Introdução à Programação

Programação C

Apontadores

Prof. Roberto M. de Faria/UASC/UFCG

Conteúdo

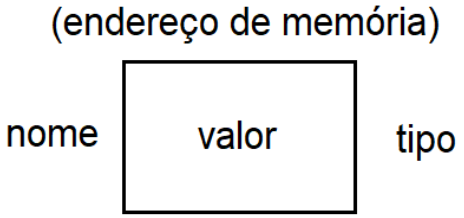
- Introdução
- Declaração e inicialização de variáveis para apontamento
- Operadores de apontadores
- Chamada de funções por referência
- Uso do qualificador ***const*** com apontadores
- Ordenação Borbulhante (Bubble Sort) usando chamada por referência
- Expressões e aritmética de apontadores
- Relação entre apontadores e arrays
- Arrays de apontadores
- Estudo de caso: simulação de embaralhamento e distribuição de cartas
- Apontadores para funções

Introdução

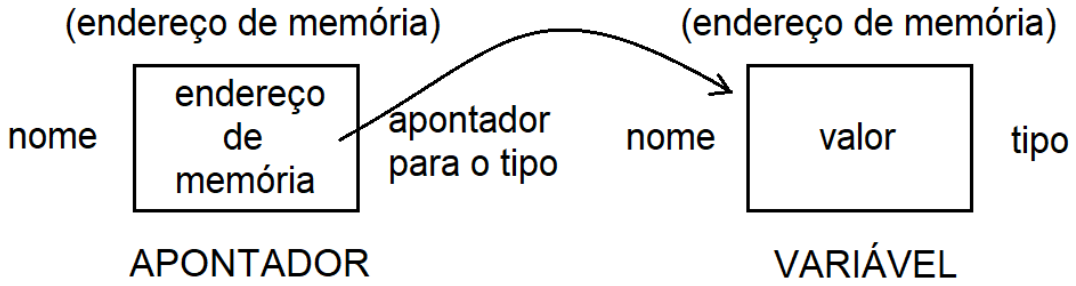
- Apontadores
  - Poderosos, apesar de domínio mais difícil
  - Possibilitam a simulação de chamadas por referência, dando acesso a dados que foram definidos (declarados) fora da função
  - Relação íntima com ***arrays*** e ***cadeias de caracteres*** (***strings***) – o nome de um array é um apontador constante que contém o endereço do primeiro byte dos dados do array na memória

Declaração e Inicialização de Variáveis para Apontamento

- Atributos das variáveis



- Variáveis normais contêm valores específicos de um tipo (acessados por referência direta)
- Variáveis para apontamento – ***apontadores*** ou ***ponteiros*** – contêm endereços de memória de outras variáveis
  - Apontadores contêm endereços de variáveis que contêm valores específicos (referência indireta)



  - Indireção – referência de um valor via apontador

Declaração e Inicialização de Variáveis de Apontamento

- Declaração de Apontadores
  - Um "*****" é usado para declarar variáveis de apontamento

    `tipo *nome_do_apontador;`
  - Declaração de um apontador para um **`int`** – apontador do tipo **`int*`**

    `int *apont;`
- Apontadores múltiplos requerem o uso de um "*" antes de cada declaração de variável

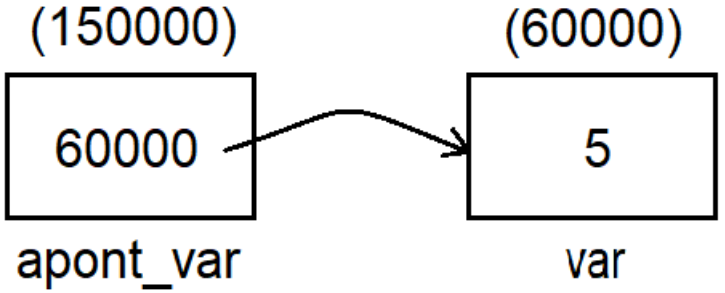
  `int *apont1, *apont2;`
- Possibilidade de declaração de apontadores qualquer tipo de variável
- Inicialização de apontadores para **`0`**, **`NULL`** ou um endereço
- Apontadores para nada – **`0`** ou **`NULL`** (preferencialmente)

Operadores de Apontamento

- & (Operador de endereçamento)
  - Retorna o endereço de um operando

```
int var = 5;
int *apont_var;
apont_var = &var;
```

  - **apont_var** contém o endereço de **var** – **apont_var** "aponta para" **var**



Operadores de Apontamento

- **`*`** (Operador de indireção/desreferenciamento)
  - Retorna uma cópia do conteúdo da locação para a qual o operando aponta – conteúdo de uma variável
  - **`*apont_var`** retorna o conteúdo de **`var`** (visto que **`apont_var`** "aponta para" **`var`**)
  - **`*`** é também usado para atribuição – para armazenar na variável apontada pelo apontador

```
*apont_var = 7;  // altera o valor de var
                 // para 7
```

  - O operando de **`*`** deve ser uma variável
  - **`*`** e **`&`** são inversos – um cancela o outro

Operadores de Apontamento

```
#include <stdio.h>
int main() {
   int val;    // val é uma variável tipo inteiro
   int *apont; // apont é um apontador para um inteiro

   apont = &val; // apont aponta para o endereço de val
   *apont = 7;
   printf("O endereço de val eh %p(%d)\n\n", &val, &val);
   printf("O valor de apont eh %p(%d)\n\n", apont, apont);
   printf("O valor de val eh %d\n\n", val);
   printf("O valor de *apont eh %d\n\n", *apont);
   printf("Prova de que * e & sao complementares:\n\n");
   printf("*&apont = %p\n\n", *&apont);
   printf("&*apont = %p\n\n", &*apont);
   return 0;
}
```

Chamada de Funções por Referência

- É passado como argumento uma cópia do endereço de uma locação de memória – não uma cópia do valor do argumento
- Passagem do endereço do argumento via operador **&** ou via outro apontador
- Possibilidade de alteração de uma variável local de outra função
- Impossibilidade de passagem de ***arrays*** com **&**, pois o nome de um array já é um apontador – passa-se o endereço do array (nome) que é recebido num apontador para o array (parâmetro da função)

Chamada de Funções por Referência

```
/*  Troca os valores de duas variáveis com o uso de uma função  */
#include <stdio.h>

void troca_valores(int*, int*);

int main() {
    int var1 = 10, var2 = 20;

    setlocale(LC_ALL, "");
    printf("Troca os valores de duas variaveis:\n\n");
    printf("main\(\) antes da troca: var1 = %d   var2 = %d\n\n",
            var1, var2);
    troca_valores(&var1, &var2);
    printf("main\(\) depois da troca: var1 = %d   var2 = %d\n\n",
            var1, var2);
    return 0;
}
```

Chamada de Funções por Referência

```
void troca_valores(int *primeiro, int *segundo) {
    int aux;

    printf("troca_valores\(\) antes da troca: *primeiro = %d"
           "   *segundo = %d\n\n", *primeiro, *segundo);
    aux = *primeiro;
    *primeiro = *segundo;
    *segundo = aux;
    printf("troca_valores\(\) depois da troca: *primeiro = %d"
           "   *segundo = %d\n\n", *primeiro, *segundo);
    return;
}
```

Exercícios

1) Faça um programa que receba o valor de um raio e calcule o comprimento de uma circunferência, a área de um círculo e o volume de uma esfera com este raio, usando uma função única **`executa_calculos()`** para executar os cálculos das três grandezas

2) Faça um programa, que usando as teclas de setas (para cima, para baixo, para a esquerda e para a direita), movimente um asterisco na tela de saída do programa, usando as funções:

- **`va_para_coordenadas_xy()`**
- **`movimenta_asterisco()`**
- **`obtem_tecla()`**
- **`mostra_asterisco_tela()`**

Use as variáveis **`coord_x`** e **`coord_y`** definidas na função main() para definir a posição momentânea do asterisco – não use variáveis globais

Use outras funções se necessário

Uso do Qualificador **`const`** com Apontadores

- Com o uso do qualificador **`const`** uma variável não pode ser alterada – uso de **`const`** se a função não precisar alterar a variável
- A tentativa de alteração de uma variável **`const`** produz um erro
- Apontadores com uso do **`const`**
  - Apontador constante (não pode ser modificado) – inicialização obrigatória no ato da declaração – para uma locação de memória que pode ser modificada

    ```
    int *const apont = &var;
    ```

  - Apontador normal (pode ser alterado) para uma locação de memória constante que não pode ser alterada

    ```
    const int *apont = &var;
    ```

  - Apontador constante para uma locação de memória constante

    ```
    const int *const apont = &var;
    ```

    Obs.: pode alterar a locação de memória usando **`var`**, mas não pode alterar usando **`*apont`**

Uso do Qualificador **`const`** com Apontadores

```
// Tentativa de modificação de um apontador constante para dados
// não constantes
#include <stdio.h>

int main() {
   int var1, var2;
   int *const apt = &var1; // apt é um apontador constante para um
                           // inteiro passível de modificação
                           // através de *apt, embora apt aponte
                           // sempre para a mesma locação de
                           // memória.

   *apt = 7;
   apt = &var2; // Será gerado um erro, pois apt é constante
   return 0;
}
```

Ordenação Borbulhante Usando Chamada por Referência

- Implementação da ordenação borbulhante (Bubble Sort) usando apontadores
- Permuta de posição de dois elementos
  - Passagem por referência dos elementos do array para a função **`permuta()`** – passagem dos endereços dos elemento com uso do operador **`&`**
  - Por definição (padrão), os elementos de um array são ***passados por valor*** para uma função
  - Uso de apontadores e do operador **`*`** para a permuta de elementos do array pela função **`permuta()`**

Ordenação Borbulhante Usando Chamada por Referência

```
// Programa que organiza valores de um array
// em ordem crescente, imprimindo o resultado
#include <stdio.h>
#define TAMANHO 10

void borbulha(int*, const int);

int main() {
   int a[TAMANHO] = {2, 6, 4, 8, 10,
                     12, 89, 68, 45, 37};
   int i;

   printf("Dados na ordem original \n");
   for (i = 0; i <= TAMANHO - 1; i++)
      printf("%4d", a[i]);
   borbulha(a, TAMANHO); // ordena o array
   printf("\nDados em ordem crescente\n");
   for (i = 0; i <= TAMANHO - 1; i++)
      printf("%4d", a[i]);
   printf("\n");
   return 0;
}
```

```
// Função que varre um array, organizando
// seus elementos em ordem crescente
void borbulha(int *array, const int tam) {
   void permuta(int*, int*);
   int passada, pos;

   for (passada = 1; passada <= tam - 1;
        passada++)
      for (pos = 0; pos <= tam - 2; pos++)
         if (array[pos] > array[pos + 1])
            permuta(&array[pos],
                    &array[pos + 1]);
   return;
}
void permuta(int *apontelem1,
             int *apontelem2) {
   int auxi = *apontelem1;

   *apontelem1 = *apontelem2;
   *apontelem2 = auxi;
   return;
}
```

Cálculo do tamanho de arrays, variáveis e tipos

- Operador **`sizeof`** – obtém o número de bytes que o operando ocupa na memória
  - **`sizeof(variável)`** – obtém o número de bytes ocupado pela variável

    `sizeof(valor)` – se valor for **`float`**, retorna 4
  - **`sizeof(tipo)`** – obtém o número de bytes ocupado por um valor deste tipo

    `sizeof(long long int)` – retorna 8
  - **`sizeof(constante)`** – obtém o número de bytes ocupado pela constante

    `sizeof(3.14159)` – retorna 4

    `sizeof('@')` – retorna 1
- Para arrays, **`sizeof(array)`**, retorna o produto do tamanho de um elemento pelo número de elementos
  - Como **`sizeof(int)`** é igual a 4 bytes, então

```
int meuArray[10];
printf("%d", sizeof(meuArray)); // imprimirá 40
```

Expressões e Aritmética de Apontadores

Array de inteiros com 5 elementos (**`int`** ocupa 4 bytes de memória

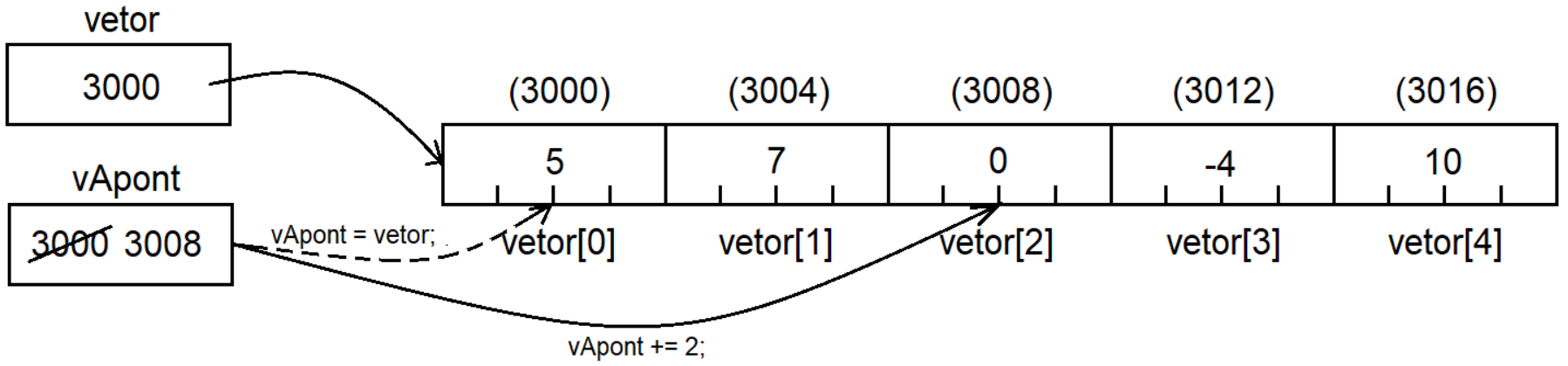
```
int vetor[5] = {5, 7, 0, -4, 10}, *vApont;
```

```
vApont = vetor;
```

- **`vApont`** aponta para o primeiro elemento – ***vetor[0]*** (se **`vApont`** receber 3000, vetor está armazenado na locação 3000

```
VApont += 2; // vApont = vApont + 2;
```

- redireciona **`vApont`** para 3008 – **`vApont`** passará a apontar para ***vetor[2]*** – incremento de 4*2



Expressões e Aritmética de Apontadores

- Subtração de Apontadores
  - Retorno do número de elementos de um apontador para o outro

    `vApont2 = &v[2];`

    `vApont1 = &v[0];`

    `vApont2 - vApont1` – produzirá 2 como resultado
- Comparação de Apontadores (<, == e >)
  - Só tem sentido se ambos os apontadores apontam para o ***mesmo array***
- Exemplos
  - Verificação de qual dos apontadores aponta para o elemento de maior índice do array
  - Verificação para determinar se se trata de um apontador ***NULL***

Expressões e Aritmética de Apontadores

- Apontadores do mesmo tipo podem ser atribuídos um ao outro
- Tipos diferentes – Necessidade de um operador de conversão
  - Transformação do tipo do apontador à direita da atribuição para o tipo do operador à esquerda da atribuição
- Exceção – Apontador para ***void*** (tipo **`void*`**)
- Apontador genérico – Representação de qualquer tipo de apontador
- Todos os tipos de apontadores podem ser atribuídos a um apontador para ***void***
  - Nenhuma conversão é necessária
  - Apontadores para ***void*** não podem ser desreferenciados

Expressões e Aritmética de Apontadores

- EXERCÍCIO

  Refazer o programa do Bubblesort tal que:

  - os valores a serem ordenados sejam obtidos de fora;
  - a função permuta tem um só parâmetro e o outro endereço seja obtido somando um ao apontador que é passado

Alocação Dinâmica de Memória em C

- Os apontadores fornecem o suporte necessário para implementar alocação dinâmica em C
- Alocação dinâmica é o meio através do qual um programa pode obter memória em tempo de execução, ou seja, criar variáveis sem prévia declaração, durante a execução do programa
- A importância da alocação dinâmica é não restringir o programa a utilizar apenas a memória já alocada nas declarações, fazendo assim, alocação de memória sob demanda
- O sistema de alocação dinâmica de C consiste nas seguintes funções
  - **`(tipo*)malloc(numero_de_bytes)`** – aloca a memória sem inicializar – retorna o endereço da locação
  - **`(tipo*)calloc(numero_de_elementos, tamanho_do_elemento)`** – aloca a memória inicializando com zeros – retorna o endereço da locação
  - **`free(apontador)`** – para liberar memória

Alocação Dinâmica de Memória em C

```
//  Programa para ler um conjunto de inteiros e mostrá-los
//  na ordem inversa a de leitura
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>

void main() {
   int *array, *ap, *apfim, num_elem;

   setlocale(LC_ALL, "");
   system("cls");
   printf("APRESENTAÇÃO DE INTEIROS NA ORDEM INVERSA A DE LEITURA\n\n");
   printf("Com quantos números deseja trabalhar?\n", num_elem);
   scanf("%d", &num_elem);
   arranjo = (int*) malloc(sizeof(int) * num_elem);
   printf("Informe os %d valores inteiros:\n", num_elem);
   for (ap = array, apfim = ap + num_elem - 1; ap <= apfim; ap++)
      scanf("%d", ap);
   printf("\nElementos na ordem inversa a de leitura:\n");
   for (ap = array + num_elem - 1, apfim = array;
        ap >= apfim; ap--)
      printf("%d\n", *ap);
   return 0;
}
```

Relação entre Apontadores e Arrays

- Arrays e Apontadores
  - Relação íntima – uso quase indiferente de ambos
  - O nome de um array é um apontador constante
  - É possível utilizar subscritos com apontadores
- Declaração de um array **`vetor[5]`** e um apontador **`vApont`**

```
int vetor[5], *vApont;
vApont = vetor; // ou
vApont = &vetor[0];
```

- Atribuição explícita de **`vApont`** para endereço do primeiro elemento de **`vetor`**

Relação entre Apontadores e Arrays

- Acesso ao elemento **`vetor[3]`** do array de várias formas
  - Pelo subscrito

    `vetor[3]`
  - Pelo uso de um deslocamento (**`+`** ou **`-`**) para o apontador, usando aritmética de apontadores

    `*(vApont + 3) // 3 é o deslocamento`
  - Pelo uso de um subscrito para um apontador

    `vApont[3]`
  - Notação apontador/subscrito

    `vApont[3]` é mesmo que **`vetor[3]`**

Arrays de Apontadores

- Arrays podem conter apontadores

```
int *array_apont[10];
```

- Array de cadeias de caracteres

```
char *naipe[4]={"Copas", "Paus", "Ouro",
                "Espadas"};
```

- Cadeias de caracteres são apontadores para o primeiro caractere
- **`char *`** – cada elemento de **`naipe`** é um apontador para um **`char`**
- As cadeias de caracteres não são realmente armazenadas no array **`naipe`**, são armazenados apenas os apontadores para as cadeias de caracteres

Arrays de Apontadores

- O array naipe tem tamanho fixo
- As cadeias de caracteres podem ter qualquer tamanho

```
char *naipe[4]= {"Copas", "Paus", "Ouros", "Espadas"};
```