algoritmos e programação

com exemplos em Pascal e C

nina edelweiss
maria aparecida castro livi

bookman

as autoras

Nina Edelweiss é engenheira eletricista e doutora em Ciência da Computação pela Universidade Federal do Rio Grande do Sul. Durante muitos anos, lecionou em cursos de Engenharia e de Ciência da Computação na UFRGS, na UFSC e na PUCRS. Foi, ainda, orientadora do Programa de Pós-Graduação em Ciência da Computação da UFRGS. É coautora de três livros, tendo publicado diversos artigos em periódicos e em anais de congressos nacionais e internacionais. Participou de diversos projetos de pesquisa financiados por agências de fomento como CNPq e FAPERGS, desenvolvendo pesquisas nas áreas de bancos de dados e desenvolvimento de software.

Maria Aparecida Castro Livi é licenciada e bacharel em Letras, e mestre em Ciência da Computação pela Universidade Federal do Rio Grande do Sul. Desenvolveu sua carreira profissional na UFRGS, onde foi programadora e analista de sistema, antes de ingressar na carreira docente. Ministrou por vários anos a disciplina de Algoritmos e Programação para alunos dos cursos de Engenharia da Computação e Ciência da Computação. Sua área de interesse prioritário é o ensino de Linguagens de Programação, tanto de forma presencial quanto a distância.

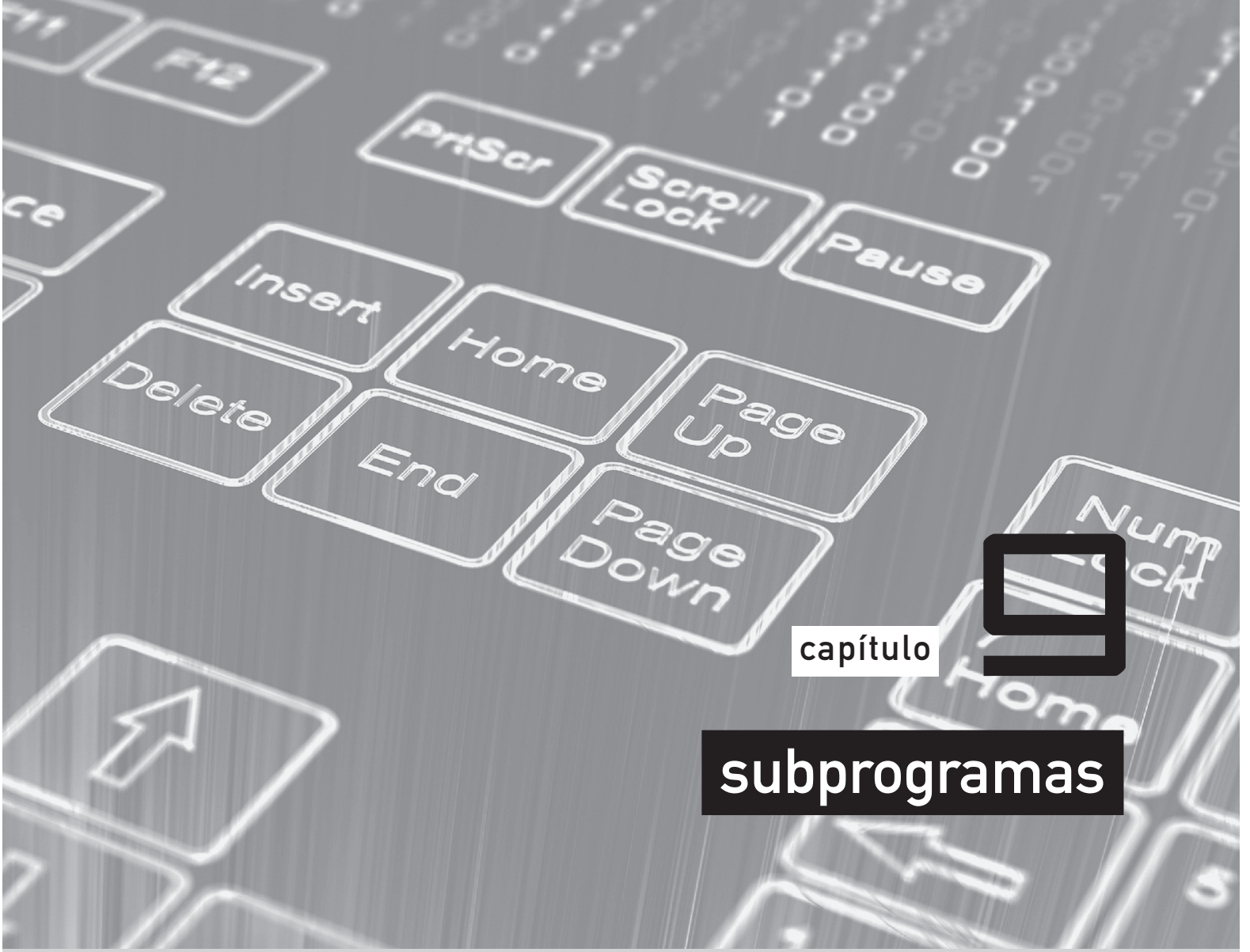
E22a Edelweiss, Nina.
Algoritmos e programação com exemplos em Pascal e C [recurso eletrônico] / Nina Edelweiss, Maria Aparecida Castro Livi. – Dados eletrônicos. – Porto Alegre : Bookman, 2014.

Editado também como livro impresso em 2014.
ISBN 978-85-8260-190-7

1. Informática. 2. Algoritmos – Programação. I. Livi, Maria Aparecida Castro. II. Título.

CDU 004.421

Catalogação na publicação: Ana Paula M. Magnus – CRB 10/2052



capítulo 9

subprogramas

Este capítulo discute o conceito de subprogramação, analisando dois tipos de subprogramas: procedimentos e funções. São detalhadas a declaração e a chamada de um subprograma. Diferentes tipos de parâmetros são analisados: formais e reais, de entrada e de saída, passados por valor ou por referência. São apresentadas as diferenças entre variáveis locais e globais, o conceito de escopo de identificadores e o conceito de desenvolvimento de algoritmos por meio de refinamentos sucessivos.

A arte de programar consiste na arte de organizar e dominar a complexidade dos sistemas.
— Dijkstra, 1972.

Um aspecto fundamental na programação estruturada é a decomposição de um algoritmo em módulos, usando a técnica denominada **programação modular** (Staa, 2000). O objetivo da programação modular é diminuir a complexidade dos programas, usando a estratégia de "dividir para conquistar": dividir problemas complexos em problemas menores.

Usando essa estratégia, um algoritmo é dividido em partes menores, chamadas de **módulos**. Cada módulo tem um único ponto de entrada, e sua execução termina em um único ponto de saída, contendo no seu fluxo de execução: sequências de ações (cap. 3), seleções (cap. 4) e iterações (cap. 5). Cada módulo é implementado por um subalgoritmo (subprograma) específico, passível de ser construído e testado de forma independente. Os diferentes subprogramas são posteriormente integrados em um só programa. A modularização tem o objetivo de facilitar a compreensão, o desenvolvimento, o teste e a manutenção dos sistemas. Programas que utilizam subprogramação resultam mais confiáveis e flexíveis.

Considere que o problema a ser resolvido através do computador é a solução da fórmula que calcula as combinações de "*n*" elementos tomados "p" a "p":

$$\binom{n}{p} = \frac{n!}{(n-p)!p!}$$

Para obter o resultado dessa fórmula, por meio de um algoritmo, é necessário fazer três vezes o cálculo de um fatorial. O cálculo desses três fatoriais é muito semelhante, só mudando o valor limite de cada um. Considerando que tanto o fatorial de 0 quanto o de 1 são iguais a 1 e utilizando uma variável inteira `contador`, esses cálculos podem ser realizados pelos seguintes trechos de programa:

```
{CÁLCULO DO FATORIAL DE N}
fatorial  ← 1
para contador de 2 incr 1 até n faça
fatorial  ← fatorial * contador
...
{CÁLCULO DO FATORIAL DE (N-P)}
fatorial  ← 1
para contador de 2 incr 1 até (n-p) faça
fatorial  ← fatorial * contador
...
{CÁLCULO DO FATORIAL DE P}
fatorial  ← 1
para contador de 2 incr 1 até p faça
   fatorial  ← fatorial * contador
```

O que muda nos três casos é somente o limite superior do comando `para/faça`. A repetição quase igual desses códigos pode ser evitada se for definido um trecho de código independente, denominado subprograma, em que o fatorial é calculado. Esse subprograma será acionado (chamado) pelo programa em execução cada vez que se quiser calcular um fatorial,

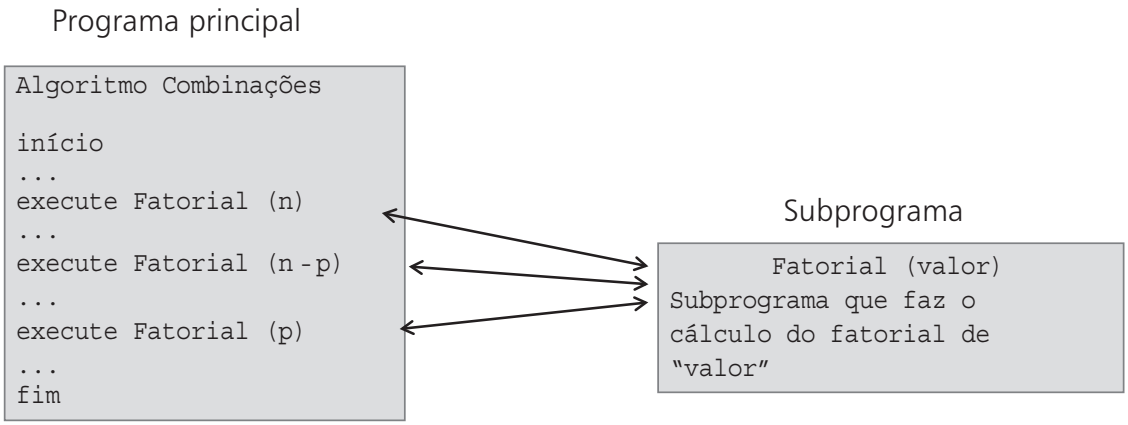


figura 9.1 Chamadas ao subprograma que calcula o fatorial.

informando somente para qual valor se quer que o fatorial seja calculado, conforme mostra a Figura 9.1.

Este capítulo discute o conceito de subprogramação, analisando dois tipos de subprogramas: procedimentos e funções.

9.1 → conceito de subprogramação

Um **subprograma** (às vezes chamado de sub-rotina) consiste de um trecho de código com estrutura semelhante à de um programa, que é executado somente quando acionado por outro trecho de código. Esse acionamento costuma-se denominar chamada ao subprograma.

Um subprograma deve executar uma única tarefa, claramente definida. Um programa, ao utilizar um subprograma para executar uma tarefa, não deve se preocupar em como essa tarefa será executada. A execução correta de um subprograma deve ser assegurada sempre que esse seja adequadamente chamado.

A utilização de subprogramas é uma técnica de programação que visa:

- a definição de trechos de código menores, mais fáceis de serem construídos e testados;
- a diminuição do tamanho dos programas, pela eliminação de redundâncias, ao evitar que códigos semelhantes sejam repetidos dentro de um programa;
- a construção mais segura de programas complexos, pela utilização de unidades menores (os subprogramas) já construídas e testadas;
- a reutilização de código em um programa ou em programas diferentes.

Todo subprograma é identificado através de um nome. Esse nome deve representar claramente a tarefa a ser executada pelo subprograma. Por exemplo, o nome adequado para o subprograma da Figura 9.1 é `Fatorial`.

A chamada a um subprograma é feita pelo seu nome, por um comando específico ou utilizando diretamente seu resultado, conforme será visto mais adiante. Um mesmo subprogra-

ma pode ser chamado e executado diversas vezes, em diferentes pontos de um programa. Adicionalmente, um subprograma também pode conter chamadas a outros subprogramas.

9.2 → implementação de chamadas a subprogramas

Quando um subprograma é chamado, o fluxo de execução do programa ou subprograma que o chamou é interrompido e o subprograma passa a ser executado. Terminada a execução do subprograma, o fluxo de execução interrompido é retomado, e o processamento segue a partir do ponto imediatamente após a chamada concluída. Na Figura 9.2, é mostrado, por meio de setas, o fluxo de execução de um programa que chama o subprograma, o qual calcula o fatorial.

Como já mencionado, um subprograma pode chamar outro subprograma. A forma de execução é sempre a mesma: o que chamou fica suspenso, esperando o término da execução do subprograma chamado, continuando depois sua execução a partir da instrução seguinte à instrução de chamada do subprograma.

Na Figura 9.3 é mostrado o fluxo de controle, representado simbolicamente por meio de setas, entre um programa e diversos subprogramas. O programa principal, durante sua execução, chama o subprograma A, ficando suspenso à espera do final da execução de A. O subprograma A, por sua vez, chama o subprograma B, ficando também suspenso enquanto B não terminar. Em seguida, o subprograma B chama o subprograma C. Durante a execução de C (Figura 9.3a), o programa principal e os subprogramas A e B estão suspensos. Quando a execução de C termina (Figura 9.3b), o fluxo de controle retorna ao subprograma B, que é executado até o fim, devolvendo, então, o controle ao subprograma A. Quando esse último termina, devolve o controle ao programa principal, que então continua sendo executado. Observar que somente um dos módulos (programa principal ou subprograma) estará sendo executado a cada momento.

Cada vez que a execução de um código é interrompida, é necessário que os recursos locais (ponto em que o programa fez a chamada, valores de variáveis locais, etc.) de quem fez a

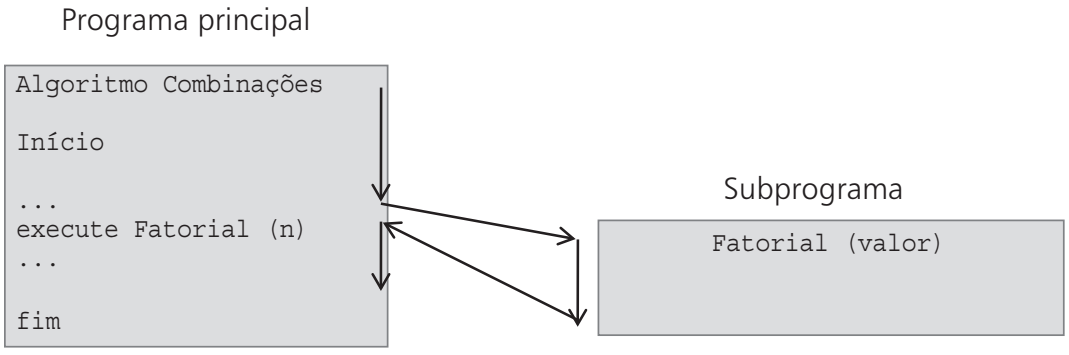


figura 9.2 Fluxo de execução entre programa e subprograma.

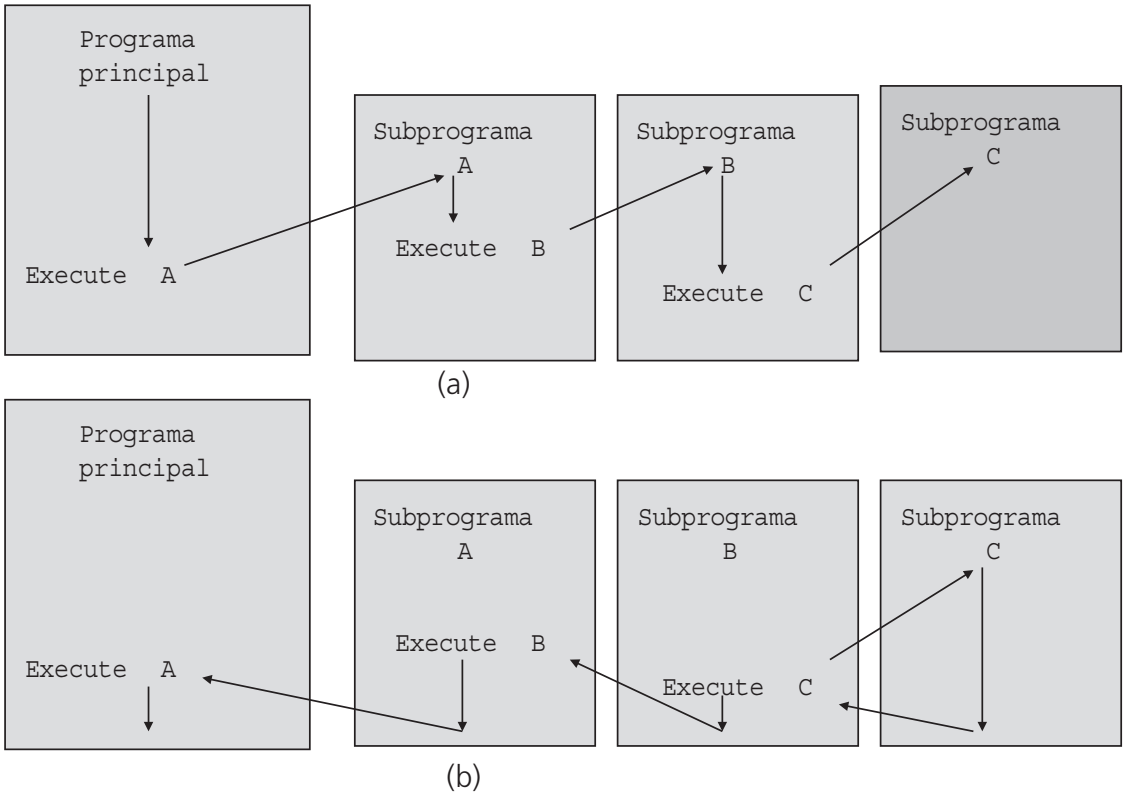


figura 9.3 Vários níveis de chamadas a subprogramas.

chamada sejam preservados e permaneçam disponíveis até o momento em que seja possível retomar essa execução.

Em consequência, a implementação das sucessivas chamadas a subprogramas nas linguagens em discussão neste livro implica na criação, pelo sistema, de uma estrutura de pilha, para armazenamento dos elementos locais das chamadas interrompidas, assim como dos endereços dos pontos a partir de onde as execuções devem ser retomadas. À medida que chamadas vão sendo feitas e interrompidas, a pilha vai recebendo valores. Quando a condição de término das chamadas é atingida, progressivamente as chamadas suspensas vão sendo concluídas e seus elementos locais restaurados, a partir da pilha. Embora não seja usual haver limitação para o número de níveis de chamadas a subprogramas nas linguagens, o espaço de armazenamento reservado para a pilha de elementos locais é finito e, se ultrapassado, pode determinar a ocorrência de erros de processamento.

A execução de um subprograma, considerando a estrutura da pilha associada, compreende os seguintes passos:

a início da execução – criação de elementos locais, declarados no subprograma;

b processamento – tentativa de execução do código do subprograma até o seu final. Se a execução for interrompida por uma chamada a um subprograma, empilhamento dos elementos locais e do endereço de retorno, antes de ativação de nova chamada;

c final da execução de uma chamada do subprograma – liberação das áreas de memória locais e retorno ao ponto de onde o subprograma foi chamado. O ponto da chamada pode integrar outra chamada em suspenso, que só então poderá ser concluída;

d retomada de uma execução interrompida – elementos locais são restaurados, a partir da pilha gerida pelo sistema, e o processamento é retomado a partir do endereço também retirado da pilha.

9.3 → parâmetros

Os valores que um subprograma necessita receber para poder realizar sua tarefa, ou os valores que produz e que devem ser visíveis externamente após concluída sua execução, devem ser sempre armazenados em parâmetros. Parâmetros são espaços de armazenamento que permitem a comunicação do subprograma com o mundo externo. No subprograma que calcula o fatorial, tanto o número para o qual deve ser realizado esse cálculo, quanto o resultado produzido, são valores que potencialmente devem ser armazenados em parâmetros. Um subprograma que não utiliza parâmetros e não utiliza informação do mundo exterior para seu acionamento, produzirá sempre o mesmo resultado, não importa de onde seja chamado.

9.3.1 parâmetros formais e reais

Todos os elementos utilizados em um subprograma devem ser nele declarados. As declarações de parâmetros, além de nomeá-los para uso interno no subprograma, definem seu tipo. Os parâmetros que aparecem na declaração dos subprogramas são chamados de **parâmetros formais** porque, durante a execução, na chamada dos subprogramas, são substituídos por variáveis ou valores do mesmo tipo, muitas vezes com nomes totalmente diferentes.

Como exemplo, vamos considerar um subprograma que calcula o fatorial de um número:

```
Subprograma Fatorial
  Parâmetro: número (inteiro)
```

O parâmetro formal `número` não provoca a reserva de espaço na memória. Ele simplesmente indica que, ao ser chamado o subprograma `Fatorial`, deve ser fornecido um número inteiro para sua execução.

Os parâmetros utilizados na chamada de um subprograma, chamados de **parâmetros reais**, substituem os formais durante sua execução. Os parâmetros reais devem sempre concordar em quantidade e tipo com os respectivos parâmetros formais, na ordem em que esses foram definidos. Podem ser fornecidos como parâmetros reais nomes de variáveis, valores literais ou resultados de expressões. As variáveis utilizadas como parâmetros reais devem ter sido declaradas no programa que chama o subprograma.

No exemplo anterior do subprograma `Fatorial`, o parâmetro real que vai substituir `número` pode ser fornecido por meio de um valor literal inteiro, do conteúdo de uma variável inteira ou de uma expressão que resulte em um valor inteiro. Se, no programa principal, estiverem

declaradas as variáveis inteiras `int1` e `int2`, os seguintes comandos podem ser utilizados para chamar o subprograma `Fatorial`:

```
execute Fatorial (5)
execute Fatorial (int1)
execute Fatorial (int1 + int2)
```

Na primeira chamada, o parâmetro formal `número` é substituído pelo valor 5; na segunda chamada, `número` é substituído pela variável `int1`; e, na última chamada, `número` é substituído pelo resultado da expressão `int1 + int2`.

9.3.2 parâmetros de entrada e de saída

A utilização de um subprograma só é efetiva se for claramente definida a tarefa que será executada e qual sua interação com o programa que o acionou. Para que isso seja possível, é importante que toda a interação seja feita somente através dos parâmetros, identificando quais os **parâmetros de entrada** (que recebem variáveis ou valores para executar a tarefa) e quais os **parâmetros de saída** (que devolvem os valores calculados ao programa que acionou o subprograma).

No exemplo anterior, do subprograma que calcula o fatorial, não foi definido como o resultado deveria ser devolvido ao programa. Um segundo parâmetro deve ser definido, por meio do qual será informado o valor calculado para o fatorial. No cabeçalho da declaração do subprograma devem ser identificados os parâmetros de entrada (que recebem variáveis ou valores para a execução) e os de saída (que devolvem valores):

```
Subprograma Fatorial
  Parâmetro de entrada: número (inteiro)
  Parâmetro de saída: fat (inteiro)
```

As chamadas a `Fatorial` devem fornecer agora dois parâmetros reais, sendo o primeiro parâmetro o número para o qual se quer calcular o fatorial, e o segundo, o nome de uma variável na qual será devolvido o resultado:

```
execute Fatorial (4, int1)
execute Fatorial (int1 + 7, int1)
```

Na primeira chamada, o resultado do cálculo do fatorial de 4 é devolvido através da variável `int1`; na segunda chamada, o conteúdo da variável `int1` é alterado após a execução, passando a conter o valor do fatorial do valor que armazenava anteriormente somado com a constante 7.

9.3.3 parâmetros por valor ou por referência

A passagem de valores a subprogramas pode acontecer **por valor** ou **por referência**.

A passagem por valor indica que somente o valor interessa ao subprograma. Se esse valor for passado por meio do nome de uma variável, somente o valor da variável é transferido para o

parâmetro. Uma cópia do conteúdo da variável é carregada em uma variável auxiliar, que será utilizada durante a execução do subprograma. Dessa forma, qualquer modificação no valor da variável auxiliar não se refletirá na variável utilizada como parâmetro real. A passagem de valores para parâmetros definidos por valor pode ser feita ainda por meio de um valor literal e do resultado de uma expressão. Na execução da chamada a um subprograma, os parâmetros passados por valor também são incluídos na pilha de execução, preservando seus valores para a continuação posterior da execução.

Na passagem de um parâmetro por referência, o endereço físico da variável utilizada como parâmetro real é passado ao subprograma, sendo essa variável utilizada durante a execução. Alterações no valor do parâmetro são feitas diretamente nessa variável. Na chamada de um subprograma, os parâmetros definidos por referência recebem nomes de variáveis existentes no programa principal. É importante observar que, na execução de uma chamada ao subprograma, os parâmetros por referência não sofrem empilhamento, já que não são locais aos subprogramas.

No exemplo anterior do subprograma `Fatorial`, o primeiro parâmetro – de entrada – é passado por valor. O segundo, que devolve o valor calculado, deve ser definido por referência. Na pseudolinguagem, um parâmetro passado por referência é identificado pela palavra `ref` antes de seu nome:

```
Subprograma Fatorial
  Parâmetro de entrada: número (inteiro)
  Parâmetro de saída: ref fat (inteiro)
```

9.4 → declarações locais e globais

Dentro de um subprograma, podem ser feitas declarações de constantes, tipo e variáveis. As declarações feitas internamente aos subprogramas são **declarações locais** ao subprograma, e só são visíveis dentro do subprograma. As áreas de memória associadas às variáveis locais são alocadas no momento em que o subprograma é acionado e são liberadas ao final de sua execução, quando deixam de existir. Todo esse processo de criação e destruição de variáveis locais ocorre novamente a cada nova chamada ao subprograma.

Como exemplo de utilização de uma variável local, será considerado um subprograma que permuta o conteúdo de duas variáveis inteiras. Os parâmetros formais `A` e `B` representam as duas variáveis e, neste exemplo, desempenham o papel tanto de parâmetros de entrada quanto de saída. Para fazer a permuta é necessário uma terceira variável para guardar um dos valores durante o processamento. Essa será definida como uma variável local, pois sua existência não é relevante para o programa principal:

```
Subprograma Permuta
{TROCA O CONTEÚDO DE DUAS VARIÁVEIS}
  Parâmetros entrada e saída: ref A, ref B (inteiro)
  Variável: aux (inteiro)  {VARIÁVEL LOCAL}
```

```
início
  aux ← A
  A ← B
  B ← aux
fim {TrocaDois}
```

No fim da execução do subprograma `Permuta`, a variável `aux` é liberada e não está mais disponível.

Tipos, constantes e variáveis declarados em um programa, visíveis aos subprogramas que estiverem nele declarados, são consideradas **declarações globais** ao programa.

Embora elementos globais possam ser utilizados dentro de subprogramas, essa prática não é recomendável, pois dificulta o entendimento e a depuração dos códigos, podendo facilmente ocasionar erros. Toda interação de um subprograma com o programa que o chama, no limite das possibilidades de cada linguagem, deve ser feita através de parâmetros, devendo os demais elementos necessários à execução de sua tarefa ser declarados localmente.

Segue um quadro resumo das características das declarações locais e globais.

Locais	Globais
Declaradas internamente aos subprogramas que as acessam.	Declaradas externamente aos subprogramas que as acessam.
Só são reconhecidas e só podem ser referenciadas nos subprogramas em que estão declaradas.	São reconhecidas e podem ser referenciadas até mesmo em subprogramas em que não foram declaradas.
Existem apenas enquanto os subprogramas em que foram declaradas estiverem em execução.	Sua existência independe dos subprogramas que as acessam. Existem antes, durante e após a ativação deles.
Internamente a um subprograma, quando têm o mesmo nome que uma global, bloqueiam o acesso à global.	

9.4.1 escopo de identificadores

A possibilidade de fazer declarações em diferentes pontos de um programa (no programa principal e em subprogramas) requer que seja claramente identificado o escopo de cada identificador. Por escopo entende-se a área de validade de um determinado identificador. O escopo de um identificador é definido pela localização de sua declaração. Declarações feitas no programa principal valem em todo o programa, inclusive, por vezes, nos subprogramas que o compõem. Declarações feitas dentro de um subprograma valem somente dentro desse subprograma. Exemplo:

```
Programa Exemplo
  Variáveis:
    um (inteiro)
    dois (real)
{-----------------------------------------------}
Subprograma Sub
  Parâmetro de entrada: valor (inteiro)
  Parâmetro de saída: ref result (real)
  Variáveis locais:
    loc1, loc2 (inteiro)
início
  {AQUI PODEM SER UTILIZADOS:
    - os parâmetros valor e result
    - as variáveis globais um e dois
    - as variáveis locais loc1 e loc2 }
...
fim {Sub}
{-----------------------------------------------}
início
  { AQUI PODEM SER UTILIZADAS:
    - as variáveis um e dois
    - chamadas ao subprograma Sub}
fim {PROGRAMA PRINCIPAL}
```

Nas declarações feitas dentro dos subprogramas, podem ser utilizados nomes iguais a outros elementos (constantes, tipos ou variáveis) já existentes no programa principal. No exemplo a seguir, no programa principal é declarada uma variável `item`, do tipo inteiro. No subprograma, é definida uma variável local com o mesmo nome (`item`), do tipo caractere. Dentro do subprograma, qualquer referência ao identificador `item` se refere à variável local, ficando a global de mesmo nome inacessível.

```
Programa Exemplo
  Variáveis:
    item (inteiro)
    glob (real)
{-----------------------------------------------}
Subprograma Sub
  Parâmetro de entrada: valor (inteiro)
  Parâmetro de saída: ref result (real)
  Variável local:
    item (caractere)
início
  {AQUI PODEM SER UTILIZADOS:
```

```
    - os parâmetros valor e result
    - a variável global glob
    - a variável local item (caractere) }
...
fim {Sub}
{-----------------------------------------------}
início
  { AQUI PODEM SER UTILIZADAS:
    - as variáveis item (inteiro) e glob
    - chamadas ao subprograma Sub}
fim {PROGRAMA PRINCIPAL}
```

O escopo dos identificadores deve ser atentamente observado quando existirem declarações de subprogramas dentro de subprogramas, todos com declarações locais. A Figura 9.4 mostra graficamente essa situação, considerando um programa principal dentro do qual está declarado um subprograma A. Dentro do subprograma A, está declarado outro subprograma B. À direita na figura estão representadas as variáveis que podem ser utilizadas por comandos dentro de cada um desses blocos – programa principal, subprograma A e subprograma B.

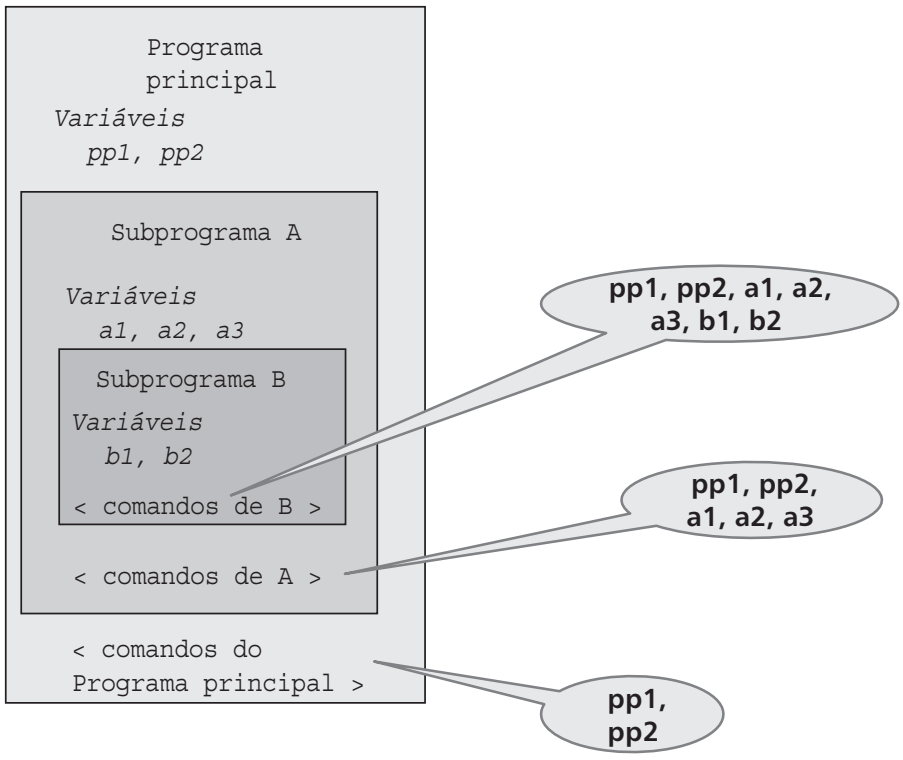


figura 9.4 Escopo dos identificadores.

Cabe lembrar mais uma vez que não é aconselhável utilizar variáveis globais dentro dos subprogramas. Sempre que possível, toda a comunicação dos subprogramas com os programas que os acionam deverá ser feita através de parâmetros.

9.5 → tipos de subprogramas

Dois tipos de subprogramas podem ser utilizados, diferenciados pela forma como são acionados e o modo como devolvem seus resultados: procedimentos e funções.

9.5.1 procedimentos

Um **procedimento** é um subprograma que executa uma determinada tarefa com ou sem a utilização de parâmetros. Na pseudolinguagem, a declaração de um procedimento é feita como segue:

```
Procedimento <nome do procedimento>
{ <Descrição da tarefa executada pelo procedimento> }
  <Lista de parâmetros formais, cada um com seu tipo,
   identificando os de entrada e os de saída>
  <Lista de constantes, tipos e variáveis locais>
início
  <Comandos do procedimento>
fim {<nome do procedimento>}
```

Na `lista de parâmetros formais`, os parâmetros por referência devem ser identificados pela sigla `ref` antes do seu nome. Os parâmetros de saída devem sempre ser definidos por referência.

Por exemplo, a declaração de um procedimento que calcula o fatorial de um número é:

```
Procedimento Fatorial
{CALCULA O FATORIAL DE UM NÚMERO}
  Parâmetro entrada: número (inteiro)
  Parâmetro de saída: ref fat (inteiro)
  Variável local: contador (inteiro)
início
  fat ← 1
  para contador de 1 incr 1 até número faça
    fat ← fat * contador
fim {Fatorial}
```

Os procedimentos são acionados por meio de um comando especial. Na pseudolinguagem, esse comando é:

```
execute <nome do procedimento> (< lista de parâmetros reais >)
```

Na chamada ao procedimento `Fatorial`, devem ser fornecidos dois parâmetros: o primeiro deve ser o valor para o qual se quer calcular o fatorial, que pode ser um valor literal, o conteúdo de uma variável ou o resultado de uma expressão aritmética; o segundo deve ser o nome da variável na qual será devolvido o resultado do cálculo efetuado.

Se `a1`, `a2` e `a3` forem variáveis inteiras, podem ser feitas as seguintes chamadas ao procedimento `Fatorial`:

```
execute Fatorial (a1, a2) {a2  DEVOLVE FATORIAL DE a1}
execute Fatorial (a2, a3) {a3  DEVOLVE FATORIAL DE a2}
execute Fatorial (7, a2)  {a2  DEVOLVE FATORIAL DE 7}
execute Fatorial (a1+2,a3){a3  DEVOLVE FATORIAL DE (a1+2)}
```

9.5.2 funções

Uma **função** é um subprograma que devolve um valor, resultante de um cálculo ou da execução de uma determinada tarefa, ao programa que o chamou por meio de seu nome. Uma função tem sempre associado a ela um tipo, que é o tipo do valor que ela devolve.

Na pseudolinguagem, a declaração de uma função é feita como segue:

```
Função <nome da função> : <tipo do resultado>
{ <Descrição do que é calculado pela função> }
  <Lista de parâmetros formais, cada um com seu tipo>
  <Lista de constantes, tipos e variáveis locais>
início
  <Comandos>
fim {<nome da função>}
```

Em algum lugar do corpo da função, o valor calculado pela função deve ser atribuído ao seu nome, para ser devolvido ao programa que a chamou. O nome da função deve, portanto, aparecer pelo menos uma vez, no corpo da função, à esquerda de uma atribuição:

```
<nome da função > ← <valor|variável|expressão|função|constante>
```

Uma função não é chamada através de um comando especial, como no caso dos procedimentos. A chamada é feita escrevendo seu nome, com os parâmetros reais, no ponto do código onde se deseja que o seu valor seja utilizado. Como exemplo, segue o cálculo do fatorial feito por meio de uma função e não de um procedimento, como mostrado na seção anterior:

```
Função Fatorial: inteiro
{CALCULA O FATORIAL DE UM VALOR INTEIRO}
  Parâmetro de entrada:
    número (inteiro)  {NÚMERO PARA O QUAL SE QUER O FATORIAL}
  Variáveis locais:
    índice (inteiro)
    fat (inteiro)
início
```

```
  fat ← 1
  para índice de 1 incr 1 até número faça
    fat ← fat * índice
  Fatorial ← fat          {DEVOLUÇÃO DE UM VALOR PELA FUNÇÃO}
fim {Fatorial}
```

Sendo `fat1`, `n`, `p` e `comb` variáveis inteiras, as chamadas à função `Fatorial` a seguir são válidas:

```
fat1 ← Fatorial(7)      {fat1 RECEBE O VALOR DO FATORIAL DE 7}
escrever (Fatorial(n))  {INFORMA FATORIAL DO VALOR CONTIDO EM n}
comb ← Fatorial(n) / (Fatorial (n-p) * Fatorial(p))
    {comb RECEBE CÁCULO DAS COMBINAÇÕES DE n ELEMENTOS p A p}
```

A execução de uma função termina quando seu final lógico é encontrado, ou seja, quando a execução atinge o ponto final da função. No caso da função `Fatorial`, por exemplo, isso ocorre quando `fim {Fatorial}`é atingido.

Embora, pelo conceito fundamental de função, deva ser devolvido somente um valor por meio de seu nome, sintaticamente parâmetros de saída também podem ser utilizados, devolvendo também valores ao programa que a aciona. Essa prática não é aconselhada porque, além de contrariar o conceito de função, torna os códigos mais complexos, favorecendo a ocorrência de erros. Assim, aconselha-se: sempre que uma função for utilizada, devolver apenas o valor em seu nome.

As chamadas a funções podem ocorrer em qualquer ponto do código em que o uso de um valor seja sintaticamente correto. Isso significa que funções podem ser usadas, por exemplo, em atribuições, em expressões, em comandos de saída, na chamada de outras funções e em testes.

9.6 → refinamentos sucessivos e programação modular

O uso de subprogramação com o objetivo de facilitar a construção de programas deve ser feito seguindo técnicas apropriadas, de acordo com a programação estruturada, conforme visto na Seção 1.4, no Capítulo 1.

Uma das técnicas aconselhadas pela programação estruturada é o desenvolvimento de algoritmos por fases ou refinamentos sucessivos. Isso é feito concentrando-se inicialmente nas tarefas maiores a serem executadas e no fluxo de controle entre elas. Uma vez essa fase estando bem definida, cada uma das tarefas identificadas é então refinada gradualmente, até que se chegue aos algoritmos finais que podem ser codificados em alguma linguagem de programação.

Essa técnica, também denominada *top down*, traz como consequência uma maior clareza no entendimento do problema como um todo, por não considerar inicialmente detalhes de implementação. Gera, também, uma documentação mais clara e compreensível dos algoritmos.