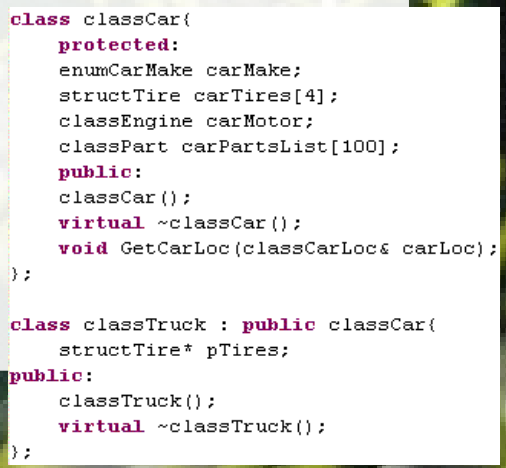
Curso de Programação C em Ambientes Linux – Aula 04

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

- Como declarar ponteiros
- Como usar ponteiros
- Como passar ponteiros para funções (por referência e não por valor)
- Exercícios

Introdução (1)

- Ponteiros são um dos recursos mais poderosos da linguagem C
- Ponteiros são variáveis cujos valores são endereços de memória
  - Ou seja, apontam para um endereço de memória
- Normalmente, as variáveis contém um valor específico relacionado com seu tipo
  - int a = 10;
- Um ponteiro, por outro lado, contém um endereço de uma variável que contém um valor

Introdução (2)

- De certa forma, um nome de variável referencia um valor diretamente, enquanto um ponteiro referencia um valor indiretamente
- A referência de um valor por meio de um ponteiro é chamada de indireção

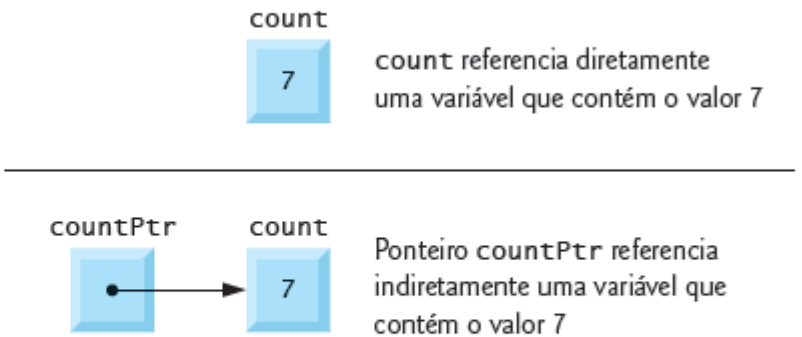


Figura 7.1 ■ Referências direta e indireta de uma variável.

Declaração e Inicialização (1)

- Declarando um ponteiro

```
int *countPtr, count;
```

- A variável countPtr é do tipo int *, ou seja, um ponteiro para um inteiro
- Note a diferença para a declaração de count
- Quando * é usado dessa maneira em uma declaração, ele indica que a variável é um ponteiro
- Os ponteiros podem ser definidos para apontar para variáveis de qualquer tipo

Declaração e Inicialização (2)

- Um ponteiro pode ser inicializado com NULL, 0 ou um endereço
- Um ponteiro de valor NULL não aponta para nada
- NULL é uma constante definida no cabeçalho <stddef.h> e outros como <stdio.h>
- Inicializar em 0 é equivalente a inicializar com NULL, mas NULL é mais conveniente

Operadores de Ponteiros (1)

- O &, ou operador de endereço, é um operador unário que retorna o endereço de seu operando
- Por exemplo, considerando as definições

```
int y = 5;
int *yPtr;
```

- A instrução:

  yPtr = &y;

- Atribui o endereço da variável ***y*** ao ponteiro ***yPtr***
- A variável **yPtr** então aponta para **y**

Exemplo

- Supondo que a variável inteira y esteja armazenada no endereço de memória 600000, e a variável de ponteiro yPtr esteja armazenada no endereço 500000
- O operador de endereço só pode ser aplicado a variável e não a constantes ou expressões

Figura 7.3 ■ Representação de y e yPtr na memória.

Operadores de Ponteiros (2)

- O operador unário *, chamado de operador de indireção, retorna o valor do objeto apontado por seu operando (um ponteiro)
- Por exemplo, a instrução:

```
printf( "%d", *yPtr );
```

- Imprime o valor da variável y, que é 5

O que faz o código abaixo?

```c
/* Fig. 7.4: fig07_04.c
   Usando os operadores & e *  */
#include <stdio.h>

int main( void )
{
   int a; /* a é um inteiro */
   int *aPtr; /* aPtr é um ponteiro para um inteiro */

   a = 7;
   aPtr = &a; /* aPtr definido para o endereço de a */

   printf( "O endereço de a é %p"
       "\nO valor de aPtr é %p", &a, aPtr );

   printf( "\n\nO valor de a é %d"
       "\nO valor de *aPtr é %d", a, *aPtr );

   printf( "\n\nMostrando que * e & são complementos um "
      "do outro\n&*aPtr = %p"
      "\n*&aPtr = %p\n", &*aPtr, *&aPtr );
   return 0; /* indica conclusão bem-sucedida */
} /* fim do main */
```

```
O endereço de a é 0012FF7C
O valor de aPtr é 0012FF7C

O valor de a é 7
O valor de *aPtr é 7

Mostrando que * e & são complementos um do outro.
&*aPtr = 0012FF7C
*&aPtr = 0012FF7C
```

Figura 7.4 ■ Usando os operadores de ponteiros & e *.

Passando argumentos para funções por referência (1)

- Existem duas maneiras de passar argumentos a uma função, denominadas chamada por valor e chamada por referência
- Todos os argumentos em C são passados por valor
- Muitas funções exigem a capacidade de modificar uma ou mais variáveis na função chamadora ou passar um ponteiro para um objeto com grande quantidade de dados, para evitar o overhead de passar o objeto por valor (o que implicaria na sobrecarga de ter de fazer uma cópia do objeto inteiro)

Passando argumentos para funções por referência (2)

- Para essas finalidades, C oferece recursos para uma simulação de chamada por referência
- Em C, você usa ponteiros e operadores de indireção para simular uma chamada por referência
- Ao chamar uma função com argumentos que devem ser modificados, os endereços dos argumentos são passados
- Isso normalmente é feito ao se aplicar o operador (&) à variável (na função chamadora), cujo valor será modificado

Exemplo: calculo do cubo de um número por valor (1)

```
1  /* Fig. 7.6: fig07_06.c
2     Cubo de uma variável usando chamada por valor */
3  #include <stdio.h>
4
5  int cubeByValue( int n ); /* protótipo */
6
7  int main( void )
8  {
9     int number = 5; /* inicializa número */
10
11    printf( "O valor original do número é %d", number );
12
13    /* passa número por valor a cubeByValue */
14    number = cubeByValue( number );
15
```

Figura 7.6 ■ Cubo de uma variável usando chamada por valor. (Parte 1 de 2.)

Exemplo: calculo do cubo de um número por valor (2)

```
16     printf( "\nO novo valor do número é %d\n", number );
17     return 0; /* indica conclusão bem-sucedida */
18  } /* fim do main */
19
20  /* calcula e retorna cubo do argumento inteiro */
21  int cubeByValue( int n )
22  {
23     return n * n * n; /* calcula cubo da variável local n e retorna resultado */
24  } /* fim da função cubeByValue */
```

```
O valor original do número é 5
O novo valor do número é 125
```

Figura 7.6 ■ Cubo de uma variável usando chamada por valor. (Parte 2 de 2.)

Exemplo: calculo do cubo de um número por referência (1)

```c
/* Fig. 7.7: fig07_07.c
   Calcula o cubo de uma variável usando chamada por referência com argumento ponteiro */

#include <stdio.h>

void cubeByReference( int *nPtr ); /* protótipo */

int main( void )
{
   int number = 5; /* inicializa número */

   printf( "O valor original do número é %d", number );

   /* passa endereço do número a cubeByReference */
   cubeByReference( &number );

   printf( "\nO novo valor do número é %d\n", number );
   return 0; /* indica conclusão bem-sucedida */
} /* fim de main */

/* calcula cubo de *nPtr; modifica variável number em main */
void cubeByReference( int *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr; /* cubo de *nPtr */
} /* fim da função cubeByReference */
```

```
O valor original do número é 5
O novo valor do número é 125
```

Figura 7.7 ■ Cubo de uma variável usando chamada por referência com um argumento ponteiro.

Passando argumentos para funções por referência (2)

- Uma função que recebe um endereço como argumento precisa definir um parâmetro de ponteiro para receber esse endereço
- No exemplo anterior, na linha 22:

```
void cubeByReference( int *nPtr )
```

- Especifica que a função recebe o endereço de uma variável inteira como argumento, armazena o endereço localmente em nPtr e não retorna valor

Passo a passo: chamada por valor (1)

Etapa1: Antes de chamar cubeByValue:

```
int main( void )
{
   int number = 5;

   number = cubeByValue( number );
}
```

number: 5

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

n: indefinido

Etapa 2: Depois de cubeByValue ter recebido a chamada:

```
int main( void )
{
   int number = 5;

   number = cubeByValue( number );
}
```

number: 5

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

n: 5

Etapa 3: Depois de cubeByValue elevar ao cubo o parâmetro n e antes de cubeByValue retornar para main:

```
int main( void )
{
   int number = 5;

   number = cubeByValue( number );
}
```

number: 5

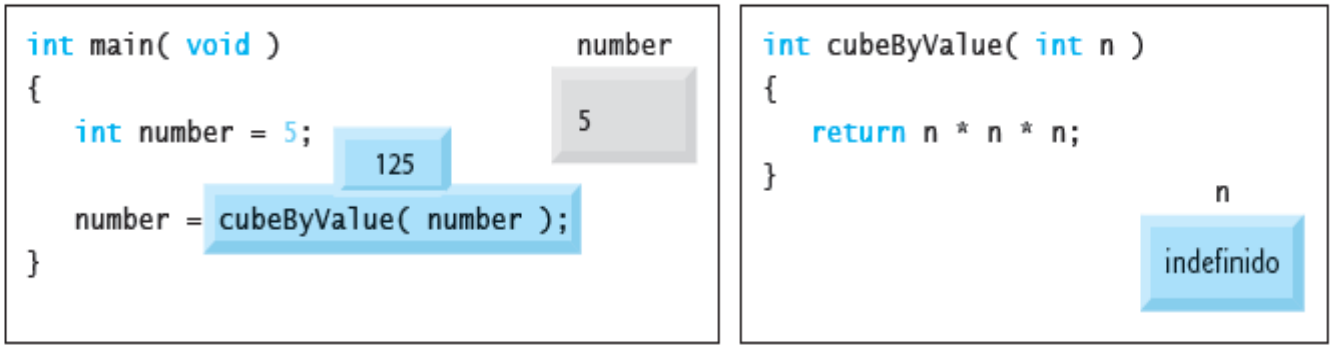
```
int cubeByValue( int n )
{
   return n * n * n;
}
```

125

n: 5

Passo a passo: chamada por valor (2)

Etapa 4: Depois de `cubeByValue` retornar para `main` e antes de atribuir o resultado a `number`:



Etapa 5: Depois de `main` completar a atribuição a `number`:

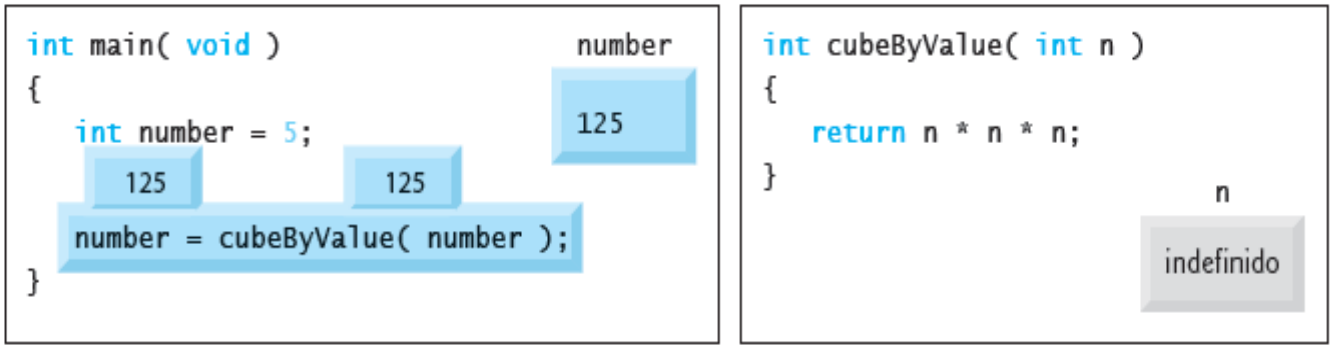
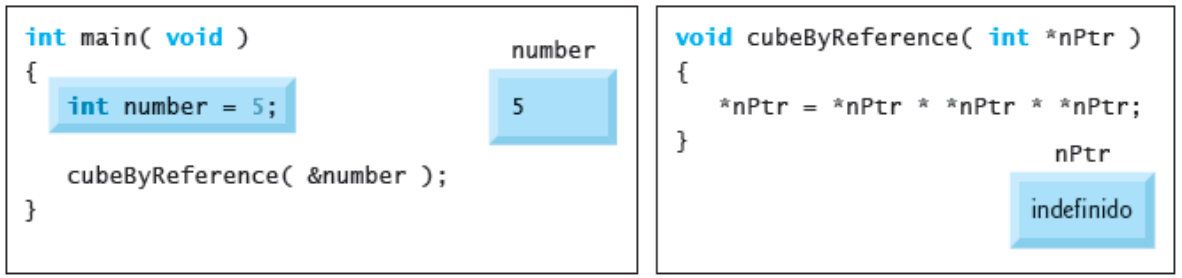


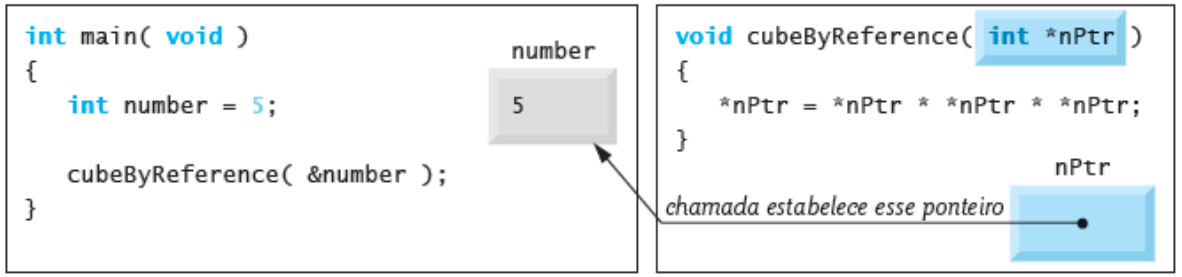
Figura 7.8 ■ Análise de uma típica chamada por valor.

Passo a passo: chamada por referência

Etapa 1: Antes de main chamar cubeByReference:



Etapa 2: Depois de cubeByReference receber a chamada e antes de *nPtr ser elevado ao cubo.



Etapa 3: Depois de *nPtr ser elevado ao cubo e antes de o controle do programa retornar a main:

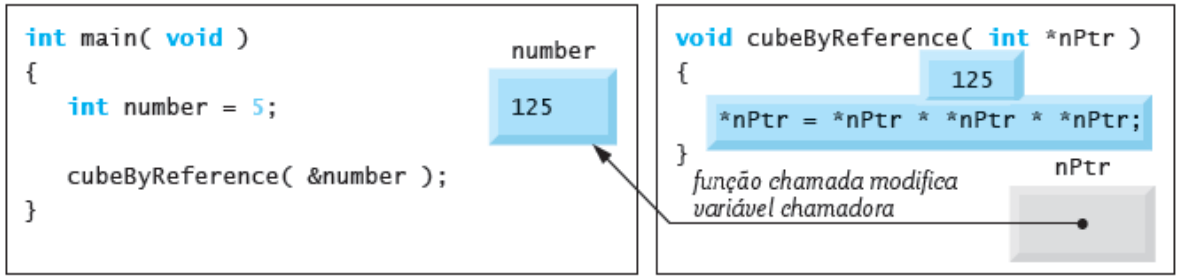


Figura 7.9 ■ Análise de uma típica chamada por referência com um argumento de ponteiro.

Passando ponteiros e arrays para funções

- O compilador não diferencia uma função que recebe um ponteiro de uma função que recebe um array unidimensional
- Exemplo:
  int func(int b[])

  *int func(int *b)*
- Significa a mesma coisa. O compilador converte o parâmetro b[] para a notação de ponteiro automaticamente

Expressões com ponteiros e aritmética de ponteiros (1)

- Ponteiros são operandos válidos em expressões aritméticas, expressões de atribuição e expressões de comparação
- Porém, nem todos os operadores normalmente usados nessas expressões são válidos em conjunto com variáveis de ponteiro
- Um ponteiro pode ser incrementado (++) ou decrementado (--), um inteiro pode ser somado a um ponteiro (+ ou +=), um inteiro pode ser subtraído de um ponteiro (- ou -=) e um ponteiro pode ser subtraído de outro

Expressões com ponteiros e aritmética de ponteiros (2)

- Suponha que o vetor int v[5] tenha sido definido e que seu primeiro elemento esteja no local 3000
- Suponha que o ponteiro vPtr tenha sido inicializado para apontar v[0] – ou seja, o valor de vPtr é 3000

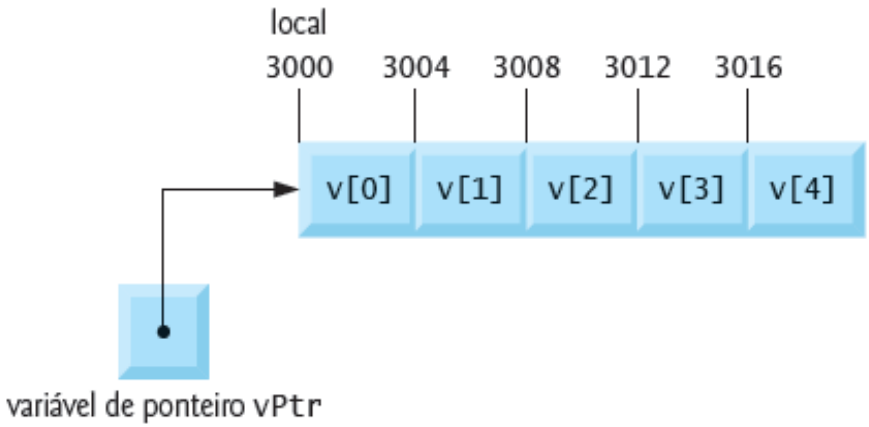


Figura 7.18 ■ Array v e uma variável de ponteiro vPtr que aponta para v.

Expressões com ponteiros e aritmética de ponteiros (3)

- Suponha que você faça a operação vPtr += 2
- Na aritmética convencional, 3000 + 2 gera 3002. Isso normalmente não acontece com a aritmética de ponteiro
- Quando um inteiro é somado ou subtraído de um ponteiro, o ponteiro não é simplesmente incrementado ou decrementado por esse inteiro, mas pelo inteiro multiplicado pelo tamanho do objeto ao qual o ponteiro se refere
- O número de bytes depende do tipo. Por exemplo, a instrução: vPtr += 2;
- Produzirá 3008 (3000 + 2 * 4), supondo que um inteiro seja de tamanho de 4 bytes

Expressões com ponteiros e aritmética de ponteiros (4)

- Voltando ao exemplo, no array v, vPtr agora apontaria para v[2]

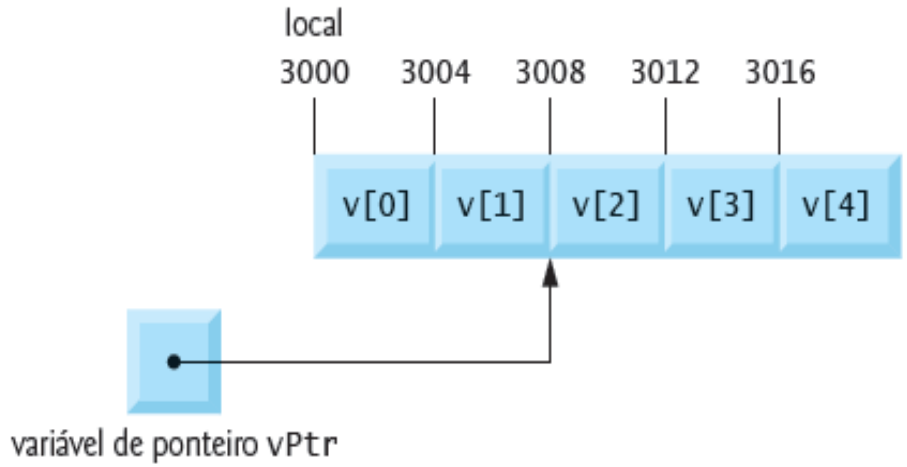


Figura 7.19 ■ O ponteiro vPtr após a execução da aritmética de ponteiro.

- Qual seria o resultado de vPtr-- ?

Exemplo 1

```
#include <stdio.h>
int main ()
{
   int num,valor;
   int *p;
   num=55;
   p=&num;   //Pega o endereco de num
   valor=*p; //Valor e igualado a num de uma maneira indireta
   printf ("\n\n%d\n",valor);
   printf ("Endereco para onde o ponteiro aponta: %p\n",p);
   printf ("Valor da variavel apontada: %d\n",*p);
   return(0);
}
```

Exemplo 2

```
#include <stdio.h>
int main ()
{
   int num,*p;
   num=55;
   p=&num;      /* Pega o endereco de num */
   printf ("\nValor inicial: %d\n",num);
   *p=100; /* Muda o valor de num de uma maneira indireta */
   printf ("\nValor final: %d\n",num);
   return(0);
}
```

Ponteiro para Ponteiro (1)

- Um ponteiro para ponteiro é como se você anotasse o endereço de um papel que tem o endereço da casa do seu amigo
- Declaração
  - Tipo **nome_da_variável;
- No C podemos declarar ponteiros para ponteiros para ponteiros, ou então, ponteiros para ponteiros para ponteiros para ponteiros (UFA!) e assim por diante
  - Basta aumentar o número de * na declaração

Ponteiro para Ponteiro (2)

- Para acessar o valor desejado apontado por um ponteiro para ponteiro, o operador asterisco deve ser aplicado duas vezes

```
#include <stdio.h>
int main()
{
    float fpi = 3.1415, *pf, **ppf;
    pf = &fpi;                  /* pf armazena o endereco de fpi */
    ppf = &pf;                  /* ppf armazena o endereco de pf */
    printf("%f", **ppf);       /* Imprime o valor de fpi */
    printf("%f", *pf);          /* Tambem imprime o valor de fpi */
    return(0);
}
```

Ponteiro para função (1)

- Um ponteiro para função contém o endereço da função na memória
- Vimos que o nome de um array é, na realidade, o endereço na memória do primeiro elemento do array
- De modo semelhante, um nome de função é o endereço inicial na memória do código que realiza a tarefa da função
- Os ponteiros para funções podem ser passados para funções, retornados de funções, armazenados em arrays e atribuídos a outros ponteiros para funções

Ponteiro para função (2)

- Declaração
  - int (*func_ptr)(void)
- func_ptr é um ponteiro para uma função que retorna um inteiro e não recebe parâmetros
- Como usar

```
int funcao(void) { … }
int main(void) {
    int (*func_ptr)(void);
    func_ptr = funcao;
   (*func_ptr)(); //invoca a função
}
```

Ponteiro para função (3)

- Os ponteiros para função são comumente usados nos sistemas controlados por menu de texto
- Um usuário precisa selecionar uma opção a partir de um menu (possivelmente, de 1 a 5), digitando o número do item de menu.
- Cada opção é atendida por uma função diferente
- Os ponteiros para cada função são armazenados em um array de ponteiros para funções
- A escolha do usuário é utilizada como um subscrito no array, e o ponteiro no array é usado para chamar a função

Exemplo (1)

```c
/* Fig. 7.28: fig07_28.c
   Demonstrando um array de ponteiros para funções */
#include <stdio.h>

/* protótipos */
void function1( int a );
void function2( int b );
void function3( int c );

int main( void )
{
   /* inicializa array de 3 ponteiros para funções que usam um
   argumento int e retornam void */
   void (*f[ 3 ])( int ) = { function1, function2, function3 };

   int choice; /* variável para manter escolha do usuário */

   printf( "Digite um número entre 0 e 2, 3 para sair: " );
   scanf( "%d", &choice );

   /* processa escolha do usuário */
   while ( choice >= 0 && choice < 3 ) {

      /* chama a função para o local selecionado do array f e passa
         choice como argumento */
      (*f[ choice ])( choice );

      printf( "Digite um número entre 0 e 2, 3 para terminar: ");
      scanf( "%d", &choice );
   } /* fim do while */

   printf( "Execução do programa concluída.\n" );
   return 0; /* indica conclusão bem-sucedida */
} /* fim do main */
```

Exemplo (2)

```
36  void function1( int a )
37  {
38     printf( "Você digitou %d, de modo que function1 foi chamada\n\n", a );
39  } /* fim de function1 */
40
41  void function2( int b )
42  {
43     printf( "Você digitou %d, de modo que function2 foi chamada\n\n", b );
44  } /* fim de function2 */
45
46  void function3( int c )
47  {
48     printf( "Você digitou %d, de modo que function3 foi chamada\n\n", c );
49  } /* fim de function3 */
```

Figura 7.28 ■ Demonstração de um array de ponteiros para funções. (Parte 1 de 2.)

Exemplo (3)

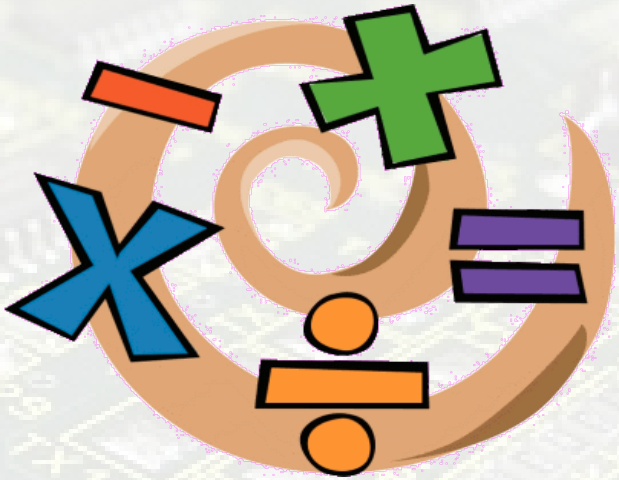
```
Digite um número entre 0 e 2, 3 para sair: 0
Você digitou 0, de modo que function1 foi chamada

Digite um número entre 0 e 2, 3 para sair: 1
Você digitou 1, de modo que function2 foi chamada

Digite um número entre 0 e 2, 3 para sair: 2
Você digitou 2, de modo que function3 foi chamada

Digite um número entre 0 e 2, 3 para sair: 3
Execução do programa concluída.
```

Figura 7.28 ■ Demonstração de um array de ponteiros para funções. (Parte 2 de 2.)

Cuidados a serem tomados com ponteiros

- Saiba sempre para onde o ponteiro está apontando
  - Nunca use um ponteiro que não foi inicializado
- Não irá acontecer nenhum erro de compilação se o ponteiro for usado sem ser inicializado
- Possível erro de execução
  - Segmentation fault

Finalizando



Revisão

- Como declarar ponteiros em C – int * p;
- Como inicializar um ponteiro
  - int a = 10;
  - int * p = &a;
- Operadores * e &
- Chamada de funções por referência
  - int funcao(int * ponteiro)
  - int a = funcao(&variavel)
- Expressão aritmética de ponteiros

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/