras informações. Implementar todo esse processamento num único bloco de código seria muito ineficiente (ou impossível) porque trata-se de muitas funções distintas entre si. Nesse caso, faz sentido implementar cada uma dessas funções num módulo próprio.

## Algumas Funções Típicas de Uma Folha de Pagamento

1. Salário (fixo ou variável);
2. Adicionais, que são de três tipos:
  - Adicional Noturno;
  - Adicional de Insalubridade;
  - Periculosidade.
3. Comissões (fixas e/ou variáveis);
4. Hora extra;
5. Faltas e Atrasos;
6. Salário Família;
7. Vale transporte;
8. Adiantamento Salarial;
9. Contribuição Sindical;
10. Contribuição Previdenciária;
11. Imposto de Renda.

# Modularização de Código – O que é (3)

## Um Exemplo de um Contracheque de Um Empregado

<b>EBCL</b> Empresa Brasileira de Comércio de Laticínios S/A CNPJ: 78.540.048/0001-80 <u>Demonstrativo de Pagamento</u>					
Func.: Antônio Ademir do Bom Sucesso				Período: 05/2017	
Cargo: Controlador de Estoque Senior			Matrícula: 13221-X	CTPS:	
Depto.: DOPEL - Operações e Logística			Admissão: 01/04/13	CPF:	
Verbas	Referência	Vencimentos		Descontos	
001 - Salário Contratual	30,00	981,88			
002 - Insalubridade	20,00	196,38			
003 - Horas-Extras 50%	5,35	42,98			
002 - Gratificação/Comissão	1,00	975,00			
501 - INSS	9,00			197,66	
502 - IRRF	7,50			7,09	
503 - Consultas/Plano de Saúde	1,00			111,48	
		Total:	2.196,24	Total:	316,23
		Valor Líquido		1.880,00	
Recebi o valor líquido acima em ____/____/____, Assinatura: _____					
Salário Base	Sal. Contr. INSS	Base Cál. FGTS	FGTS do Mês	Base Cál. IRRF	Faixa IRRF
981,88	2.196,24	2.196,24	175,70	1.998,57	7,5%

Ao lado, cada uma das rubricas mostradas no contracheque (coluna "Verbas") pode ser calculada por um módulo separado dos demais e, provavelmente, cada módulo desses será dividido em outros módulos, conforme a conveniência. A ideia é que uma atividade geral (processar a folha de pagamento) seja desempenhada mediante a execução de atividades menores e específicas que, no caso da folha de pagamento, seriam constituídas de módulos para calcular férias, horas extras, encargos sociais, etc).

O cálculo de cada um desses valores pode envolver muitas linhas de código. Não é particularmente sábio fazer todos esses cálculos num único bloco de código!



# Modularização de Código – O que é (4)

Além de compreender o que são módulos e como eles funcionam, é preciso entender como se projeta um programa modularizado. Uma das formas de fazer isso (e, provavelmente, a mais intuitiva) é a abordagem *top-down* (“de cima para baixo” ou “do todo para a parte”).

## *A Abordagem Top-Down*

1. Deve-se compreender qual é a **atividade geral** a ser implementada pelo algoritmo. Essa atividade é executada num bloco de código chamado de **algoritmo principal**;
2. A atividade geral é dividida em subatividades específicas, de escopo bem definido;
3. Cada uma das subatividades deve ser analisada e, se for conveniente, subdividida em outras subatividades;
4. O passo 3 é repetido até que nenhuma subatividade nova seja identificada;
5. Após a identificação de todas as subatividades, elas são transformadas em código. Cada uma delas será implementada como um **módulo**, que poderá ser chamado pelo **algoritmo principal**, por um outro módulo ou pelos dois.

# Modularização de Código – O que é (4)

Na folha de pagamento, a atividade geral é "processar a folha de pagamento", a qual é dividida em subatividades menores e específicas como "calcular adicionais". Muito provavelmente, "calcular adicionais" é uma subatividade muito grande e deve ser dividida em outras subatividades como "calcular horas-extra", "calcular insalubridade", "calcular comissão", "calcular participação nos lucros" e assim por diante.

## *A Abordagem Top-Down*

1. Deve-se compreender qual é a **atividade geral** a ser implementada pelo algoritmo. Essa atividade é executada num bloco de código chamado de **algoritmo principal**;
2. A atividade geral é dividida em subatividades específicas, de escopo bem definido;
3. Cada uma das subatividades deve ser analisada e, se for conveniente, subdividida em outras subatividades;
4. O passo 3 é repetido até que nenhuma subatividade nova seja identificada;
5. Após a identificação de todas as subatividades, elas são transformadas em código. Cada uma delas será implementada como um **módulo**, que poderá ser chamado pelo **algoritmo principal**, por um outro módulo ou pelos dois.



# Modularização – Um Exemplo

**Algoritmo** processaFolha;

**Variáveis**

vlSalBt, vlSalLiq, vlDesc, vlAcres, vlIns, vlHrExt: **real**;  
vlConsulta, vlINSS, vlFGTS, vlIRRF, vlGratificomis : **real**;

⋮

**Início**

**Para** codEmpregado ← 1 **Até** 50 **Faça**

**Início**

  obterDadosEmpregado (codEmpregado);  
  obterNumeroHorasTrab (codEmpregado);  
  calcSalarioBruto (codEmpregado);  
  calcEncargosSociais (codEmpregado);  
  calcDescontos (codEmpregado);  
  calcAdicionais (codEmpregado);  
  imprimeContracheque (codEmpregado);  
  efetuaOrdemPagamento (codEmpregado);

**Fim**;

**Fim.**

# Modularização – Um Exemplo

O bloco de código que é executado em primeiro lugar e que executa os módulos que tenham sido eventualmente criados chama-se de **algoritmo principal**. É comum que o algoritmo principal não contenha comandos, somente invocações de alguns módulos definidos previamente pelo programador.

**Início**

**Para** codEmpregado ← 1 **Até** 50 **Faça**

**Início**

obterDadosEmpregado (codEmpregado);

obterNumeroHorasTrab (codEmpregado);

calcSalarioBruto (codEmpregado);

calcEncargosSociais (codEmpregado);

calcDescontos (codEmpregado);

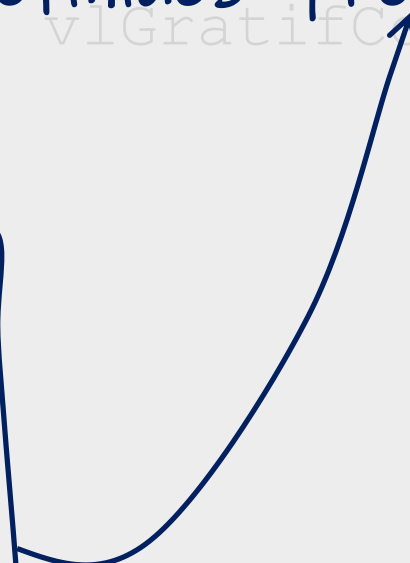
calcAdicionais (codEmpregado);

imprimeContracheque (codEmpregado);

efetuaOrdemPagamento (codEmpregado);

**Fim;**

**Fim.**



# Modularização – Um Exemplo

Algoritmo processaFolha;

## Variáveis

```
vlSalBt, vlSalLiq, vlDesc, vlAcres, vlIns, vlHrExt: real;  
vlConsulta, vlINSS, vlFGTS, vlIRRF, vlGratifComis : real;
```

As variáveis declaradas no programa principal são acessíveis por todos os módulos utilizados no programa. Por essa razão, são chamadas de "variáveis globais", termo que está em oposição a "variáveis locais" (variáveis declaradas num módulo e que não são acessíveis pelo programa principal nem tampouco pelos outros módulos).

Oportunamente, estudaremos que a utilização de variáveis globais nos módulos deve ser evitada tanto quanto possível, pois isso causa dependências indesejáveis (a variável precisa existir para que o módulo execute). Conforme estudaremos, o ideal é utilizar sempre variáveis locais e passagem de parâmetros, que serão estudados oportunamente.

# Modularização – Um Exemplo

As linhas de código abaixo, destacadas com números, consistem em invocação de módulos, e cada um deles é constituído de um bloco de código definido previamente pelo programador. Com a utilização de identificadores sugestivos, fica bem claro para quem lê esse algoritmo que o módulo "1" obtém os dados de um empregado (possivelmente, o que é identificado pelo código numérico armazenado em "codEmpregado"), que o módulo "2" obtém o número de horas trabalhadas por esse empregado e assim por diante.

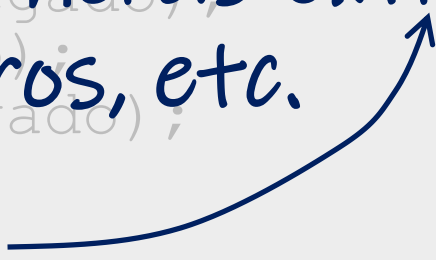
```
① obterDadosEmpregado (codEmpregado) ;  
② obterNumeroHorasTrab (codEmpregado) ;  
③ calcSalarioBruto (codEmpregado) ;  
④ calcEncargosSociais (codEmpregado) ;  
⑤ calcDescontos (codEmpregado) ;  
⑥ calcAdicionais (codEmpregado) ;  
⑦ imprimeContracheque (codEmpregado) ;  
⑧ efetuaOrdemPagamento (codEmpregado) ;  
Fim;  
Fim.
```

Note que essas linhas de código não contém comandos como "leia" e "escreva", mas invocam os 8 módulos numerados à esquerda. Cada um desses módulos é constituído pode executar comandos e/ou invocar outros módulos.

# Modularização – Um Exemplo

Como certos módulos acabam ficando muito grandes, é comum que eles sejam ser divididos em módulos menores. Esse é o caso do módulo "calcAdicionais", cuja finalidade é calcular os adicionais que um empregado deve receber em sua folha. Conforme mencionado anteriormente, devido ao seu tamanho esse módulo deve ser dividido em outros mais específicos para calcular horas-extra, insalubridade, comissões, participação nos lucros, etc.

```
① obterDadosEmpregado (codEmpregado);  
② obterNumeroHorasTrab (codEmpregado);  
③ calcSalarioBruto (codEmpregado);  
④ calcEncargosSociais (codEmpregado);  
⑤ calcDescontos (codEmpregado);  
⑥ calcAdicionais (codEmpregado);  
⑦ imprimeContracheque (codEmpregado);  
⑧ efetuaOrdemPagamento (codEmpregado);  
Fim;  
Fim.
```





# Modularização – Um Exemplo

**Procedimento** calcAdicionais(codEmpregado: **inteiro**);  
**Inicio**

calcHorasExtra(codEmpregado);

calcInsalubridade(codEmpregado);

calcComissoes(codEmpregado);

calcHorasSobreaviso(codEmpregado);

calcGratificacoes(codEmpregado);

calcPLR(codEmpregado);

**Fim.**

# Modularização – Um Exemplo

**Procedimento** calcAdicionais (codEmpregado: **inteiro**);

**Início** "codEmpregado" é uma variável passada como parâmetro nesse módulo.

calcHorasExtra (codEmpregado);

calcInsalubridade (codEmpregado);

calcComissoes (codEmpregado);

calcHorasSobreaviso (codEmpregado);

calcGratificacoes (codEmpregado);

calcPLR (codEmpregado);

**Fim.** Este tipo de módulo ("procedimento") se limita a executar comandos (ou invocar outros módulos) em sequência. Um outro tipo de módulo, chamado de "função", além de executar uma sequência de comandos retorna um valor.

Dentro desse bloco de código, é referenciada como uma variável qualquer, mas não pode ser escrita. Estudaremos passagem de parâmetros oportunamente.

Note que o bloco de código que define esse módulo invoca muitos outros módulos, algo que é muito comum em programas da vida real.

# Por que modularizar? (1)

Modularizar é uma forma de lidar com a complexidade do código, tornando-o mais simples. Um código devidamente modularizado é mais fácil de se compreender e revela um maior padrão de qualidade técnica do software.



Modularizar é uma **boa prática** que auxilia o programador a dominar o código que produz. Pode parecer incrível, mas nem sempre o programador sabe como os seus programas estão funcionando!



# Por que modularizar? (2)

Trivia:

## O Padrão Spaguetti de Programação

O aspecto de um código não-modularizado é difícil de compreender e intimida mesmo o mais experiente dos profissionais. Por ser “enrolado” é conhecido como *código spaguetti*.



“Código Spaghetti” é uma expressão depreciativa para um programa mal redigido, de difícil compreensão e manutenção. Pode ser a consequência de requisitos voláteis mas, frequentemente, é fruto da inexperiência ou falta de um estilo de programação apropriado.

# Procedimentos – Um Exemplo Mais Simples

## Algoritmo para Calcular Áreas

Imagine que você precise criar um algoritmo que seja capaz de calcular a área de diversas figuras geométricas, como um círculos, quadriláteros (quadrados ou retângulos), triângulos e losangos. É claro que é possível criar um algoritmo com uma única estrutura de decisão para fazer isso: basta o usuário informar qual é a figura em questão e o algoritmo executa um bloco de código específico, o qual solicita os dados necessários para determinar a área desejada e a calcula conforme apropriado. No caso de um círculo, por exemplo, o algoritmo deve pedir ao usuário o valor do raio; no caso do quadrilátero, o deve pedir a base e altura e assim por diante.

Contudo, dentro de uma estratégia modular, qual seria a melhor forma de criar esse algoritmo? Qual seria o aspecto do código criado de forma modular?



# Procedimentos – Um Exemplo Mais Simples

```
Algoritmo calculaArea;  
Variáveis  
    vlOpcao : integer; vlArea : real;  
Inicio  
Escreva("(1) Circulo, (2) quadrilátero, (3) triangulo, (4) losango");  
Leia(vlOpcao);  
Se (vlOpcao = 1) então  
    Inicio  
    [código para calcular a área de um circulo]  
    Fim  
Senão Se (vlOpcao = 2) então  
    Inicio  
    [código para calcular a área de um quadrilatero]  
    Fim  
Senão Se (vlOpcao = 3) então  
    Inicio  
    [código para calcular a área de um triangulo]  
    Fim  
Senão  
    Inicio  
    [código para calcular a área de um losango]  
    Fim;  
Escreva("A área da figura eh", vlArea);  
Fim.
```

# Procedimentos – Um Exemplo Mais Simples

```
Algoritmo calculaArea;
```

```
Variáveis
```

```
    vlOpcao : integer; vlArea : real;
```

```
Início
```

```
Escreva("(1) Circulo, (2) quadrilátero, (3) triangulo, (4) losango");
```

```
Leia(vlOpcao);
```

```
Se (vlOpcao = 1) então
```

```
    Início
```

```
    [código para calcular a área de um circulo]
```

```
    Fim
```

```
Senão Se (vlOpcao = 2) então
```

```
    Início
```

```
    [código para calcular a área de um quadrilatero]
```

```
    Fim
```

```
Senão Se (vlOpcao = 3) então
```

```
    Início
```

```
    [código para calcular a área de um triangulo]
```

```
    Fim
```

```
Senão
```

```
    Início
```

```
    [código para calcular a área de um losango]
```

```
    Fim;
```

```
Escreva("A área da figura eh", vlArea);
```

```
Fim.
```

"vlArea" deve armazenar o valor da área

Claramente, é necessário inserir blocos de código nesses locais para que a área seja calculada conforme o tipo de figura. É possível inserir esses blocos diretamente aqui, sem utilizar módulos, mas isso deixaria o código mais extenso e, ademais, o cálculo da área de uma figura não se relaciona em nada ao cálculo da área de outra figura; então, por uma questão de coesão e legibilidade, seria melhor utilizar módulos aqui.

# Procedimentos – Um Exemplo Mais Simples

```
Algoritmo calculaArea;
Variáveis
    vlOpcao : integer; vlArea : real;
Início
Escreva("(1) Circulo, (2) quadrilátero, (3) triangulo, (4) losango");
Leia(vlOpcao);
Se (vlOpcao = 1) então
    Início
        procCalcAreaCirculo;
    Fim
Senão Se (vlOpcao = 2) então
    Início
        procCalcAreaQuadrilatero;
    Fim
Senão Se (vlOpcao = 3) então
    Início
        procCalcAreaTriangulo;
    Fim
Senão
    Início
        procCalcAreaLosango;
    Fim;
Escreva("A área da figura eh", vlArea);
Fim.
```

# Procedimentos – Um Exemplo Mais Simples

```
Algoritmo calculaArea;  
Variáveis  
    vlOpcao : integer; vlArea : real;  
Inicio  
Escreva("(1) Circulo, (2) quadrilátero, (3) triangulo, (4) losango");  
Leia(vlOpcao);  
Se (vlOpcao = 1) então  
    Inicio  
    procCalcAreaCirculo;  
    Fim  
Senão Se (vlOpcao = 2) então  
    Inicio  
    procCalcAreaQuadrilatero;  
    Fim  
Senão Se (vlOpcao = 3) então  
    Inicio  
    procCalcAreaTriangulo;  
    Fim  
Senão  
    Inicio  
    procCalcAreaLosango;  
    Fim;  
Escreva("A área da figura eh", vlArea);  
Fim.
```

Cada um desses procedimentos calcula a área de uma figura diferente. Dentro da lógica deste algoritmo, somente um desses procedimentos será invocado de acordo com o valor da variável "vlOpcao". Cada procedimento é um bloco de código independente dos demais.

# Procedimentos – Um Exemplo Mais Simples

```
Algoritmo calculaArea;  
Variáveis  
    vlOpcao : integer; vlArea : real;  
Início  
Escreva("(1) Circulo, (2) quadrilátero, (3) triangulo, (4) losango");  
Leia(vlOpcao);  
Se (vlOpcao = 1) então  
    Início  
    procCalcAreaCirculo;  
    Fim  
Senão Se (vlOpcao = 2) então  
    Início  
    procCalcAreaQuadrilatero;  
    Fim  
Senão Se (vlOpcao = 3) então  
    Início  
    procCalcAreaTriangulo;  
    Fim  
Senão  
    Início  
    procCalcAreaLosango;  
    Fim;  
Escreva("A área da figura eh", vlArea);  
Fim.
```

As variáveis declaradas aqui, no algoritmo principal, são **variáveis globais** (ou de "escopo global"). São ditas "globais" porque podem ser acessadas por qualquer um dos procedimentos, mas recomenda-se que isso não seja feito, pois cria uma dependência desnecessária (o procedimento que acessa uma variável global só executa em algoritmos que declarem essa variável; o que torna o reuso do procedimento mais difícil).



# Procedimentos – Um Exemplo Mais Simples

```
Procedimento procCalcAreaCirculo;
```

```
Variáveis
```

```
    vlRaio : real;
```

```
Inicio
```

```
Escreva("Entre com o raio do circulo");
```

```
Leia(vlRaio);
```

```
vlArea ← 3.141592654 * (vlRaio^2);
```

```
Fim;
```

```
Procedimento procCalcAreaQuadrilatero;
```

```
Variáveis
```

```
    vlBase, vlAltura : real;
```

```
Inicio
```

```
Escreva("Entre com a base do quadrilatero");
```

```
Leia(vlBase);
```

```
Escreva("Entre com a altura do quadrilatero");
```

```
Leia(vlAltura);
```

```
vlArea ← vlBase * vlAltura;
```

```
Fim;
```

# Procedimentos – Um Exemplo Mais Simples

**Procedimento** procCalcAreaCirculo;

**Variáveis**

vlRaio : real;

→ Variável local

**Início**

**Escreva** ("Entre com o raio do circulo");

**Leia** (vlRaio);

vlArea ← 3.141592654 \* (vlRaio^2);

**Fim;**

→ Variável global (declarada no algoritmo principal)

**Procedimento** procCalcAreaQuadrilatero;

**Variáveis**

vlBase, vlAltura : real;

→ Variáveis locais

**Início**

**Escreva** ("Entre com a base do quadrilatero");

**Leia** (vlBase);

**Escreva** ("Entre com a altura do quadrilatero");

**Leia** (vlAltura);

vlArea ← vlBase \* vlAltura;

**Fim;**

→ Variável global (declarada no algoritmo principal)

As variáveis locais possuem "escopo local"; ou seja, só podem ser utilizadas dentro do procedimento onde foram declaradas.

Os procedimentos podem utilizar as variáveis globais (afinal elas têm "escopo global"), mas isso não é recomendado porque cria dependências desnecessárias.

Como essas variáveis são locais, não podem ser acessadas fora do procedimento onde foram declaradas.

# Procedimentos – Um Exemplo Mais Simples

```
Procedimento procCalcAreaTriangulo;
```

```
Variáveis
```

```
    vlBase, vlAltura : real;
```

```
Inicio
```

```
Escreva("Entre com a base do triangulo");
```

```
Leia(vlBase);
```

```
Escreva("Entre com a altura do triangulo");
```

```
Leia(vlAltura);
```

```
vlArea ← (vlBase * vlAltura)/2;
```

```
Fim;
```

```
Procedimento procCalcAreaLosango;
```

```
Variáveis
```

```
    vlDiag1, vlDiag2 : real;
```

```
Inicio
```

```
Escreva("Entre com a primeira diagonal do losango");
```

```
Leia(vlDiag1);
```

```
Escreva("Entre com a segunda diagonal do losango");
```

```
Leia(vlDiag2);
```

```
vlArea ← (vlDiag1 * vlDiag2)/2;
```

```
Fim;
```

# Procedimentos – Um Exemplo Mais Simples

**Procedimento** procCalcAreaTriangulo;

**Variáveis**

vlBase, vlAltura : **real**;

→ Variáveis locais (ou seja, de escopo local)

**Início**

**Escreva** ("Entre com a base do triangulo");

**Leia** (vlBase);

**Escreva** ("Entre com a altura do triangulo");

**Leia** (vlAltura);

vlArea ← (vlBase \* vlAltura) / 2;

**Fim**;

→ Variável global (ou de "escopo global", declarada

**Procedimento** procCalcAreaLosango;

**Variáveis**

vlDiag1, vlDiag2 : **real**;

→ Variáveis locais

no algoritmo principal)

**Início**

**Escreva** ("Entre com a primeira diagonal do losango");

**Leia** (vlDiag1);

**Escreva** ("Entre com a segunda diagonal do losango");

**Leia** (vlDiag2);

vlArea ← (vlDiag1 \* vlDiag2) / 2;

**Fim**;

→ Variável global (declarada no algoritmo principal)

# Procedimentos em Pascal: as *procedures*

```
procedure procCalcAreaCirculo;
Var
    vlRaio : real;
begin
    writeln('Entre com o raio do circulo');
    readln(vlRaio);
    vlArea := 3.141592654 * sqr(vlRaio);
end;

procedure procCalcAreaQuadrilatero;
Var
    vlBase, vlAltura : real;
begin
    writeln('Entre com a base do quadrilatero');
    readln(vlBase);
    writeln('Entre com a altura do quadrilatero');
    readln(vlAltura);
    vlArea := vlBase * vlAltura;
end;
```



# Procedimentos em Pascal: as *procedures*

```
procedure procCalcAreaTriangulo;
Var
    vlBase, vlAltura : real;
begin
    writeln('Entre com a base do triangulo');
    readln(vlBase);
    writeln('Entre com a altura do triangulo');
    readln(vlAltura);
    vlArea := (vlBase * vlAltura)/2;
end;

procedure procCalcAreaLosango;
Var
    vlDiag1, vlDiag2 : real;
begin
    writeln('Entre com a primeira diagonal do losango');
    readln(vlDiag1);
    writeln('Entre com a segunda diagonal do losango');
    readln(vlDiag2);
    vlArea := (vlDiag1 * vlDiag2)/2;
end;
```

# Procedimentos em Pascal: usando as *procedures*

```
program calculaArea;
Var
    vlOpcao : integer; vlArea : real;
Begin
    writeln(' (1) Circulo, (2) quadrilátero, (3) triangulo, (4) losango');
    readln(vlOpcao);
    if (vlOpcao = 1) then
        begin
            procCalcAreaCirculo;
        end
    else if (vlOpcao = 2) then
        begin
            procCalcAreaQuadrilatero;
        end
    else if (vlOpcao = 3) then
        begin
            procCalcAreaTriangulo;
        end
    else
        begin
            procCalcAreaLosango;
        end;
    writeln('A área da figura eh ', vlArea);
end.
```

# Funções: módulos que retornam valores (1)

Assim como os procedimentos, as funções são módulos de um programa. Contudo, ao contrário dos procedimentos, as funções retornam um valor.

## Funções

A palavra “função” tem origem na matemática, e é utilizada na informática com uma conotação parecida: um conjunto de operações que transforma certos parâmetros num valor.

Assim como os procedimentos, as **funções** são um tipo de módulo. Ambos executam uma sequência de comandos, mas as **funções** produzem um valor que pode ser lido pelo programa (ou seja, “retornam um valor”). Por essa razão, *as funções podem ser usadas em expressões lógicas e aritméticas de maneira similar a uma variável*. Então, se uma variável retornar um valor inteiro, esse valor poderia ser multiplicado por 5 e armazenado numa variável inteira, por exemplo.

## Exemplos de funções matemáticas

$$x = \sqrt{144} \rightarrow \text{Função Raiz}$$

$$y = x^2 \rightarrow \text{Função Quadrado}$$

$$y = |w| \rightarrow \text{Função Módulo}$$

$$y = \log_{10}(x) \rightarrow \text{Função Logaritmo}$$

## Exemplos de funções comumente encontradas em algoritmos

raiz( ), potencia( ), abs( ), log( )

↓      ↓      ↓      ↓  
raiz   potência   módulo   logaritmo

# Funções: módulos que retornam valores (2)

## Usando Funções no Algoritmo para Calcular Áreas

O algoritmo que apresentamos para calcular a área de figuras utilizava uma estratégia modular, onde a área de cada tipo de figura era calculada por um procedimento diferente (*procCalcAreaCirculo*, *procCalcAreaQuadrilatero*, *procCalcAreaTriangulo* e *procCalcAreaLosango*). Para armazenar o resultado, todos os procedimentos utilizavam uma mesma variável global chamada de “*vlArea*”. Como vimos, essa abordagem não é interessante porque cria uma dependência: *os procedimentos mostrados só podem ser utilizados em algoritmos que utilizem uma variável global com esse mesmo nome e tipo.*

O ideal é evitar o uso de variáveis globais, e isso pode ser feito de duas formas diferentes: **passagem de parâmetros por referência** ou utilizando **funções** ao invés de procedimentos. Nesse momento, trabalharemos com funções; posteriormente, trabalharemos com passagem de parâmetros por referência.

# Funções: módulos que retornam valores (3)

```
Algoritmo calculaArea;  
Variáveis  
    vlOpcao : integer; vlArea : real;  
Início  
Escreva("(1) Circulo, (2) quadrilátero, (3) triangulo, (4) losango");  
Leia(vlOpcao);  
Se (vlOpcao = 1) então  
    Início  
        vlArea ← calcAreaCirculo;  
    Fim  
Senão Se (vlOpcao = 2) então  
    Início  
        vlArea ← calcAreaQuadrilatero;  
    Fim  
Senão Se (vlOpcao = 3) então  
    Início  
        vlArea ← calcAreaTriangulo;  
    Fim  
Senão  
    Início  
        vlArea ← calcAreaLosango;  
    Fim;  
Escreva("A área da figura eh", vlArea);  
Fim.
```



# Funções: módulos que retornam valores (3)

**Algoritmo** calculaArea;

**Variáveis**

vlOpcao : integer; vlArea : real;

**Início**

**Escreva** (" (1) Circulo, (2) quadrilátero, (3) triangulo, (4) losango");

**Leia** (vlOpcao);

**Se** (vlOpcao = 1) **então**

**Início**

vlArea ← **calculaAreaCirculo**;

**Fim**

**Senão Se** (vlOpcao = 2) **então**

**Início**

vlArea ← **calculaAreaQuadrilatero**;

**Fim**

**Senão Se** (vlOpcao = 3) **então**

**Início**

vlArea ← **calculaAreaTriangulo**;

**Fim**

**Senão**

**Início**

vlArea ← **calculaAreaLosango**;

**Fim**;

**Escreva** ("A área da figura eh", vlArea);

**Fim.**

→ Variáveis globais

→ Função que retorna um valor real  
(a área de um círculo)

→ Função que retorna um valor real  
(a área de um quadrilátero)

→ Função que retorna um valor real  
(a área de um triângulo)

→ Função que retorna um valor real  
(a área de um losango)

→ Note que a área calculada  
é sempre armazenada na  
variável "vlArea"

# Funções: módulos que retornam valores (4)

```
Funcao calcAreaCirculo: real;
```

```
Variáveis
```

```
    vlRaio : real;
```

```
Inicio
```

```
Escreva("Entre com o raio do circulo");
```

```
Leia(vlRaio);
```

```
calcAreaCirculo  $\leftarrow$  3.141592654 * (vlRaio^2);
```

```
Fim;
```

```
Funcao calcAreaQuadrilatero: real;
```

```
Variáveis
```

```
    vlBase, vlAltura : real;
```

```
Inicio
```

```
Escreva("Entre com a base do quadrilatero");
```

```
Leia(vlBase);
```

```
Escreva("Entre com a altura do quadrilatero");
```

```
Leia(vlAltura);
```

```
calcAreaQuadrilatero  $\leftarrow$  vlBase * vlAltura;
```

```
Fim;
```

# Funções: módulos que retornam valores (4)

**Funcao** calcAreaCirculo: **real;** → Tipo de retorno da função

**Variáveis**

**vlRaio : real;** → Variável local "calcAreaCirculo"

**Inicio**

**Escreva** ("Entre com o raio do circulo");

**Leia** (vlRaio);

calcAreaCirculo ← 3.141592654 \* (vlRaio^2);

**Fim;**

→ Retorno da função (área do círculo)

**Funcao** calcAreaQuadrilatero: **real;** → Tipo de retorno da função

**Variáveis**

**vlBase, vlAltura : real;** → Variáveis locais "calcAreaQuadrilatero"

**Inicio**

**Escreva** ("Entre com a base do quadrilatero");

**Leia** (vlBase);

**Escreva** ("Entre com a altura do quadrilatero");

**Leia** (vlAltura);

calcAreaQuadrilatero ← vlBase \* vlAltura;

**Fim;**

→ Retorno da função (área do quadrilátero)

# Funções: módulos que retornam valores (5)

```
Funcao calcAreaTriangulo: real;
```

```
Variáveis
```

```
    vlBase, vlAltura : real;
```

```
Inicio
```

```
Escreva("Entre com a base do triangulo");
```

```
Leia(vlBase);
```

```
Escreva("Entre com a altura do triangulo");
```

```
Leia(vlAltura);
```

```
calcAreaTriangulo ← (vlDiag1 * vlDiag2)/2;
```

```
Fim;
```

```
Funcao calcAreaLosango: real;
```

```
Variáveis
```

```
    vlDiag1, vlDiag2 : real;
```

```
Inicio
```

```
Escreva("Entre com a primeira diagonal do losango");
```

```
Leia(vlDiag1);
```

```
Escreva("Entre com a segunda diagonal do losango");
```

```
Leia(vlDiag2);
```

```
calcAreaLosango ← (vlDiag1 * vlDiag2)/2;
```

```
Fim;
```

# Funções: módulos que retornam valores (5)

```
Funcao calcAreaTriangulo: real;  
Variáveis  
    vlBase, vlAltura : real;  
Início  
Escreva ("Entre com a base do triangulo");  
Leia (vlBase);  
Escreva ("Entre com a altura do triangulo");  
Leia (vlAltura);  
calcAreaTriangulo ← (vlDiag1 * vlDiag2) / 2;  
Fim;
```

→ Tipo de retorno da função "calcAreaTriangulo"

→ Variável local "calcAreaTriangulo"

Retorno da função (área do círculo)

```
Funcao calcAreaLosango: real;  
Variáveis  
    vlDiag1, vlDiag2 : real;  
Início  
Escreva ("Entre com a primeira diagonal do losango");  
Leia (vlDiag1);  
Escreva ("Entre com a segunda diagonal do losango");  
Leia (vlDiag2);  
calcAreaLosango ← (vlDiag1 * vlDiag2) / 2;  
Fim;
```

→ Tipo de retorno da função "calcAreaLosango"

→ Variáveis locais

Retorno da função (área do losango)



# Passagem de Parâmetros por Valor (1)

Um argumento é um valor que é informado a um módulo quando ele é invocado no código. Um parâmetro é uma variável que armazena um argumento e faz parte da definição do módulo.

Informando Valores para um Módulo:

## Passagem de Parâmetros

Muitas vezes, é necessário informar um ou mais valores a um módulo, chamados de **argumentos**. Para isso, o módulo deve ser definido com o auxílio de uma **variável de parâmetro** (ou simplesmente **parâmetro**). Existem dois tipos de parâmetros, os **passados por valor** e os **passados por referência**. No código, ambos são utilizados como variáveis locais, mas somente os últimos podem ter seu valor alterado pelo módulo. Como veremos posteriormente, *enquanto as alterações efetuadas nos parâmetros passados por referência afetam o valor do argumento, as efetuadas nos parâmetros passados por valor não persistem.*

```
Funcao dobraNum (x:inteiro): inteiro;
```

**Variáveis**

```
Y      : inteiro;
```

**Inicio**

```
Y ← 2 * x;
```

```
dobraNum ← y;
```

**Fim;**

A função *dobraNumero* recebe como argumento um valor numérico inteiro e o duplica. Note que a declaração dessa função (sua **assinatura**) inclui um **parâmetro** inteiro passador por valor (a variável “x”).

# Passagem de Parâmetros por Valor (1)

Um argumento é um valor que é informado a um módulo quando ele é invocado no código. Um parâmetro é uma variável que armazena um argumento e faz parte da definição do módulo.

## Informando Valores para um Módulo: Passagem de Parâmetros

Muitas vezes, é necessário informar um ou mais valores a um módulo, chamados de **argumentos**. Para isso, o módulo deve ser definido com o auxílio de uma **variável de parâmetro** (ou simplesmente **parâmetro**). Existem dois tipos de parâmetros, os **passados por valor** e os **passados por referência**. No código, ambos são utilizados como variáveis locais, mas somente os últimos podem ter seu valor alterado pelo módulo. Como veremos posteriormente, *enquanto as alterações efetuadas nos parâmetros passados por referência afetam o valor do argumento, as efetuadas nos parâmetros passados por valor não persistem.*

```
Funcao dobraNum (x:inteiro): inteiro;
```

**Variáveis** *Parâmetro passado por valor*  
y : inteiro;

**Início** *Note que o parâmetro é manipulado como se fosse uma variável local qualquer.*  
y ← 2 \* x;

```
dobraNum ← y;
```

```
Fim;
```

A função *dobraNumero* recebe como argumento um valor numérico inteiro e o duplica. Note que a declaração dessa função (sua **assinatura**) inclui um **parâmetro** inteiro passador por valor (a variável “x”).

# Passagem de Parâmetros por Valor (2)

**Algoritmo** testaDobra;

**Variáveis**

vlNum, vlDobroNum : inteiro;

**Funcao** dobraNum(x : inteiro) : inteiro;

**Variáveis**

y : inteiro;

**Início**

y ← 2 \* x;

dobraNum ← y;

**Fim;**

**Início**

**Escreva** ("Entre com um número") ;

**Leia** (vlNum) ;

vlDobroNum ← dobraNum(vlNum) ;

**Escreva** ("O dobro de ", vlNum, " eh ", vlDobroNum) ;

**Fim.**

# Passagem de Parâmetros por Valor (2)

**Algoritmo** testaDobra;

**Variáveis**

vlNum, vlDobroNum : inteiro;

**Variáveis globais:** são utilizadas no algoritmo principal e também nos módulos desse algoritmo.

→ **Algoritmo principal:** contém o código que é executado

**Início** em primeiro lugar.

**Escreva** ("Entre com um número");

**Leia** (vlNum);

vlDobroNum ← dobroNum(vlNum);

**Escreva** ("O dobro de ", vlNum, " eh ", vlDobroNum);

**Fim.**

→ Invocação da função "dobroNum"

→ O valor de "vlNum" é o argumento passado à função "dobroNum"

# Passagem de Parâmetros por Valor (2)

## Declaração da função "dobraNum"

```
vlNum, vlDobroNum : inteiro;
```

```
Funcao dobraNum (x : inteiro) : inteiro;
```

**Variáveis**

```
y : inteiro;
```

**Início**

```
y ← 2 * x;  
dobroNum ← y;
```

**Fim;**

→ Parâmetro  
(passado por valor)

**Variável local:** só pode ser utilizada no corpo dessa função.

→ Note que o parâmetro "x" é utilizado como uma variável qualquer e que o seu escopo é local!

No corpo dessa função, evitou-se utilizar quaisquer variáveis globais. Com isso, essa função poderia ser reaproveitada facilmente em outro algoritmo.



# Passagem de Parâmetros por Referência (1)

**Argumentos** e **parâmetros** são mantidos separadamente na memória. Quando um argumento é passado por valor, ele é atribuído (copiado) para a variável de parâmetro. Por isso, se esse parâmetro for alterado dentro do módulo, *essa alteração não terá efeito sobre o argumento*.

Modificando o valor de um Argumento:

## **Passagem de Parâmetros por Referência**

Sabemos que uma variável é *uma referência a uma posição de memória*, que armazena um determinado valor. Quando um argumento é **passado por referência**, a variável de parâmetro passa a referenciar a mesma posição de memória que o argumento. É por essa razão que a variável de parâmetro também é chamada de **variável de referência**: as modificações feitas no parâmetro se refletem no argumento (ou seja, o argumento pode ser modificado, ao contrário do que acontece na passagem de parâmetros por valor).

```
Procedimento dobraNum (x :inteiro,  
                        var y :inteiro) ;
```

**Início**

```
y ← 2 * x;
```

**Fim;**

No procedimento “*dobraNum*”, são definidos dois parâmetros inteiros. “x” é **passado por valor** e, se sofrer alterações, elas não se refletirão no argumento. Já “y” é **passado por referência** (note a *palavra reservada var*) e, ao ser alterado, modifica o valor da variável usada como argumento.

Uma **variável de referência** é um *alias* para a variável usada como argumento. Quaisquer alterações feitas nela (ou seja, no parâmetro passado por referência) acabam alterando o próprio argumento.

# Passagem de Parâmetros por Referência (1)

**Parâmetro passado por valor:** quaisquer alterações não são propagadas fora desse procedimento.

"y" é alterado no corpo do procedimento. Como se trata de um parâmetro passado por referência, a modificação se reflete no argumento. Se alterássemos o valor de "x" (passado por referência), essa alteração não teria efeito fora do escopo desse procedimento.

```
Procedimento dobraNum(x :inteiro,  
var y :inteiro);  
Inicio  
y ← 2 * x;  
Fim;
```

No procedimento "dobraNum", são definidos dois parâmetros inteiros. "x" é **passado por valor** e, se **Parâmetro passado por referência:** se alterado, modifica o valor do argumento.

# Passagem de Parâmetros por Referência (2)

```
Algoritmo testaDobra2;
```

```
Variáveis
```

```
    vlNum, vlDobroNum      : inteiro;
```

```
    Procedimento dobraNum(x:inteiro, var y :inteiro);
```

```
    Inicio
```

```
    y ← 2 * x;
```

```
    Fim;
```

```
Início
```

```
Escreva ("Entre com um número") ;
```

```
Leia (vlNum) ;
```

```
dobraNumero (vlNum, vlDobroNum) ;
```

```
Escreva ("O dobro de ", vlNum, " eh ", vlDobroNum) ;
```

```
Fim.
```

# Discussão e Resolução de Exercícios Propostos