Fundamentos de Programação Estruturada em C

Luiz Fernando Martha

Referências Básicas

- Gries, D., *The Science of Programming*, Springer-Verlag, New York, 1985.
- Hehl, M.E., *Linguagem de Programação Estruturada FORTRAN 77*, McGraw-Hill, 1986.
- Arakaki, R. et. alli., *Fundamentos de Programação em C*, Livros Técnicos e Científicos.
- Kernighan, B.W.; Ritchie, D.M., *The C Programming Language*, Prentice-Hall, Second Edition, 1988.
- Savitch, W.J., *Pascal: An Introduction of the Art and Science of Programming*, The Benjamin / Cummings Publishing Company, 1984.

Introdução

Principal desafio:
"Como escrever eficientemente programas grandes?"

Um enfoque:
1. Pegue uma tarefa longa, quebre em pedaços (blocos) menores.
2. Repita 1. até que os pedaços sejam unidades inteligíveis e fáceis de serem manipuladas.

Programa bem projetado:
Unidades pequenas, independentes e bem documentadas.

Programação estruturada:
Formaliza a idéia de dividir em blocos. Isto força o programador a saber exatamente o estado do programa antes e depois de cada bloco e evita o chamado "código espaguete", onde não se tem noção em que estágio da execução o programa se encontra em uma determinada instrução.

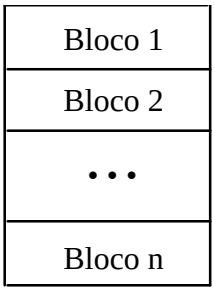
Diagrama de blocos:
- Utilizado para mostrar graficamente o fluxo de controle de um programa.
- Adotado a notação de diagrama de blocos N-S (Nassi-Schneiderman) [Arakaki, Apêndice B].

Comentários em C:
```
/* Este é um comentário no meio de um
   código em C */
```

Programação seqüencial

O fluxo de controle de um programa estruturado é feito de bloco em bloco, seqüencialmente, onde cada bloco pode ser:
- uma instrução;
- um conjunto de instruções;
- um aninhamento de outros blocos;
- uma rotina (função);
- um programa.

Diagrama N-S:



Para se poder garantir que o fluxo de controle é feito de forma seqüencial, associado a cada bloco existe uma asserção de pré-condição e uma asserção de pós-condição:
```
/* pre: asserção de pré-condição */
/* pos: asserção de pós-condição */
```

Uma asserção é um comentário que afirma alguma coisa que o programador espera ser verdadeira quando a execução do programa atinge a asserção.

Exemplo:
```
/* pre: x + 1 > 0 */
x = x + 1;
/* pos: x > 0 */
```

Uma "tripla" pre-bloco-pos diz que se a asserção pre é verdadeira antes da execução do bloco, então a asserção pos é verdadeira após a execução. A "tripla" não diz absolutamente coisa alguma a respeito da execução do bloco quando pre é falsa.

Estruturas de controle

Na programação estruturada, a estruturação está baseada em um pequeno número de estruturas de controle:

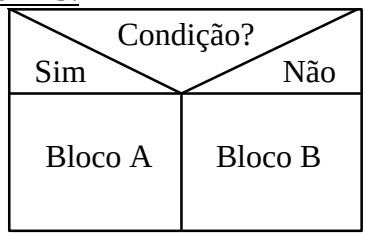
- controle seqüencial;
- controle de decisão;
- controle de seleção múltipla;
- controle de iteração do tipo *enquanto-faça*;
- controle de iteração do tipo *repita-enquanto*.

Pode-se estabelecer o Princípio da Programação Estruturada (Hehl, 1986):

"Qualquer algoritmo pode ser escrito combinando-se blocos formados pelas estruturas de controle acima".

Controle de decisão

Diagrama N-S:



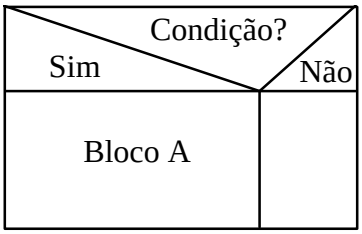
Pseudo-código:

```
/* exp: expressão booleana */
if( exp )
{
   Bloco A
}
else
{
   Bloco B
}
```

Exemplo em C:

```
/* Armazene em z o max(a,b) */
if( a > b )
{
   z = a;
}
else
{
   z = b;
}
/* Alternativa em C */
z = ( a > b ) ? a : b;
```

Bloco de Seleção com apenas um caso verdadeiro:



Pseudo-código:

```
if( exp )
{
   Bloco A
}
```

Expressão booleana:

Nos exemplos acima **exp** é uma expressão lógica relacional (expressão booleana) que contem:

- operadores relacionais:
  - maior que
  - maior ou igual a
  - menor que
  - menor ou igual a
  - igual a
  - diferente de
- operadores lógicos:
  - conjunção (**AND**)
  - disjunção (**OR**)
  - negação (**NOT**)

Expressões booleanas em C:

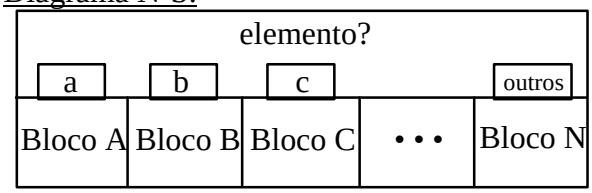
- operadores relacionais:
  - `>` (maior que)
  - `>=` (maior ou igual a)
  - `<` (menor que)
  - `<=` (menor ou igual a)
  - `==` (igual a)
  - `!=` (diferente de)
- operadores lógicos:
  - `&&` (conjunção)
  - `||` (disjunção)
  - `!` (negação)

Exemplos de asserções de pré- e pós-condição:

```
/* Armazene em z o max(a,b) */
/* pre0: TRUE */
if( a > b )
   /* pre1: a > b */
   z = a;
   /* pos1: z = a > b */
else
   /* pre2: a ≤ b */
   z = b;
   /* pos2: a ≤ b = z */
/* pos0: z = a > b  OR  a ≤ b = z */
/* z = max(a,b) */

/* De uma forma geral: */
/* pre0: TRUE */
if( exp )
{
   /* pre1: pre0 AND exp */
   Bloco A
   /* pos1: */
}
else
{
   /* pre2: pre0 AND NOT exp */
   Bloco B
   /* pos2: */
}
/* pos0: pos1 OR pos2 */
```

Controle de seleção múltipla

Diagrama N-S:



(onde a, b, c são constantes)

Implementação com switch em C:

```
switch( elemento )
{
   case a:
     Bloco A
     break;
   case b:
     Bloco B
     break;
   case c:
     Bloco C
     break;
   default:
     Bloco N
     break;
}
```

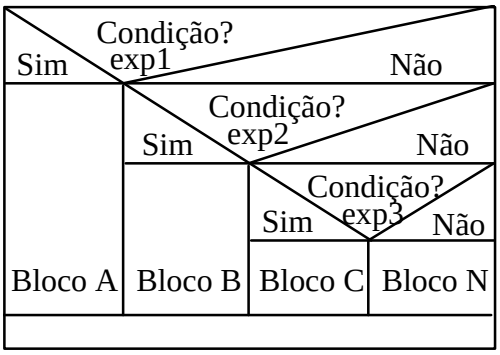
A instrução **break** é necessária ao final de cada opção pois senão as instruções do bloco seguinte (pertencentes à opção seguinte) seriam executadas. Isto é feito de forma a permitir que dois ou mais valores diferentes da variável de seleção fiquem associados a um mesmo bloco. Isto é mostrado abaixo:

```
switch( elemento )
{
   case a:
   case b:
   case c:
     Bloco A
     break;
   default:
     Bloco N
     break;
}
```

Implementação com if–else if–else em C:

```
if( elemento == a )
{
   Bloco A
}
else if( elemento == b )
{
   Bloco B
}
else if( elemento == c )
{
   Bloco C
}
else
{
   Bloco N
}
```

A estrutura if–else if–else também pode ser representada no diagrama N-S como mostrado abaixo:



Controle de Iteração (Laço ou Loop)

Existem basicamente dois tipos de controle de iteração:

- Enquanto ... faça ... (**`while`**)
- Repita ... enquanto ... (**`do while`**)

Diagrama N-S do loop **`while`**:

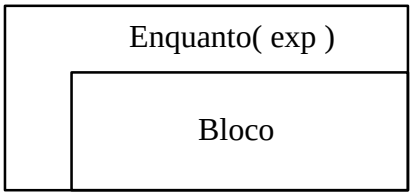
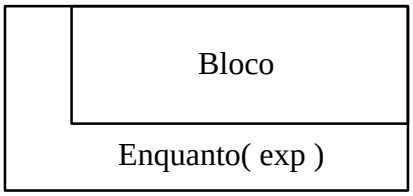


Diagrama N-S do loop **`do while`**:



Tanto o loop **`while`** quanto o loop **`do while`** terminam quando suas expressões booleanas são falsas. Uma diferença importante entre os dois tipos de loops é que no loop **`while`** a expressão booleana é verificada primeiro e se o seu valor for falso, o corpo do loop nunca é executado; no loop do tipo **`do while`**, o corpo do loop é sempre executado pelo menos uma vez pois a expressão booleana é verificada no final do loop.

Implementação de laços em C:

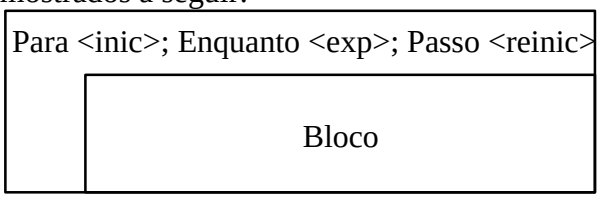
```
/* Enquanto ... faça ... */
while( exp )
{
   Bloco
}

/* Repita ... enquanto ... */
do
{
   Bloco
}
while( exp );
```

É interassante observar que na linguagem Pascal o loop **`do while`** é substituido pelo loop **`repeat until`** que termina quando a expressão booleana é verdadeira.

Um exemplo do loop **`while`** é mostrado na implementação da função **`binary`** que acha a posição de um número inteiro **`x`** em um vetor **`v[0..n-1]`** ordenado seqüencialmente (veja Kernighan & Ritchie, 1988):

```
int binary( int x, int v[], int n )
{
   int   low, high, mid;

   low = 0;
   high = n – 1;
   while( low <= high )
   {
     mid = (low + high) / 2;
     if( x < v[mid] )
          high = mid – 1;
     else if( x > v[mid] )
          low = mid + 1;
     else /* found match */
          return( mid );
   }
   return( –1 );
}
```

A linguagem C também apresenta um outro tipo de laço “enquanto ... faca ...”. É o loop do tipo **`for`**, cujo diagrama N-S e instruções são mostrados a seguir:




```
for( inic; exp; reinic )
{
   Bloco
}
```

Onde **`inic`** é um bloco de instruções de inicialização do laço, **`exp`** é a expressão booleana para repetição da iteração e **`reinic`** é um bloco de instruções para reinicialização da iteração. O laço **`for`** acima é inteiramente análogo ao laço **`while`** abaixo:

```
init;
while( exp )
{
   Bloco
   reinic;
}
```

A utilização mais freqüente do loop **`for`** é para fazer algum tipo de iteração com vetores, tal como inicializar com zeros o vetor de inteiros **`v[0..n-1]`**:

```
void init_vector( int v[], int n )
{
   int    i;

   for( i=0; i<n; i++ )
          v[i] = 0;
}
```

Alteradores do fluxo seqüencial

Um programa escrito com uma disciplina de programação estruturada é muito mais fácil de ser documentado e entendido. No entanto, a eficiência deve prevalecer. Por isso, alguns 'desvios' da forma estruturada são permitidos desde que não comprometam a clareza do programa e que quebrem de uma forma harmônica o fluxo seqüencial. São quatro as instruções de desvio do fluxo seqüencial em C:

- **break**;
- **continue**;
- **goto**;
- **return**.

A instrução **break** em C causa a terminação do comando **while**, **do while**, **for** ou **switch** que envolve o break e que é o mais interno. O fluxo passa para a instrução seguinte ao comando terminado.

A instrução **continue** em C faz com que o fluxo de um laço (**while**, **do while** ou **for**) passe para o fim do laço de forma que a expressão booleana para repetição seja imediatamente verificada. Os comandos **continue** e **goto** podem ser entendidos pelos dois loops inteiramente equivalentes mostrados a seguir:

```
while( exp )        | while( exp )
{                   | {
   Bloco A          |    Bloco A
   continue;        |    goto CONTIN;
   Bloco B          |    Bloco B
}                   | CONTIN:
                    | }
/* CONTIN é um rótulo */
```

A instrução **return** interrompe o fluxo pois causa o retorno imediato para o função que chamou a função ativa.

Desenvolvimento de Loops

Dentro da programação estruturada foi caracterizado a existência de asserções que podem ser feitas sobre o estado do programa antes de depois de cada bloco. Esta asserções são uma garantia que o programador tem sobre o comportamento do programa durante a sua execução. Dada a importância do laço em programação, é interessante que se possa estabelecer outras asserções a respeito do seu comportamento de forma a se ter uma gantantia de que as iterações de um laço irão terminar, convergindo para o resultado desejado.

Considere o loop a seguir, que armazena na variável **s** a soma dos elementos do vetor **b[0..n–1]**. O loop está comentado de forma que a asserção **inv** descreve os valores de **s** e **i** antes (e depois) de cada iteração.

```
/* pre:
 *  b[0..n–1], onde n ≥ 0;
 *  (n = 0 => b[0..–1], vetor vazio)
 */
i = 0;  s = 0;
/* inv:
 *  i = 0;
 *  s é a soma de b[0..–1];
 *  n iterações a fazer
 */
while( i != n )
{
/* inv:
 *  0 ≤ i < n;
 *  s é a soma de b[0..i–1];
 *  n–i iterações a fazer
 */
   s += b[i++];
}
/* inv = pos:
 *  i = n;
 *  s é a soma de b[0..n–1];
 *  0 iterações a fazer
 */
```

Algumas observações podem ser feitas a respeito do loop comentado:

- Se **n = 0** (vetor vazio), então a soma final é **s = 0**.
- Tem-se sempre **n–i** iterações a fazer:
  - **n–i** decresce a medida que o loop é executado;
  - se existe alguma iteração a fazer, então **n–i > 0**;

  - **n–i** é chamado de função teto (**ft**) do loop pois estabelece um valor máximo (um teto) para o número de iterações que ainda devem ser feitas durante a execução do loop. Savitch (1984) chama esta função de "variant expression".
- Se **inv** é uma asserção verdadeira e **ft=0**; então a asserção de pós-condição **pos** também é verdadeira.

A asserção **inv** é chamada de invariante do loop porque ela é invariavelmente verdadeira antes e depois de cada iteração. A função teto **ft** é uma garantia de que o loop irá terminar.

Com base neste exemplo, pode-se descrever um método geral para o entendimento de um loop genérico, tal como o mostrado a seguir, e para demonstrar que ele irá terminar:

```
/* pre: ... */
initS; /* instruções para iniciar */
/* inv: ... */
while( exp )
{ /* inv: ... */
   S;  /* instruções do loop */
}
/* inv: ... */
/* pos: ... */
```

Método:

1. Ache um invariante **inv**.

2. Mostre que **initS;** estabelece o invariante, isto é, comprove a 'tripla':
   /* pre: */ initS; /* inv: */

3. Mostre que cada iteração do loop mantém a validade do invariante:
   /* inv: AND exp */ S; /* inv: */

4. Mostre que a asserção **pos** é verdadeira após o término do loop:
   /* inv: AND NOT exp */⇒/* pos: */

5. Demonstre que o loop termina, isto é, ache uma função teto **ft** tal que:
   - **ft** decresce em cada iteração;
   - **/* inv: AND exp */ ⇒ ft > 0**.

Um loop fica convenientemente comentado tal como mostrado abaixo. Este formato deixa claro o que são o invariante e a função teto, sendo que eles só precisam ser escritos uma só vez:

```
/* pre: ... */
initS;
/* inv: ... */
/* ft: ...  */
while( exp )
{
   S;
}
/* pos: ... */
```

Exemplos de desenvolvimento de loops:

1. Este programa acha a primeira posição **i** do valor **x** no vetor **b[0..n–1]**, dado que **x** está em **b[]**:

```
/* pre: x ∈ b[0..n–1]
 */
i = 0;
/* inv: x ∉ b[0..i–1] ∴
 *      x ∈ b[i..n–1]
 */
/* ft: n – i
 */
while( b[i] != x ) i++;
/* pos: b[i] = x
 */
```

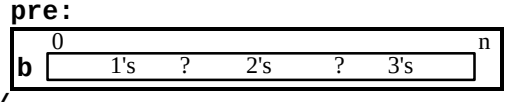
2. Este segundo programa calcula quociente **q** e o resto **r** quando **x** é dividido por **y**. Para **x>y>0**, **q** e **r** são definidos por:

   x = q*y + r e **0 ≤ r < q**

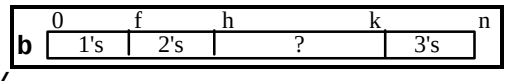
```
/* pre: x > y > 0
 */
q = 0;  r = x;
/* inv: x = q*y + r   e
 *      0 ≤ r
 */
/* ft: r
 */
while( r >= q )
{
   r –= y;
   q++;
}
/* pos: x = q*y + r   e
 *      0 ≤ r < q
 */
```

3. O terceiro programa um vetor **b[0..n–1]**, dado que contem números 1's, 2's e 3's, de tal forma que os 1's precedam os 2's que por sua vez precedam os 3's:

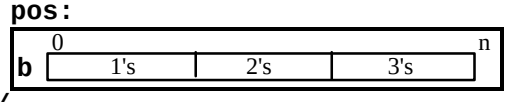
```
/* pre:
```




```
 */
f = 0;  h = 0;  k = n – 1;
/* inv:
```



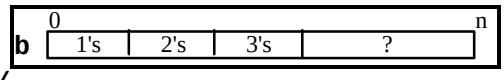

```
 */
/* ft: k – h
 */
while( h <= k )
   switch( b[h] )
   {
     case 1:
          t = b[h];
          b[h] = b[f];
          b[f] = t;
          f++;
          h++;
          break;
     case 2:
          h++;
          break;
     case 3:
          t = b[h];
          b[h] = b[k];
          b[k] = t;
          k––;
          break;
   }
/* pos:
```




```
 */
```

Obs.: O invariante abaixo seria menos eficiente pois exigiria até 2 trocas em cada iteração do loop:

```
/* inv:
```




```
 */
```

Exercícios propostos

1. Indique se as seguintes 'triplas' de instruções são falsas ou verdadeiras. Em caso de falsa, justifique. Todas as variáveis são inteiras.

(a)
```
/* i é par */
   i = i/2;
   /* i é par */
```

(b)
```
/* i é negativo */
   i = i*i;
   /* i é não-negativo */
```

(c)
```
/* z = a**b (a elevado a b) */
   b = b/2;
   a = a*a;
   /* z = a**b */
```

(d)
```
/* TRUE */
   while( i é par ) i = i/2;
   /* i é impar */
```

2. Escreva um loop, com inicialização, para somar os primeiros **n** números naturais pares e positivos. Utilize as asserções de pré-condição, pós-condição e invariante, e a função teto dadas abaixo:

```
/* pre:   n > 0
 */
/* pos:   s = 2 + 4 + 6 + ... + (2*n)
 */
/* inv:   2 ≤ k ≤ 2*n e k é par  e
 *        s = k + (k+2) + ... + (2*n)
 */
/* ft:    k/2
 */
```

3. É dado um vetor **b[0..n–1]**, onde **n ≥ 0**. Escreva um loop (com inicialização) para inverter o vetor **b** (por exemplo, se inicialmente **b = [1,3,5,2,6]**, então para invertê-lo significa modificá-lo para **b = [6,2,5,3,1]**). Utilize as asserções de pré-condição, pós-condição e invariante, e a função teto dadas abaixo:

```
/* pre:   b[0..n–1] não invertido
 */
/* pos:   b[0..n–1] invertido
 */
/* inv:   b[0..k–1] invertido
 *        b[k..j] não invertido
 *        b[j+1..n–1] invertido
 */
/* ft:    j–k
 */
```

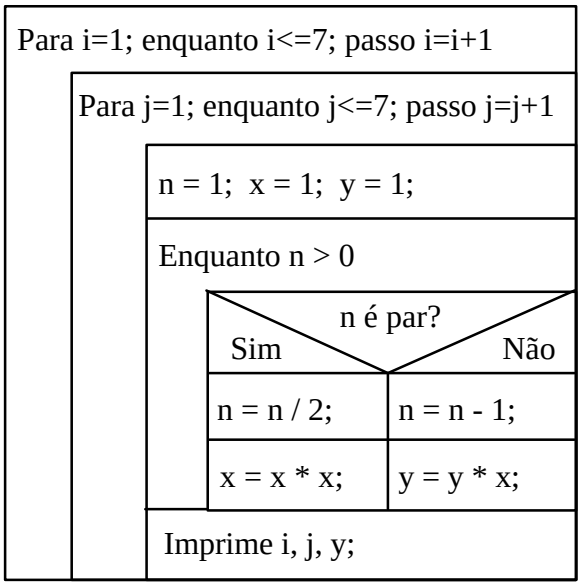
4. Um polinômio de grau **n** (**n** ≥ **0**) em **y** tem a seguinte forma:

   $b_0 + b_1*y + b_2*y^2 + ... + b_n*y^n$

   Suponha que os coeficientes sejam dados em um vetor **b[0..n]**. Escreva um loop (com inicialização) para achar o valor **v** do polinômio para um dado valor de **y**. A função teto é **ft: n–i**. Usando uma variável auxiliar **x**, o invariante do loop é:

```
/* inv:   0 ≤ i ≤ n  e
 *        x = y^i  e
 *        v = b0 + b1*y + ... + bn*y^i
 */
```

5. (Hehl, 1986) Escreva um programa em C para o algoritmo apresentado no diagrama N-S abaixo. O que faz este programa?



Funções, Variáveis e Utilização de Memória em C

Luiz Fernando Martha

Referências Básicas

- Kernighan, B.W.; Ritchie, D.M., *The C Programming Language*, Prentice-Hall, Second Edition, 1988.
- Savitch, W.J., *Pascal: An Introduction of the Art and Science of Programming*, The Benjamin / Cummings Publishing Company, 1984.

Introdução

Um dos aspectos que caracterizam uma linguagem de programação é o mecanismo utilizado para o acionamento de uma rotina (função). Este mecanismo varia quanto ao escopo de utilização de memória do computador e quanto à forma como os argumentos (parâmetros) são passados da rotina que chama para a rotina que é chamada. Uma das diferenças básicas entre Fortran e C está justamente neste mecanismo.

Funções e utilização de memória

Do ponto de vista de um programador de Fortran, uma subrotina pode ser entendida como um bloco de instruções e dados locais que tem uma localização fixa na memória do computador. Quando uma subrotina é acionada em Fortran, o fluxo de controle salta ('jump') para a sua primeira instrução.

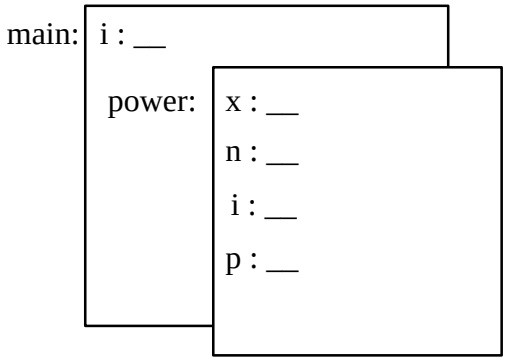
O problema com este mecanismo é que em Fortran não é permitido que uma rotina chame a si própria recursivamente, mesmo que indiretamente (alguns compiladores podem permitir isto, mas tal procedimento não é padronizado). Isto ocorre porque os dados locais de uma rotina têm endereço fixo na memória.

O mesmo não ocorre quando uma rotina (função) é acionada em C (ou mesmo Pascal). Neste caso, o programa requisita do sistema operacional um bloco de memória para executar a função enquanto esta estiver ativa. Esta memória é usada para armazenar as variáveis locais que são necessárias para executar a função. Estas variáveis locais incluem cópias dos eventuais argumentos que são passados para a função.

Tomemos, por exemplo, a função **power**, mostrada abaixo, que retorna o valor do número inteiro **x** elevado à **n**-ésima potência. Esta função é chamada pela rotina principal **main** que imprime na tela todas as potências do número 2 de 0 até 9.

```
int power( int x, int n )
{
   int   i, p;

   p = 1;
   for( i=1; i<=n; i++ )
     p *= x;
   return( p );
}

void main( void )
{
   int   i;

   for( i=0; i<10, i++ )
     printf( "%d %d\n",
          i, power( 2, i ) );
}
```

O escopo de utilização de memória deste programa pode ser esquematizado gráficamente através de um diagrama de retângulos, tal como mostrado a seguir.



No diagrama, cada retângulo representa a memória ativada para executar uma função. Ao lado de cada variável local é colocado um espaço para preencher o seu valor. As variáveis locais em C são chamadas de automáticas pois têm alocação de memória automática.

Mecanismos de passagem de argumentos

É importante observar que em C os argumentos de uma função chamada são passados como variáveis temporárias que contêm cópias das variáveis locais da função que chamou. A este mecanismo dá-se o nome "passagem de argumentos por valor". Uma alternativa seria a "passagem de argumentos por referência", onde o endereço do argumento é passado. Isto é o que é feito em Fortran. A linguagem Pascal contem os dois mecanismos de passagem de parâmetros.

Um problema da passagem de argumentos por valor é que a função chamada não pode alterar diretamente uma variável que chamou, visto que a função chamada só recebe cópias de argumentos. Por exemplo, suponha que se deseja escrever uma rotina **swap** para trocar os valores de duas variáveis entre si. Em Fortran um programa que utilize esta rotina poderia ser:

```
subroutine swap( x, y )
integer x, y
integer temp
temp = x
x = y
y = temp
return
end
```

```
program main
integer a, b
a = 2
b = 5
call swap( a, b )
stop
end
```

Em C, a maneira errada de chamar e escrever esta rotina (função) está mostrada a seguir:

```
void swap( int x, int y )
{
   int temp;

   temp = x;
   x = y;
   y = temp;
}


void main( void )
{
   int a = 2, b = 5;

   swap( a, b );
}
```

Isto está errado porque, como os argumentos são passados por valor, a função **swap** não pode afetar os valores de **a** e **b**. Isto só seria conseguido se **a** e **b** fossem passados por referência. É óbvio que este mecanismo também pode ser implementado em C, conforme será mostrado na próxima seção.

A vantagem da passagem de argumentos por valor está na garantia de privacidade de dados de uma determinada rotina, se este for o objetivo. Quando uma rotina chama uma função passando parâmetros por valor, ela pode ter certeza de que as variáveis locais que foram usadas como parâmetros não serão alteradas pela função chamada.

Implementação de passagem de argumentos por referência em C

Em C, o endereço (posição na memória) de uma variável pode ser acessado explicitamente. Por exemplo, seja o número inteiro **int a;**. O indentificador **&a** é identificado pelo compilador como o endereço da variável **a**.

Também é permitido a criação de variáveis que contêm endereços de outras variáveis. São os chamados ponteiros, que são declarados por exemplo como **`int *p;`**. Neste caso, **`p`** é um ponteiro para um inteiro. Isto quer dizer que o valor de **`p`** é um endereço na memória de um inteiro. Por exemplo, para um inteiro **`a`**, a instrução **`p = &a;`** atribui a **`p`** o endereço de **`a`**.

Além disso, existe a possibilidade de acessar o conteúdo do endereço que é "apontado" por um ponteiro. Por exemplo, considere o programa abaixo:

```
void main( void )
{
   int a = 10;
   int *p;

   p = &a;
   printf( "%d\n", *p );
}
```

Este programa imprime o valor 10 na tela, pois o identificador **`*p`** é identificado pelo compilador como o conteúdo do endereço que é identificado por **`p`**.

Com base no que foi exposto acima, a implementação da passagem de argumentos por referência em C pode ser feita facilmente. Isto é, como se tem um acesso explícito ao endereço de uma variável, quando se deseja passar um argumento por referência (para alterar o valor da variável correspondente na função chamada, por exemplo), basta que se passe (por valor) o endereço (ou um ponteiro contendo o endereço) da variável.

A implementação correta da função **`swap`** para trocar os valores de duas variáveis entre si está mostrada abaixo:

```
void swap( int *px, int *py )
{
   int temp;

   temp = *px;
   *px = *py;
   *py = temp;
}

void main( void )
{
   int a = 2, b = 5;

   swap( &a, &b );
}
```

A princípio a implementação deste mecanismo de passagem de argumento por referência em C pode parecer complicada, principalmente se o programador estiver acostumado com Fortran, onde não se tem um acesso explícito a endereços de variáveis e não existe variáveis ponteiros para outras variáveis. No entanto, a lógica da linguagem C é consistente (sempre passa argumentos por valor, mesmo que sejam endereços de variáveis) e consegue implementar os dois mecanismos de passagem de argumentos.

Simulação de passagem de argumentos por diagrama de retângulos

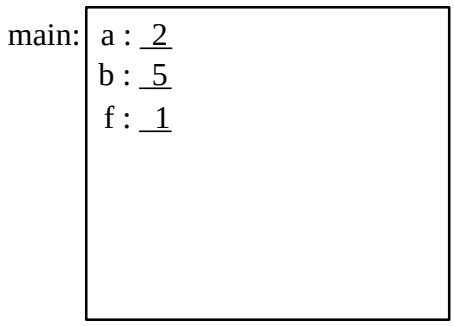
O objetivo desta seção é descrever por diagramas de retângulos o escopo de utilização de memória e o mecanismo de passagem de argumentos durante a execução de um programa. Uma extensão do exemplo anterior da função **`swap`** é utilizada para descrever esta execução. A extensão é tal que a troca dos valores das variáveis só é feita se um terçeiro parâmetro dado (por valor) for positivo. Este programa é mostrado abaixo:

```
void swap( int f, int *px, int *py )
{
   int temp;

   if( f > 0 )
   {
L2:       temp = *px;
L3:       *px = *py;
L4:       *py = temp;
L5:}
}

void main( void )
{
   int a = 2, b = 5, f = 1;

L1:  swap( f, &a, &b );
L6:
}
```

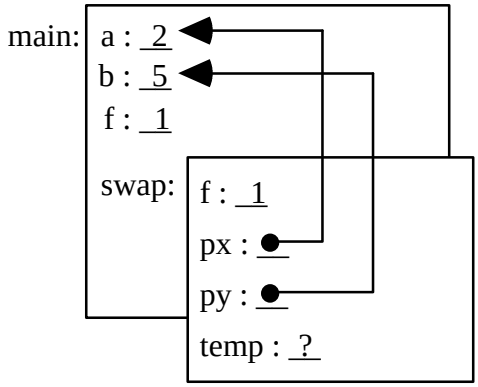
No diagrama de retângulos, o ativamento de uma função é representado por um retângulo. As variáveis locais são identificadas por um identificador que representa a memória utilizada pela variável. As variáveis locais que são ponteiros também têm setas que apontam para os identificadores correspondentes. A seguir são mostrados diagramas para a execução do programa acima imediatamente antes de cada

uma das instruções com rótulos (“labels” L1, L2, L3, L4, L5 e L6).

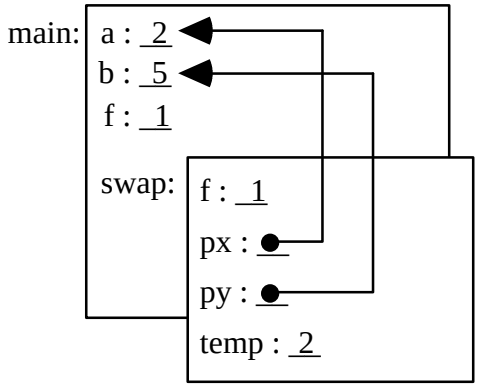
Antes de L1:



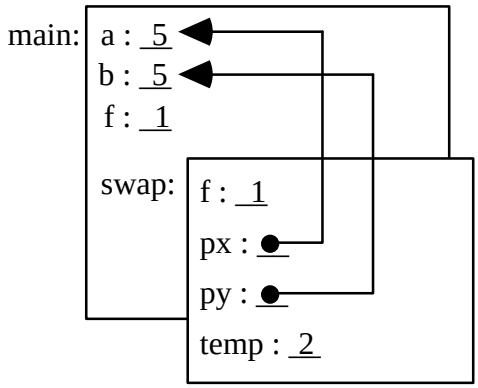
Antes de L2:



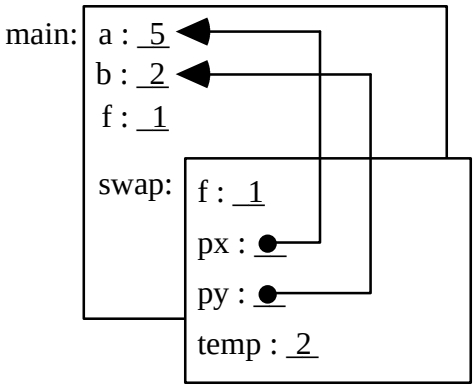
Antes de L3:



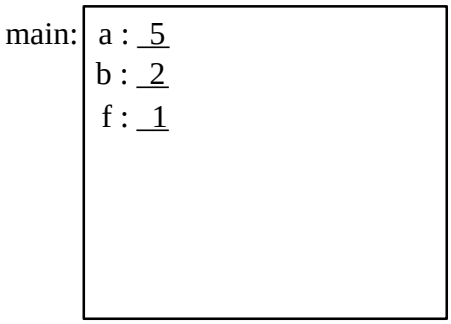
Antes de L4:



Antes de L5:



Antes de L6:



Variáveis externas

Um programa em C consiste de um conjunto de objetos externos, que podem ser tanto variáveis quanto funções. O adjetivo “externo” é utilizado para diferenciar de objetos “internos” a uma função, que são os seus parâmetros (cópias de variáveis passadas) e variáveis locais, chamadas de variáveis automáticas pois têm alocação de memória automática.

Variáveis externas são definidas fora de qualquer função, e são portanto potencialmente disponíveis para muitas funções (variáveis externas são semelhantes às variáveis definidas dentro de uma instrução COMMON de Fortran). As funções são sempre externas, pois C não permite uma função ser definida dentro de outra função.

A alocação de variáveis externas é estática, isto é, existe um espaço de memória reservado para cada variável externa já na hora que o programa é ativado.

As variáveis externas, porque são acessadas globalmente, fornecem um meio alternativo para a comunicação entre as funções. Assim,

no exemplo da troca de valores de variáveis, a implementação poderia ser:

```
/* Variáveis externas
 */
int  a = 2;
int  b = 5;

void swap_ab( void )
{
   int temp;

   temp = a;
   a = b;
   b = temp;
}

void main( void )
{
   swap_ab( );
}
```

Uma variável externa pode ser compartilhada por funções que estão localizadas em módulos de compilação distintos (arquivos fontes distintos). Para isso, ela deve ser definida em um módulo e declarada em outros módulos que a utilizem, o que é feito usando a instrução **extern**. Exemplos disso podem ser:

```
extern int  a;
extern int  b;
```

Variáveis estáticas

Além da variável automática (alocada automaticamente na memória quando a sua função é acionada) e da variável externa, existe uma terceira classe que se chama variável estática. Variáveis estáticas podem tanto ser internas quanto externas.

Variáveis estáticas internas são locais para uma determinada função tanto quanto são as variáveis automáticas, mas, diferentemente destas últimas, elas permanecem existentes na memória ao invés de irem e virem cada vez que a função é ativada. A declaração de uma variável estática interna difere da declaração de uma variável automática apenas pela instrução **static** que precede o tipo da variável:

```
static  int  a;
```

Variáveis estáticas externas são locais para um módulo de compilação (arquivo fonte), isto é, só são visíveis para as funções que compõem o módulo. Elas diferem das variáveis externas globais (que são visíveis para todas as funções de todos os módulos de compilação) pela instrução **static** que precede o tipo da variável.

Funções estáticas

Funções estáticas são funcões que são visíveis apenas por um módulo de compilação. Somente as funções do módulo podem acionar (chamar) uma função estática. A vantagem de funções estáticas é que elas são privadas ao seu módulo de compilação, podendo até repetir nomes de outras funções estáticas de outros módulos.

Em C, **static** significa não somente permanência em memória, mas também um certo grau de "privacidade": variáveis estáticas internas são visíveis apenas por uma função; e variáveis estáticas externas e funções estáticas são visíveis apenas por um módulo de compilação. O uso de funções estáticas e variáveis estáticas externas auxilia a modularização de um programa pois aumenta o grau de independência de um módulo em relação aos demais.

Exercícios propostos

1. Desenhe uma seqüência de diagramas de retângulos para indicar o estado de execução do programa abaixo imediatamente antes de cada instrução com rótulo Li:. Indique nestes diagramas um identificador para a variável externa que aparece, com o valor correspondente para cada estado.

```
int i;

void fncA( void )
{
L5: i = 13;
}

void fncB( void )
{
L4: fncA();
}

void fncC( void )
{
L3: fncB();
}

void main( void )
{
L1: i = 12;
L2: fncC();
L6:
}
```

2. Execute os programas abaixo e mostre uma seqüência de diagramas de retângulos, com identificadores para as variáveis e o argumento de função. Os diagramas devem mostrar o estado dos programas antes de cada instrução com rótulo Li:. Coloque os valores correspondentes para os identificadores.

```
int x, y;
void fnc(int a)
{
L1: a = 12;
L2: y = x;
}
void main(void)
{
    x = 4;
L0: fnc( x );
L3:
}
```

```
int x, y;
void fnc(int *a)
{
L1: *a = 12;
L2: y = x;
}
void main(void)
{
    x = 4;
L0: fnc( &x );
L3:
}
```

3. Execute o programa abaixo e mostre uma seqüência de diagramas de retângulos, com identificadores para as variáveis e os argumentos de função. Os diagramas devem mostrar o estado do programa antes de cada instrução com rótulo Li:. Coloque os valores correspondentes para os identificadores. Este exercício foi projetado para forçá-lo(a) a entender o mecanismo de utilização de memória e de passagem de argumentos de funções em C, mesmo em face à chamada recursiva de funções.

```
void fnc( int *y, int z )
{
    int x;

    if( z <= 1 )
L2:     *y = 16;
    else
    {
        x = z – 2;
L1:     fnc( &x, *y – 4 );
    }
L3:
}

void main( void )
{
    int x = 6;

L0: fnc( &x, x – 3 );
L4:
}
```

Estruturas e Alocação Dinâmica de Memória em C

Luiz Fernando Martha

Referência Básica

- Kernighan, B.W.; Ritchie, D.M., *The C Programming Language*, Prentice-Hall, Second Edition, 1988.

Estruturas

Uma estrutura (**struct**) é uma coleção de uma ou mais variáveis, possivelmente de tipos diferentes, agrupadas segundo um único nome para facilidade de manuseio. Um exemplo de uma declaração de uma estrutura seria:

```
struct data {
   int    dia;
   int    mes;
   int    ano;
   int    dia_ano;
   char   nome_mes[4];
   };
```

A definição de uma variável do tipo estrutura pode ser feita como exemplificada abaixo:

```
/* sem inicialização
 */
struct data  d;

/* com inicialização
 */
struct data d = {4,7,1776,186,"Jul"};
```

Alternativamente, pode-se criar um tipo para esta estrutura:

```
struct data {
   int    dia;
   int    mes;
   int    ano;
   int    dia_ano;
   char   nome_mes[4];
   };

/* criação do tipo Data
 */
typedef struct data  Data;

/* definição de uma variável deste
 * tipo
 */
Data  d;
```

Pode-se inclusive declarar a estrutura e definir um tipo para ela de uma só vez:

```
typedef struct data {
   int    dia;
   int    mes;
   int    ano;
   int    dia_ano;
   char   nome_mes[4];
   } Data;
```

O acesso a um campo (membro) de uma estrutura é feito da seguinte maneira:

```
int   t;
Data  d;

t = d.dia;
```

Estruturas também podem ser declaradas dentro de outras estruturas, tal como mostrado abaixo:

```
struct pessoa {
   char         nome[81];
   struct data  nasc;
   };
```

E o acesso a um campo desta estrutura pode ser exemplificado como:

```
struct pessoa  empreg;

t = empreg.nasc.dia;

/* equivalente a:
 t = (empreg.nasc).dia;
 */
```

Ponteiros para estruturas

Considere o problema de escrever uma função para preencher os campos de uma estrutura. Existem duas alternativas para se passar uma estrutura como parâmetro para esta função: ou passa-se por valor ou por referência. Neste caso, não faz sentido passar a estrutura por valor pois seria passado uma cópia da estrutura e o preenchimento dos seus campos pela função não surtiria efeito. A maneira correta é passar a estrutura por referência, isto é, passar o endereço (ponteiro) da estrutura. Isto é exemplificado abaixo.

```
void preenche( Data *pd )
{
 pd->dia      = 4;
 pd->mes      = 7;
 pd->ano      = 1776;
 pd->dia_ano  = 186;
 pd->nome_mes = "Jul";
}

void main( void )
{
 Data  d;

 preenche( &d );
}
```

Neste caso **pd** é um ponteiro para uma estrutura do tipo **Data**. O acesso de um campo da estrutura, cujo ponteiro é dado, é exemplificado por **pd->ano**, o que é equivalente a **(*pd).ano**.

Uma estrutura também pode conter campos que são ponteiros para estruturas. Um exemplo disto é:

```
typedef struct pessoa {
   char         nome[81];
   struct data  *nasc;
   } Pessoa;
```

Neste caso, a estrutura **Pessoa** tem dois campos, um é o nome da pessoa e o outro é um ponteiro para a data do seu nascimento.

A função abaixo mostra um exemplo do acesso a um campo da data de nascimento de uma pessoa cujo registro é passado (por referência):

```
void acessa_dia( Pessoa *empreg )
{
 int t;

 t = empreg->nasc->dia;

/* isto é equivalente a:
 t = (empreg->nasc)->dia;
 */
}
```

A declaração de uma estrutura também pode ser auto-recursiva. Isto está exemplificado abaixo para a declaração e definição de tipo de um elemento de lista que aponta para o elemento seguinte (do seu mesmo tipo) na lista:

```
typedef struct lista {
   struct lista  *next;
   int           dados;
   } ListElem;
```

Vetor de estruturas

Assim como pode-se declarar um vetor de caracteres, ou um vetor de inteiros, pode-se também declarar um vetor de estruturas:

```
/* Declaração estática:
 */
struct pessoa  staff[100];

/* ou ainda:
 */
Data  dias[366];
```

E a indexação é feita da mesma forma que é feita para qualquer outro vetor:

```
/* Atrubui a "d" todos os valores
 * do vigésimo (índice = 19) dia do
 * vetor "dias":
 */
Data  d;

d = dias[19];
```

Alocação dinâmica de estruturas

A linguagem C possibilita a requisição de memória para variáveis de uma forma dinâmica em tempo de execução. Isto é feito através de chamadas a funções que são disponíveis em bibliotecas padrões que acompanham todos os compiladores C.

O modelo de utilização das funções para alocação dinâmica de memória é o seguinte:

- Toda vez que o programa precisa de um espaço na memória para uma variável (que pode ser uma estrutura) ou um vetor, ele chama uma função de alocação de memória passando o tamanho em palavras ("bytes") da memória necessária.
- O sistema operacional verifica se existe o tanto de memória disponível em uma área contígua e retorna, como valor de retorno da função chamada, o endereço (ponteiro) onde inicia a porção de memória requisitada. Se não existe memória disponível o valor retornado é nulo.
- Esta memória requisitada pelo programa pode ser liberada também em qualquer instante da execução. Isto é feito através da chamada de uma função de liberação. Isto resulta na devolução ao sistema operacional desta porção de memória para que ela possa ser utilizada por outros clientes ou em outra fase da execução.
- Existe também a possibilidade de realocação de uma memória que foi requisitada anteriormente para aumentar ou diminir o seu tamanho. O sistema operacional verifica se existe uma área de memória contígua (que sempre existirá caso a realocação for para um tamanho menor) e retorna o ponteiro para o nova área (que pode inclusive ser o mesmo endereço). Caso não exista uma área contígua, a função de realocação retorna um valor nulo. Quando há a realocação, a porção de memória que havia sido requisitada anteriormente é copiada para a nova área.

As declarações ("prototypes") das funções de alocação são as seguintes:

```
void *malloc( int total_size );
void *calloc( int n, int size );
```

A função **malloc** recebe como parâmetro o tamanho total de memória requisitada. A função **calloc** recebe um parâmetro para o número de entidades requisitadas e outro para o tamanho de cada entidade.

As declarações das funções de liberação e realocação estão mostradas abaixo:

```
void free( void *pointer_to_area );
void *realloc( void *old_area,
               int total_new_size );
```

As funções abaixo mostram um exemplo de alocação dinâmica de um vetor de estruturas:

```
typedef struct pessoa {
   char         nome[81];
   struct data  *nasc;
   } Pessoa;

static Pessoa  *pessoas;

void aloca_pessoas( int npessoas )
{
 pessoas = (Pessoa *)calloc(npessoas,
                  sizeof( Pessoa ) );
}

void libera_pessoas( void )
{
 free( pessoas );
}
```

Abaixo se encontra um exemplo de alocação dinâmica de uma lista com três elementos:

```
typedef struct lista {
   struct lista  *next;
   int           dados;
   } ListElem;

static ListElem  *list_head = NULL;

void aloca_lista( void )
{
 ListElem  *elem1, *elem2, *elem3;

 elem1 = (ListElem *)malloc(
                sizeof( ListElem ) );
 elem2 = (ListElem *)malloc(
                sizeof( ListElem ) );
 elem3 = (ListElem *)malloc(
                sizeof( ListElem ) );

 list_head = elem1;
 elem1->next = elem2;
 elem2->next = elem3;
 elem3->next = NULL;
}

void libera_lista( void )
{
 ListElem  *elem;
 while( list_head != NULL ) {
  elem = list_head;
  list_head = elem->next;
  free( elem );
 }
}
```