Introdução à Ciência da Computação II

Revisão de Linguagem C
Pt. III: Ponteiros

Prof. Ricardo J. G. B. Campello

Sumário

- Introdução
- Definição de Ponteiros
- Declaração de Ponteiros em C
- Manipulação de Ponteiros em C
  - Operações
  - Ponteiros e Arranjos
  - Alocação Dinâmica de Memória
  - Passagem/Retorno de Ponteiros em Funções

Introdução

- Por quê ponteiros são importantes?
  - Permitem passagem de parâmetros para funções por referência
  - São usados para alocar e liberar memória dinamicamente (em tempo de execução)
  - Possibilitam a implementação eficiente de certas **estruturas de dados**

3

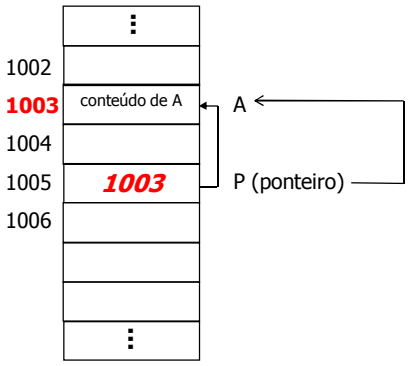
Definição

- Um **ponteiro** (ou **apontador**) é uma variável que armazena um endereço de memória
  - Normalmente, esse endereço é a posição de outra variável na memória
  - Dizemos portanto que um ponteiro "aponta" para uma variável
  - O tipo do ponteiro é normalmente associado ao tipo da variável apontada

4

Definição

Exemplo:



5

Declaração em C

- *tipo *var1, *var2, ...*;
  - O símbolo * indica que as variáveis *var1, var2, etc* são ponteiros para variáveis do tipo *tipo*.
- Alternativamente pode-se declarar como:
  - *tipo* var1, var2, ...*;
- Exemplos:
  - **float** *a, *t23, *lista;
  - **int*** b, s2;

6

Manipulação em C

- Operadores:
  - **Operador de endereço**: **&**
    - Ponteiros só podem receber endereços de memória
    - Para isso, pode-se utilizar o operador &
  - **Operador de conteúdo**: ***
    - Recupera o conteúdo da variável apontada por um ponteiro
  - **Ponteiro nulo**: **NULL**
    - Um ponteiro que recebe o valor NULL aponta para nada

7

Manipulação em C

- Atribuição (P1 e P2 ponteiros – A, B e C variáveis):
  - **Endereço de Variável para Ponteiro**
    - Exemplos: P1 = &A; P2 = &B;
  - **Conteúdo de Endereço Apontado para Variável**
    - Exemplo: C = *P2;
  - **Ponteiro para Ponteiro (ou para nulo)**
    - Exemplos: P1 = P2; P2 = **NULL**;
  - **Conteúdo para Endereço Apontado**
    - Exemplos: *P1 = A; *P1 = 34;

8

Manipulação em C

Exemplo:

```
#include <stdio.h>

void main(void){
   float A, *P;
   A = 3.145;
   printf("%f\n", A);
   P = &A;
   *P = 2.892;
   printf("%f\n",*P);
   printf("%f\n",A);
   getchar();
}
```

9

Manipulação em C

- Comparação:
  - **Como para qualquer variável**
  - Exemplos:
    - **if** (P1 < P2) /* Verifica se o endereço em P1 é menor que aquele em P2 */
    - **if** (P1 == P2) /* Verifica se P1 e P2 apontam para o mesmo endereço */
    - **if** (P1 == **NULL**) /* Verifica se P1 aponta para NULL */

10

Manipulação em C

- Aritmética:
  - **Toda a aritmética de ponteiros é relativa ao seu Tipo Base**
  - Exemplo:
    - P1 é ponteiro para **int**
    - P1 armazena o endereço de memória 4568
    - **int** ocupa 4 bytes (32 bits) no sistema em questão
    - P1++ faz com que P1 passe a armazenar o endereço 4572
      - Mesmo que P1 = P1 + 1 ou P1 += 1
    - P1-- faz com que P1 passe a armazenar o endereço 4564

11

Ponteiros para Estruturas

- Podemos declarar ponteiros para estruturas
- Por exemplo:

```
struct estrutura {
  float  campo1;
  char   campo2; }
struct estrutura  est, *pt;
```

- Na verdade, é possível declarar ponteiros para tipos definidos pelo usuário em geral

12

Ponteiros para Estruturas

- No exemplo anterior, poderíamos definir o tipo:

```
typedef struct {
                float campo1;
                char  campo2; } minha_estrutura;
minha_estrutura  est, *pt;
```

- Em qualquer caso, o acesso aos campos de uma estrutura através de um ponteiro demanda o **operador seta** (–>)

13

Ponteiros para Estruturas

- No exemplo anterior, poderíamos utilizar o operador seta para acessar os campos da var. composta `est` através de um ponteiro `pt`

```
pt = &est;
printf("O valor do campo1 é %f", pt->campo1);
printf("O valor do campo2 é %c", pt->campo2);
```

- Notas:
  - Operador `->` acessa conteúdo, não endereço !
  - `pt->campo1` é equivalente a `est.campo1`
    - equivale também a `*(&pt->campo1)` e `(*pt).campo1` Porquê?

Ponteiros e Arranjos

- Há uma relação estreita entre ponteiros e arranjos (vetores e matrizes) em linguagem C
- Por exemplo, considere as seguintes declarações:
  - **char** str[10], *p1;
    - vetor de 10 chars (string) e ponteiro para chars
  - As seguintes atribuições são equivalentes:
    - p1 = str **ou** p1 = &str[0]
    - o nome do vetor referencia o endereço do seu 1º elemento
    - o mesmo vale para matrizes, armazenadas linearmente em memória
      - como se fosse um vetor, linha por linha

15

Ponteiros e Arranjos

- Há uma relação estreita entre ponteiros e arranjos (vetores e matrizes) em linguagem C
- Por exemplo, considere as seguintes declarações:
  - **char** str[] = {'b', 'l', 'a', 'b', 'l', 'a', 'b', 'l', 'a', '\0'}, *p1;
    - vetor de 10 chars (string) e ponteiro para chars
  - As seguintes expressões são equivalentes:
    - str[5] **ou** *(str + 5)
    - ambas representam o conteúdo do quinto elemento da string ("a")
      - lembrando que arranjos em C são indexados a partir de zero

16

Ponteiros e Arranjos

- **Nota 1:** embora o nome do arranjo referencie seu 1º elemento, não podemos atribuir ou modificar o valor deste como se fosse um ponteiro qualquer
  - c.c. perderíamos a referência para o início do arranjo
- Por exemplo, o seguinte trecho de código não compila:

```
char str[]={'b','l','a','b','l','a','b','l','a','\0'}, *p1;
while (*str)  putchar(*str++);
```

- Para corrigir:

```
p1 = str;
while (*p1) putchar(*p1++); /* imprime string */
```

17

Ponteiros e Arranjos

- **Nota 2:** a relação entre arranjos e ponteiros é tão próxima que um ponteiro para um arranjo pode ser indexado como se o próprio arranjo fosse (sem *)
- Por ex., o código anterior poderia ser rescrito como:

```
char  str[10] = "blablabla", *p1;
int i;
p1 = str;
for (i=0; i<9; i++) putchar(p1[i]);
```

18

Alocação Dinâmica

- Uma função importante dos ponteiros é que esses permitem realizar **alocação dinâmica** de memória
- Uma variável convencional é declarada com tamanho pré-estabelecido, normalmente no início do programa, e a respectiva região de memória permanece alocada até o término da execução
- Já uma variável dinâmica pode ser gerada sob demanda e descartada a qualquer momento
- Uma variável dinâmica não possui identificador, e portanto deve estar associada a um ponteiro

19

Alocação Dinâmica

- Uma variável dinâmica associada a um ponteiro é gerada com o comando **malloc**(...) e liberada com **free**(...)
  - São as funções básicas de alocação dinâmica em C, mas não são únicas
- Essas funções fazem parte da biblioteca STDLIB
  - Logo, qualquer programa que as utilize deve incluir o cabeçalho `stdlib.h`
  - `#include <stdlib.h>`
- **malloc** recebe o número de bytes a serem alocados e retorna o endereço do primeiro desses bytes, ou seja, um ponteiro
  - ponteiro será **NULL** em caso de insucesso (p. ex. memória insuficiente)
- **free** recebe um ponteiro que tenha sido retornado por malloc

20

Alocação Dinâmica

- Exemplo de alocação dinâmica de memória para 50 inteiros:

```
int *p;
p = malloc(50*sizeof(int)); /* sizeof para portabilidade */
```

- Se o computador utiliza 4 bytes para um **int**, é equivalente a:

```
p = malloc(200);
```

- Os 50 espaços disponíveis para inteiros (tipo do ponteiro) podem ser acessados de forma indexada, como um vetor, ou seja, p[0], ..., p[49]
- Para liberar a memória alocada acima:

```
free(p);
```

21

Passagem de Ponteiros

- Como sabemos, a passagem de parâmetros para funções em C é, em geral, **por valor**
  - Isso significa que uma **cópia** do parâmetro é feita
- Por exemplo, a função abaixo é inócua:

```
void troca(int x, int y) {
  int temp;
  temp = x;
  x = y;
  y = temp; }
```

pois não altera as variáveis passadas como parâmetro

22

Passagem de Ponteiros

- Para forçar uma passagem por **referência**, é preciso utilizar ponteiros
- Nesse caso, a função do exemplo anterior fica:

```
void troca(int *x, int *y) {
  int temp;
  temp = *x;
  *x = *y;
  *y = temp;
}
```

  e troca os valores contidos nos **endereços** passados como parâmetro
- Como é uma chamada para esta função ???
  - Isso explica a sintaxe de scanf ???

Passagem de Ponteiros

- A passagem de arranjos para funções é uma exceção à convenção de passagem por valor da linguagem C
  - Quando uma função possui um arranjo (vetor/matriz) como parâmetro, apenas uma cópia do **endereço** do arranjo é passada na chamada, não uma cópia do arranjo todo
  - Como já sabemos, o endereço do arranjo é um ponteiro para o seu 1º elemento e é referenciado pelo seu próprio nome
- Exemplo:
  - Função para imprimir string sem usar `fprintf("%s", var)`

Passagem de Ponteiros

- Exemplo:

```
void imprime_string(char *str) {
  while (*str)  putchar(*str++); }
```

- Formas alternativas **equivalentes**:
  - `void imprime_string(char str[]) { ... }`
  - `void imprime_string(char str[10]) { ... }`
- Em qualquer caso, a chamada seria:
  - `imprime_string(var); /* var = vetor de chars (string) */`

25

Passagem de Ponteiros

- **Exemplo:**

```
#include <stdio.h>

void imprime_vetor(int vet[], int tamanho);

void main(void){
       int vet[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
       imprime_vetor(vet, 10);
       getchar();
}

void imprime_vetor(int vet[], int tamanho) {
   int cont;
   for (cont=0; cont<tamanho; cont++) printf("%d\n", vet[cont]);
}
```

Retorno de Ponteiros

- Ponteiros também podem ser retornados por funções, assim como outro tipo qualquer
  - "*" precede o nome da função para indicar retorno de ponteiro
- Por exemplo, função que aloca e retorna vetor de inteiros de tamanho estabelecido pelo usuário:

```
int *aloca_vet_int(int tamanho){
    int *pt;
    pt = malloc(tamanho*sizeof(int));
    return *pt;
}
```

27

Retorno de Ponteiros

- Exemplo:

```
#include <stdio.h>
#include <stdlib.h>

int *aloca_vet_int(int tamanho);

void main(void){
        int *p, cont;
        p = aloca_vet_int(10);
        for (cont = 0; cont < 10; cont++) {
             p[cont] = cont;
             printf("%d\n", p[cont]);
        }
        getchar();
        free(p);
}

int *aloca_vet_int(int tamanho){
    return malloc(tamanho*sizeof(int));
}
```

Outros Tópicos sobre Ponteiros

- Tópicos Avançados
  - Matrizes de Ponteiros
  - Ponteiros para Ponteiros
  - Ponteiros para Funções
  - ...
- Para saber mais ...
  - (Schildt, 1997)
  - (Damas, 2007)

29

Exercício

- Definir um tipo de registro (struct) com três campos: um campo numérico real, um campo string e um campo dado por um vetor de inteiros
- Definir um ponteiro para esse tipo de estrutura
- Alocar dinamicamente em memória um registro (estrutura) do tipo definido acima
- Atribuir valores para todos os campos do registro alocado dinamicamente

30

Bibliografia

- Schildt, H. "C Completo e Total", 3a. Edição, Pearson, 1997.
- Damas, L. “Linguagem C”, 10a. Edição, LTC, 2007

31