Curso de Programação C em Ambientes Linux – Aula 03

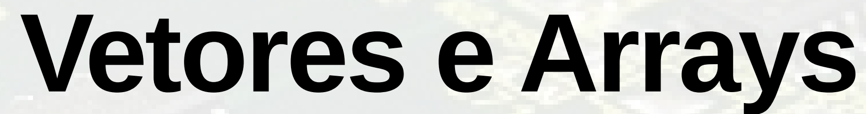
Centro de Engenharias da Mobilidade - UFSC

Professores Gian Berkenbrock e Giovani Gracioli
http://www.lisha.ufsc.br/C+language+course+resources

```
class classCar{
    protected:
    enumCarMake carMake;
    structTire carTires[4];
    classEngine carMotor;
    classPart carPartsList[100];
    public:
    classCar();
    virtual ~classCar();
    void GetCarLoc(classCarLoc& carLoc);
};

class classTruck : public classCar{
    structTire* pTires;
public:
    classTruck();
    virtual ~classTruck();
};
```

Conteúdo desta aula

- Vetores e arrays.
- Funções: declaração, comando return,
- função main, tipo void,
- passagem de parâmetros por valor e por referência,
- escopo de nomes e variáveis locais e globais,
- protótipo de função,
- recursão.



Curso de Programação C em Ambientes Linux

Aula 03 de 5 – 06/08/2014

Introdução

- Arrays ou vetores são estruturas de dados que representam itens de dados do mesmo tipo
- Um array é um conjunto de espaços de memória que se relacionam pelo fato de que todos têm o mesmo nome e o mesmo tipo

Arrays

- Para se referir a um local ou elemento em particular no array, especificamos o nome do array e o número da posição do elemento em particular no array
- Array de inteiros
  int c[12];
- c[0], c[1]..
- printf(“%d\n”, c[0] + c[1] + c[2]);

Nome do array (observe que todos os elementos desse array têm o mesmo nome, c)

Elemento	Valor
c[0]	–45
c[1]	6
c[2]	0
c[3]	72
c[4]	1543
c[5]	–89
c[6]	0
c[7]	62
c[8]	–3
c[9]	1
c[10]	6453
c[11]	78

Número da posição do elemento dentro do array c

Figura 6.1 ■ Array de 12 elementos.

Declarando arrays

- Os arrays ocupam espaço de memória
- Você especifica o tipo de cada elemento e o número de elementos exigidos por array de modo que o computador possa reservar a quantidade de memória apropriada
- A declaração

```
int c[ 12 ];
```

- é usada para pedir ao computador que reserve 12 elementos para o array de inteiros c
- int b[100], x[20];

Exemplo de arrays (1)

```
 1  /* Figura 6.3: fig06_03.c
 2     Inicializando um array */
 3  #include <stdio.h>
 4
 5  /* função main inicia a execução do programa */
 6  int main( void )
 7  {
 8     int n[ 10 ]; /* n é um array de 10 inteiros */
 9     int i; /* contador */
10
11     /* inicializa elementos do array n como 0 */
12     for ( i = 0; i < 10; i++ ) {
13        n[ i ] = 0; /* define elemento no local i como 0 */
14     } /* fim do for */
15
16     printf( "%s%13s\n", "Elemento", "Valor" );
17
18     /* saída na tela de conteúdo do array n em formato tabular */
19     for ( i = 0; i < 10; i++ ) {
20        printf( "%7d%13d\n", i, n[ i ] );
21     } /* fim do for */
22
23     return 0; /* indica conclusão bem-sucedida */
24  } /* fim do main */
```

```
Elemento        Valor
       0            0
       1            0
       2            0
       3            0
       4            0
       5            0
       6            0
       7            0
       8            0
       9            0
```

Figura 6.3 ■ Inicializando os elementos de um array com zeros.

Exemplo de arrays (2)

- Os elementos de um array também podem ser inicializados quando o array é declarado a partir de uma declaração com um sinal de igual e chaves, {}, que contenha uma lista de inicializadores separados por vírgula

```
1  /* Figura 6.4: fig06_04.c
2     Inicializando um array com uma lista de inicializadores */
3  #include <stdio.h>
4
5  /* função main inicia a execução do programa */
6  int main( void )
7  {
8     /* usa lista de inicializadores para inicializar array n */
9     int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
10    int i; /* contador */
11
12    printf( "%s%13s\n", "Elemento", "Valor" );
13
14    /* lista conteúdo do array em formato tabular */
15    for ( i = 0; i < 10; i++ ) {
16       printf( "%7d%13d\n", i, n[ i ] );
17    } /* fim do for */
18
19    return 0; /* indica conclusão bem-sucedida */
20 } /* fim do main */
```

```
Elemento        Valor
       0           32
       1           27
       2           64
       3           18
       4           95
       5           14
       6           90
       7           70
       8           60
       9           37
```

Figura 6.4 ■ Inicializando os elementos de um array com uma lista de inicializadores.

Exemplo de arrays (3)

- A declaração de array

  ```
  int n[ 5 ] = { 32, 27, 64, 18, 95, 14 };
  ```

  causa um erro de sintaxe porque existem seis inicializadores e apenas cinco elementos de array
- Se o tamanho do array for omitido de uma declaração com uma lista de inicializadores, o número de elementos no array será o número de elementos na lista de inicializadores
- Por exemplo,

  ```
  int n[] = { 1, 2, 3, 4, 5 };
  ```

  criaria um array de cinco elementos

A diretiva #define

```
 1  /* Figura 6.5: fig06_05.c
 2     Inicializa elementos do array s como inteiros pares de 2 a 20 */
 3  #include <stdio.h>
 4  #define SIZE 10 /* tamanho máximo do array */
 5
 6  /* função main inicia a execução do programa */
 7  int main( void )
 8  {
 9     /* constante simbólica SIZE pode ser usada para especificar tamanho do array */
10     int s[ SIZE ]; /* array s tem SIZE elementos */
11     int j; /* contador */
12
13     for ( j = 0; j < SIZE; j++ ) { /* define os elementos */
14        s[ j ] = 2 + 2 * j;
15     } /* fim do for */
16
17     printf( "%s%13s\n", "Elemento", "Valor" );
18
19     /* lista de impressão do conteúdo do array s em formato tabular */
20     for ( j = 0; j < SIZE; j++ ) {
21        printf( "%7d%13d\n", j, s[ j ] );
22     } /* fim do for */
23
24     return 0; /* indica conclusão bem-sucedida */
25  } /* fim do main */
```

```
Elemento        Valor
       0            2
       1            4
       2            6
       3            8
       4           10
       5           12
       6           14
       7           16
       8           18
       9           20
```

Figura 6.5 ■ Inicializando os elementos do array s com inteiros pares de 2 a 20.

Exemplo de uso (1)

- Considere um problema com o seguinte enunciado
  - Pedimos a 40 alunos que avaliassem a comida da cantina estudantil e dessem notas que fossem de 1 a 10 (1 significaria horrorosa e 10 significaria excelente). Coloque as 40 respostas em um array de inteiros e resuma os resultados da pesquisa.
- Queremos resumir o número de respostas de cada tipo (ou seja, de 1 a 10)

Exemplo de uso (2)

```
1  /* Figura 6.7: fig06_07.c
2     Programa de pesquisa com estudantes */
3  #include <stdio.h>
4  #define RESPONSE_SIZE 40 /* define tamanhos de array */
5  #define FREQUENCY_SIZE 11
6
7  /* função main inicia a execução do programa */
8  int main( void )
9  {
10    int answer; /* contador para percorrer 40 respostas */
11    int rating; /* contador para percorrer frequências 1-10 */
12
13    /* inicializa contadores de frequência em 0 */
14    int frequency[ FREQUENCY_SIZE ] = { 0 };
15
16    /* coloca as respostas da pesquisa no array responses */
17    int responses[ RESPONSE_SIZE ] = { 1, 2, 6, 4, 8, 5, 9, 7, 8, 10,
18       1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7, 5, 6, 6,
19       5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };
20
21    /* para cada resposta, seleciona valor de um elemento do array responses
22       e usa esse valor como subscrito na frequência do array para
23       determinar elemento a ser incrementado */
```

Exemplo de uso (30

```
24      for ( answer = 0; answer < RESPONSE_SIZE; answer++ ) {
25         ++frequency[ responses [ answer ] ];
26      } /* fim do for */
27
28      /* mostra resultados */
29      printf( "%s%17s\n", "Avaliação", "Frequência" );
30
31      /* listas de impressão das frequências em um formato tabular */
32      for ( rating = 1; rating < FREQUENCY_SIZE; rating++ ) {
33         printf( "%6d%17d\n", rating, frequency[ rating ] );
34      } /* fim do for */
35
36      return 0; /* indica conclusão bem-sucedida */
37   } /* fim do main */
```

```
Avaliação          Frequência
1                  2
2                  2
3                  2
4                  2
5                  5
6                  11
7                  5
8                  7
9                  1
10                 3
```

Figura 6.7 ■ Programa de análise de pesquisa de alunos.

Atenção ao acessar elementos

- C não faz a verificação dos limites do array para impedir que o programa se refira a um elemento que não existe
- Assim, um programa em execução pode ultrapassar o final de um array sem aviso e não causar nenhum erro visível no momento
- Você deverá garantir que todas as referências de array permaneçam dentro dos limites do array

Pesquisando um valor em um array

- Talvez seja necessário determinar se um array contém um valor que combina com certo valor de chave
- O processo de encontrar determinado elemento de um array é chamado pesquisa
- A pesquisa linear compara cada elemento do array com a chave de pesquisa.
- Como o array não está em uma ordem em particular, o valor pode ser encontrado tanto no primeiro elemento quanto no último

Pesquisando um valor em um array

```
1  /* Figura 6.18: fig06_18.c
2     Pesquisa linear de um array */
3  #include <stdio.h>
4  #define SIZE 100
5
6  /* protótipo de função */
7  int linearSearch( const int array[], int key, int size );
8
9  /* função main inicia a execução do programa */
10 int main( void )
11 {
12    int a[ SIZE ]; /* cria array a */
13    int x; /* contador para inicializar elementos 0-99 do array a */
14    int searchKey; /* valor para localizar no array a */
```

Figura 6.18 ■ Pesquisa linear de um array. (Parte 1 de 2.)

Pesquisando um valor em um array

```
15     int element; /* variável para manter local de searchKey or -1 */
16
17     /* criar dados */
18     for ( x = 0; x < SIZE; x++ ) {
19        a[ x ] = 2 * x;
20     } /* fim do for */
21
22     printf( "Digite chave de pesquisa de inteiro:\n" );
23     scanf( "%d", &searchKey );
24
25     /* tenta localizar searchKey no array a */
26     element = linearSearch( a, searchKey, SIZE );
27
28     /* exibe resultados */
29     if ( element != -1 ) {
30        printf( "Valor encontrado no elemento %d\n", element );
31     } /* fim do if */
32     else {
33        printf( "Valor não encontrado\n" );
34     } /* fim do else */
35
36     return 0; /* indica conclusão bem-sucedida */
37  } /* fim do main */
38
```

Pesquisando um valor em um array

```
39  /* Compara chave com cada elemento do array até o local ser encontrado
40     ou até o final do array ser alcançado; retorna subscrito do elemento
41     se chave foi encontrada ou -1 se chave não encontrada */
42  int linearSearch( const int array[], int key, int size )
43  {
44     int n; /* contador */
45
46     /* loop pelo array */
47     for ( n = 0; n < size; ++n ) {
48
49        if ( array[ n ] == key ) {
50           return n; /* retorna local da chave */
51        } /* fim do if */
52     } /* fim do for */
53
54     return -1; /* chave não encontrada */
55  } /* fim da função linearSearch */
```

```
Digite chave de pesquisa de inteiro:
36
Valor encontrado no elemento 18
```

```
Digite chave de pesquisa de inteiro:
37
Valor não encontrado
```

Figura 6.18 ■ Pesquisa linear de um array. (Parte 2 de 2.)

Arrays multidimensionais (1)

- Os arrays de subscritos múltiplos (também chamados arrays multidimensionais) podem ser usados na representação de tabelas de valores que consistem em informações organizadas em linhas e colunas
- Para identificar determinado elemento de tabela, devemos especificar dois subscritos: o primeiro (por convenção) identifica a linha do elemento, e o segundo (por convenção) identifica a coluna do elemento
- Exemplo:
  int a[3][4];

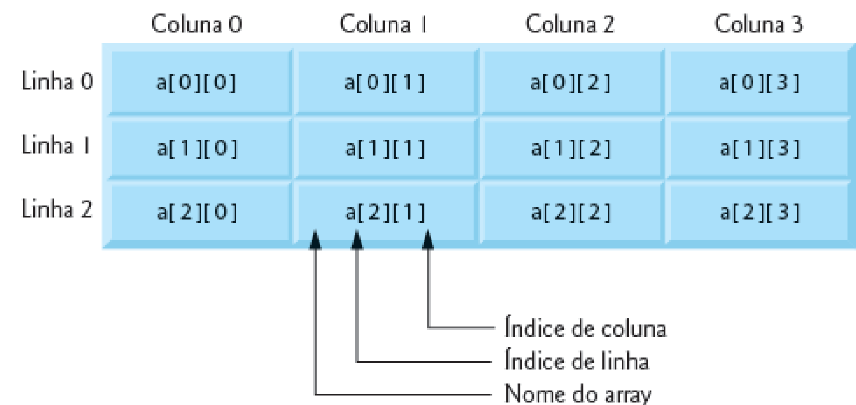


Figura 6.20 ■ Array de subscrito duplo com três linhas e quatro colunas.

Arrays multidimensionais (2)

- Um array de subscritos múltiplos pode ser inicializado quando definido, assim como um array de único subscrito.

```
int b[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };
```

- Os valores são agrupados por linha em chaves
- Os valores no primeiro conjunto de chaves inicializam a linha 0, e os valores no segundo conjunto de chaves inicializam a linha 1

Arrays multidimensionais (3)

- A estrutura de repetição for é utilizada para percorrer as linhas e colunas dos arrays

```
total= 0;

for ( linha = 0; linha <= 2; linha++ ){
  for ( coluna = 0; coluna <= 3; coluna++ ){
    total+= a[ linha ][ coluna ];
  }
}
```

Exemplo de uso (1)

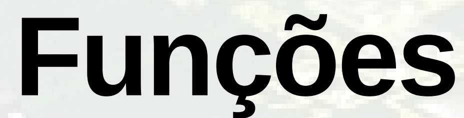
```c
1  /* Figura 6.21: fig06_21.c
2     Inicializando arrays multidimensionais */
3  #include <stdio.h>
4
5  void printArray( const int a[][ 3 ] ); /* protótipo de função */
6
7  /* função main inicia a execução do programa */
8  int main( void )
9  {
10    /* inicializa array1, array2, array3 */
11    int array1[ 2 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 } };
12    int array2[ 2 ][ 3 ] = { 1, 2, 3, 4, 5 };
13    int array3[ 2 ][ 3 ] = { { 1, 2 }, { 4 } };
14
15    printf( "Valores em array1 por linha são:\n" );
16    printArray( array1 );
17
18    printf( "Valores em array2 por linha são:\n" );
19    printArray( array2 );
20
21    printf( "Valores em array3 por linha são:\n" );
22    printArray( array3 );
23    return 0; /* indica conclusão bem-sucedida */
24 } /* fim do main */
25
```

Exemplo de uso (2)

```
26 /* função para mostrar array com duas linhas e três colunas */
27 void printArray( const int a[][ 3 ] )
28 {
29    int i; /* contador de linha */
30    int j; /* contador de coluna */
31
32    /* loop pelas linhas */
33    for ( i = 0; i <= 1; i++ ) {
34
35       /* imprime valores nas colunas */
36       for ( j = 0; j <= 2; j++ ) {
37          printf( "%d ", a[ i ][ j ] );
38          } /* fim do for interno */
39
40       printf( "\n" ); /* inicia nova linha de resultados */
41    } /* fim do for externo */
42 } /* fim da função printArray */
```

```
Os valores em array1 por linha são:
1 2 3
4 5 6
Os valores em array2 por linha são:
1 2 3
4 5 0
Os valores em array3 por linha são:
1 2 0
4 0 0
```

Figura 6.21 ■ Inicializando arrays multidimensionais.



Curso de Programação C em Ambientes Linux

Aula 03 de 5 – 06/08/2014

Introdução

- Módulos ou funções são a base da programação estruturada em C
- Os programas em C normalmente são escritos combinando-se novas funções (escritas por vocês) com funções “pré-definidas” disponíveis na biblioteca padrão de C
- Exemplos
  - Funções para cálculos matemáticos
  - Manipulação de string e caracteres
  - Entrada e saída

Ex: Funções Matemáticas

Função	Descrição	Exemplo
`sqrt( x )`	raiz quadrada de ***x***	`sqrt( 900,0 )` é 30,0 `sqrt( 9,0 )` é 3,0
`exp( x )`	função exponencial e^x	`exp( 1,0 )` é 2,718282 `exp( 2,0 )` é 7,389056
`log( x )`	logaritmo natural de ***x*** (base ***e***)	`log( 2,718282 )` é 1,0 `log( 7,389056 )` é 2,0
`log10( x )`	logaritmo de ***x*** (base 10)	`log10( 1,0 )` é 0,0 `log10( 10,0 )` é 1,0 `log10( 100,0 )` é 2,0
`fabs( x )`	valor absoluto de ***x***	`fabs( 13,5 )` é 13,5 `fabs( 0,0 )` é 0,0 `fabs( -13,5 )` é 13,5
`ceil( x )`	arredonda ***x*** ao menor inteiro não menor que ***x***	`ceil( 9,2 )` é 10,0 `ceil( -9,8 )` é -9,0
`floor( x )`	arredonda ***x*** ao maior inteiro não maior que ***x***	`floor( 9,2 )` é 9,0 `floor( -9,8 )` é -10,0
`pow( x, y )`	***x*** elevado à potência ***y*** (x^y)	`pow( 2, 7 )` é 128,0 `pow( 9, 0,5 )` é 3,0
`fmod( x, y )`	módulo (resto) de ***x/y*** como um número em ponto flutuante	`fmod( 13,657, 2,333 )` é 1,992
`sin( x )`	seno trigonométrico de ***x*** (***x*** em radianos)	`sin( 0,0 )` é 0,0
`cos( x )`	cosseno trigonométrico de ***x*** (***x*** em radianos)	`cos( 0,0 )` é 1,0
`tan( x )`	tangente trigonométrica de ***x*** (***x*** em radianos)	`tan( 0,0 )` é 0,0

Figura 5.2 ■ Funções da biblioteca matemática comumente usadas.

Importância de Funções

- Todas as variáveis declaradas dentro de uma função são variáveis locais
- Reutilização de software
  - Uso de funções existentes para montar novos programas
- Repetição de código em um programa
  - Permite a execução do mesmo trecho de código em diferentes locais do programa
- Abstração
  - Boa nomeação e definição de função

Definição de função

- O formato de uma definição de função é
  - Tipo-valor-retorno nome-da-função (lista de parâmetros)

```
{
    instruções
    return valor de retorno se houver;
}
```

- Exemplo: função para soma de dois números inteiros

```
int soma(int a, int b) {
    return a + b;
}
```

Protótipos de Função

- O protótipo de função diz ao compilador o tipo de dado retornado, o número de parâmetros que a função espera receber, os tipos dos parâmetros e a ordem em que esses parâmetros são esperados
- O compilador utiliza os protótipos de função para validar as chamadas

Protótipo x Definição

- Protótipo de uma função que retorna o valor máximo entre 3 inteiros

```
int maximum( int x, int y, int z );
```

- O protótipo indica que maximum utiliza 3 argumentos do tipo *int* e retorna um resultado do tipo *int*
- Se a função for chamada com outros parâmetros, um erro irá ocorrer
- Diferença entre o protótipo (declaração) e a definição (implementação)

Protótipo de Funções

- Se não há protótipo de função para uma função, o compilador forma seu próprio protótipo, usando a primeira ocorrência da função – ou a definição de função ou uma chamada para a função
- Um protótipo de função colocado fora de qualquer definição de função se aplica a todas as chamadas após o protótipo

Passando arrays para funções (1)

- Todos os arrays, exceto o array de char, não possuem um finalizador especial
- Sendo assim, uma função que recebe um array, também deve conhecer o tamanho do array recebido

Passando arrays para funções (2)

- C passa arrays a funções por referência automaticamente — as funções chamadas podem modificar os valores de elemento nos arrays originais das funções que os utilizam
- O nome do array é avaliado como o endereço do primeiro elemento do array
- Visto que o endereço inicial do array é passado, a função chamada sabe exatamente onde o array está armazenado

Passando arrays para funções (3)

- Exemplo: imprimindo o endereço de memória de um array de char

```
 1  /* Figura 6.12: fig06_12.c
 2     O nome do array é o mesmo que o endereço de &array[ 0 ] */
 3  #include <stdio.h>
 4
 5  /* função main inicia a execução do programa */
 6  int main( void )
 7  {
 8     char array[ 5 ]; /* define um array de tamanho 5 */
 9
10     printf( "  array = %p\n&array[0] = %p\n   &array = %p\n",
11        array, &array[ 0 ], &array );
12     return 0; /* indica conclusão bem-sucedida */
13  } /* fim do main */
```

```
array = 0012FF78
&array[0] = 0012FF78
&array = 0012FF78
```

Figura 6.12 ■ O nome do array é o mesmo que o endereço do primeiro elemento do array.

Passando arrays para funções (4)

- Para uma função receber um array por meio de uma chamada de função, a lista de parâmetros da função precisa mencionar que um array será recebido.

```
void modifyArray( int b[], int size )
```

Passando arrays para funções (4)

- Exemplo: passando um array e um inteiro

```
1  /* Figura 6.13: fig06_13.c
2     Passando arrays e elementos individuais do array para funções */
3  #include <stdio.h>
4  #define SIZE 5
5
6  /* protótipos de função */
7  void modifyArray( int b[], int size );
8  void modifyElement( int e );
9
10 /* função main inicia a execução do programa */
11 int main( void )
12 {
13    int a[ SIZE ] = { 0, 1, 2, 3, 4 }; /* inicializa a */
14    int i; /* contador */
15
16    printf( "Efeitos da passagem do array inteiro por referência:\n\nOs "
17       "valores o array original são:\n" );
18
19    /* imprime na tela array original */
20    for ( i = 0; i < SIZE; i++ ) {
21       printf( "%3d", a[ i ] );
22    } /* fim do for */
23
24    printf( "\n" );
25
26    /* passa array a um modifyArray por referência */
27    modifyArray( a, SIZE );
```

Figura 6.13 ■ Passagem de arrays e de elementos individuais do array para funções. (Parte 1 de 2.)

Passando arrays para funções (5)

```
28
29    printf( "Os valores do array modificado são:\n" );
30
31    /* array modificado na saída */
32    for ( i = 0; i < SIZE; i++ ) {
33       printf( "%3d", a[ i ] );
34    } /* fim do for */
35
36    /* valor de saída de a[ 3 ] */
37    printf( "\n\n\nEfeitos de passar elemento do array "
38       "por valor:\n\nO valor de a[3] é %d\n", a[ 3 ] );
39
40    modifyElement( a[ 3 ] ); /* passa elemento do array a[ 3 ] por valor */
41
42    /* mostra valor de a[ 3 ] */
43    printf( "O valor de a[ 3 ] é %d\n", a[ 3 ] );
44    return 0; /* indica conclusão bem-sucedida */
45 } /* fim do main */
46
47 /* na função modifyArray, "b" aponta para o array original "a"
48    na memória */
49 void modifyArray( int b[], int size )
50 {
51    int j; /* contador */
52
53    /* multiplica cada elemento do array por 2 */
54    for ( j = 0; j < size; j++ ) {
55       b[ j ] *= 2;
56    } /* fim do for */
57 } /* fim da função modifyArray */
58
59 /* na função modifyElement, "e" é uma cópia local do elemento
60    do array a[ 3 ] passado de main */
61 void modifyElement( int e )
62 {
63    /* multiplica parâmetro por 2 */
64    printf( "Valor em modifyElement é %d\n", e *= 2 );
65 } /* fim da função modifyElement */
```

```
Efeitos da passagem do array inteiro por referência:

Os valores do array original são:
 0 1 2 3 4
Os valores do array modificado são:
 0 2 4 6 8

Efeitos da passagem do elemento do array por valor:
O valor de a[3] é 6
Valor em modifyElement é 12
O valor de a[ 3 ] é 6
```

Figura 6.13 ■ Passagem de arrays e de elementos individuais do array para funções. (Parte 2 de 2.)

Chamando funções por valor e por referência (1)

- Quando os argumentos são passados por valor, uma cópia do valor do argumento é feita e passada para a função chamada
- As mudanças na cópia não afetam o valor original
- Quando um argumento é passado por referência, o chamador permite que a função chamada modifique o valor da variável original
- A chamada por valor deverá ser usada sempre que a função chamada não precisar modificar o valor da variável original

Chamando funções por valor e por referência (2)

- Em C, todas as chamadas são feitas por valor, a não ser que explicitamente passadas por referência
- Entretanto, vetores e arrays são automaticamente passados por referência
- Exemplo
  soma(a, b); //passando arg. por valor

  soma(&a, &b); //passando arg. por referência, a e b passam a ser ponteiros

Recursão

- Uma função recursiva é uma função que chama a si mesma direta ou indiretamente
- A função recursiva sabe somente como resolver os casos mais simples (casos básicos) de um problema
- Se a função é chamada para resolver um caso básico, ela simplesmente retorna

Exemplo: fatorial de um Nº

- O fatorial (n!) de um nº é:
  - N * (N - 1) * (N – 2) * … * 1
- Por exemplo, 5! é o produto 5 * 4 * 3 * 2 * 1 = 120
- O fatorial de um inteiro maior ou igual a 0 pode ser calculado iterativamente usando for:

```
fatorial = 1;

for(contador = número; contador >= 1; contador--)
    fatorial *= contador;
```

Exemplo: fatorial de um Nº

- Uma definição recursiva da função fatorial é obtida observando-se o seguinte relacionamento:
  - N! = N * (N – 1)!
- Por exemplo, 5! é igual a 5 * 4!:
  - 5 * (4 * 3 * 2 * 1)

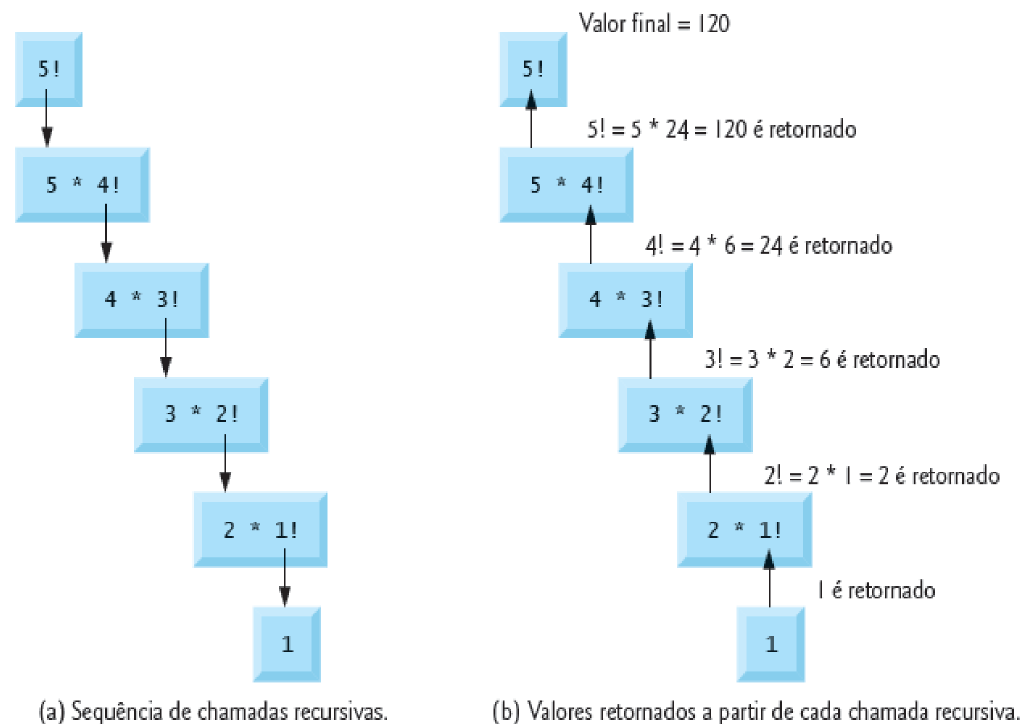


Figura 5.13 ■ Avaliação recursiva de 5!.

Exemplo: fatorial de um Nº

```
1  /* Fig. 5.14: fig05_14.c
2     Função recursiva fatorial */
3  #include <stdio.h>
4
5  long factorial( long number ); /* protótipo de função */
6
7  /* função main inicia a execução do programa */
8  int main( void )
9  {
10    int i; /* contador */
11
12    /* loop 11 vezes; durante cada iteração, calcula
13       fatorial( i ) e mostra o resultado */
14    for ( i = 0; i <= 10; i++ ) {
15       printf( "%2d! = %ld\n", i, fatorial( i ) );
16    } /* fim do for */
17
18    return 0; /* indica conclusão bem-sucedida */
19 } /* fim do main */
20
21 /* definição recursiva da função fatorial */
22 long factorial( long number )
23 {
24    /* caso básico */
25    if ( number <= 1 ) {
26       return 1;
27    } /* fim do if */
28    else { /* etapa recursiva */
29       return ( number * factorial( number - 1 ) );
30    } /* fim do else */
31 } /* fim da função fatorial */
```

Figura 5.14 ■ Calculando fatoriais com uma função recursiva. (Parte 1 de 2.)

Exemplo: fatorial de um Nº

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

Figura 5.14 ■ Calculando fatoriais com uma função recursiva. (Parte 2 de 2.)

Recursão x Iteração (1)

- Ambas se baseiam em uma estrutura de controle: a iteração usa repetição; a recursão usa uma estrutura de seleção
- Ambas envolvem repetição: a iteração usa explicitamente uma estrutura de repetição; a recursão consegue a repetição por meio de chamadas de função repetitivas

Recursão x Iteração (2)

- Tanto uma quanto a outra envolvem testes de término: a iteração termina quando a condição de continuação do loop falha; a recursão termina quando um caso básico é reconhecido
- A iteração continua a modificar um contador até que ele passe a ter um valor que faça com que a condição de continuação do loop falhe; a recursão continua a produzir versões mais simples do problema original até que o caso básico seja alcançado

Recursão x Iteração (3)

- Ambas podem ocorrer infinitamente: um loop infinito ocorre com iteração se o teste de continuação do loop nunca se tornar falso; a recursão infinita ocorre se a etapa de recursão não reduzir o problema a cada vez, de maneira que ele convirja para o caso básico
- A recursão tem muitos pontos negativos. Ela chama o mecanismo repetidamente, e, por conseguinte, também gera um overhead (sobrecarga) com as chamadas de função

Recursão x Iteração (4)

- Cada chamada recursiva faz com que outra cópia da função (na realidade, apenas as variáveis da função) seja criada; isso pode consumir uma memória considerável
- A iteração normalmente ocorre dentro de uma função, de modo que o overhead de chamadas de função repetidas e a atribuição extra de memória sejam omitidos
- Logo, por que escolher a recursão?
- Quando espelha o problema mais naturalmente e resulta em um programa mais fácil

Escopo de variáveis

- Podem existir diferentes escopos relacionados com as variáveis
- **Locais**: declaradas dentro de uma função ou bloco. A cada chamada de função, o valor da variável é iniciado
- **Globais**: pode ser usada em qualquer parte do programa
- **Estáticas**: se declarada dentro de uma função, o seu valor é iniciado apenas na primeira chamada da função

Exemplo: escopo

```
1  /* Fig. 5.12: fig05_12.cC
2     Um exemplo de escopo */
3  #include <stdio.h>
4
5  void useLocal( void ); /* protótipo de função */
6  void useStaticLocal( void ); /* protótipo de função */
7  void useGlobal( void ); /* protótipo de função */
8
9  int x = 1; /* variável global */
10
11 /* função main inicia a execução do programa */
12 int main( void )
13 {
14    int x = 5; /* variável local para main */
15
16    printf(“x local no escopo externo de main é %d\n”, x );
17
18    { /* inicia novo escopo */
19       int x = 7; /* variavel local para novo escopo */
20
21       printf( “x local no escopo interno de main é %d\n”, x );
22    } /* fim do novo escopo */
23
24    printf( “x local no escopo externo de main é %d\n”, x );
25
26    useLocal(); /* useLocal tem x local automática */
27    useStaticLocal(); /* useStaticLocal tem x local estática */
28    useGlobal(); /* useGlobal usa x global */
29    useLocal(); /* useLocal reinicializa x local automática */
30    useStaticLocal(); /* x local estática retém seu valor anterior */
31    useGlobal(); /* x global também retém seu valor */
32
33    printf( “\nx local em main é %d\n”, x );
34    return 0; /* indica conclusão bem-sucedida */
35 } /* fim do main */
36
```

Exemplo: escopo

```
37 /* useLocal reinicializa variável local x durante cada chamada */
38 void useLocal( void )
39 {
40    int x = 25; /* inicializada toda vez que useLocal é chamada */
41
42    printf( "\nx local em useLocal é %d após entrar em useLocal\n", x );
43    x++;
44    printf( "x local em useLocal é %d antes de sair de useLocal\n", x );
45 } /* fim da função useLocal */
46
47 /* useStaticLocal inicializa variável local estática x somente na
48    primeira vez em que essa função é chamada; o valor de x é
49    salvo entre as chamadas a essa função */
50 void useStaticLocal( void )
```

Figura 5.12 ■ Exemplo de escopo. (Parte 1 de 2.)

Exemplo: escopo

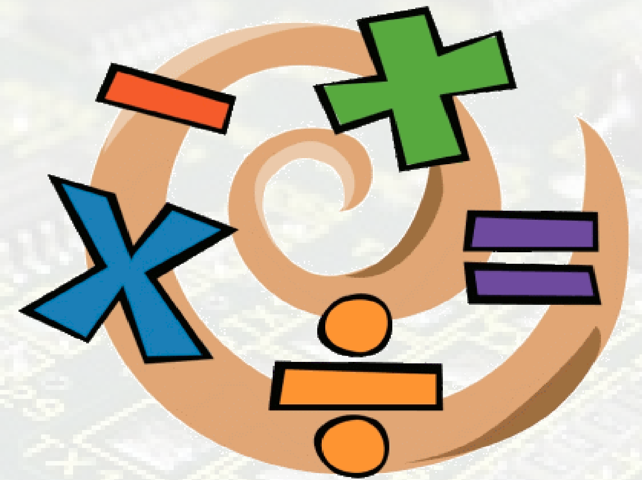
```
51  {
52     /* inicializada apenas na primeira vez que useStaticLocal é chamada */
53     static int x = 50;
54
55     printf( "\nx estática local é %d na entrada de useStaticLocal\n", x );
56     x++;
57     printf( "x estática local é %d na saída de useStaticLocal\n", x );
58  } /* fim da função useStaticLocal */
59
60  /* função useGlobal modifica variável global x durante cada chamada */
61  void useGlobal( void )
62  {
63     printf( "\nx global é %d na entrada de useGlobal\n", x );
64     x *= 10;
65     printf( "x global é %d na saída de useGlobal\n", x );
66  } /* fim da função useGlobal */
```

Exemplo: escopo

```
x local no escopo externo de main é 5
x local no escopo interno de main é 7
x local no escopo externo de main é 5

x local em useLocal é 25 após entrar em useLocal
x local em useLocal é 26 antes de sair de useLocal

x local estática é 50 na entrada de useStaticLocal
x local estática é 51 na saída de useStaticLocal

x global é 1 na entrada de useGlobal
x global é 10 na saída de useGlobal

x local em useLocal é 25 após entrar em useLocal
x local em useLocal é 26 antes de sair de useLocal

x local estática é 51 na entrada de useStaticLocal
x local estática é 52 na saída de useStaticLocal

x global é 10 na entrada de useGlobal
x global é 100 na saída de useGlobal

x local em main é 5
```

Figura 5.12 ■ Exemplo de escopo. (Parte 2 de 2.)

Finalizando



Revisão

- Vetores e arrays.
- Funções: declaração, comando return,
- função main, tipo void,
- passagem de parâmetros por valor e por referência,
- escopo de nomes e variáveis locais e globais,
- protótipo de função,
- recursão.

Sugestões Finais

- Resolvam a lista de exercícios relacionada com os temas da aula
  - Só se aprende a programar, programando
- Dúvidas sobre os exercícios podem ser enviadas por e-mail
- Leiam o material de apoio
  - Curso de C da UFMG: http://mico.ead.cpdee.ufmg.br/cursos/C/

Referências Bibliográficas

- Paul Deitel e Harvey Deitel, C: como programar, 6a edição, Ed. Prentice Hall Brasil, 2011.
- Curso de C da UFMG: http://mico.ead.cpdee.ufmg.br/cursos/C/