Programação Estruturada

Ponteiros — Parte 1

Professores Emílio Francesquini e Carla Negri Lintzmayer

2018.Q3

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

UFABC

Ponteiros

Ponteiro

- Ponteiros são tipos especiais de dados que armazenam endereços de memória.
- Uma variável do tipo ponteiro deve ser declarada da seguinte forma:

```
1  tipo *nome_variável;
```

- A variável ponteiro armazenará um endereço de memória de uma outra variável do tipo especificado.
- Exemplo:

```
1  int *mema; float *memb;
```

mema armazena um endereço de memória de variáveis do tipo **int**.
memb armazena um endereço de memória de variáveis do tipo **float**.

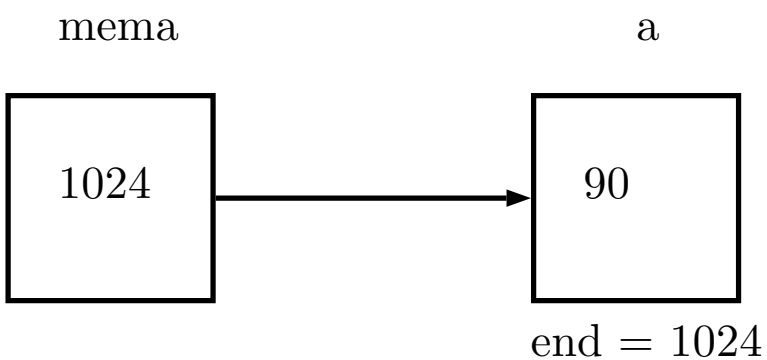
Operadores de Ponteiro

- Existem dois operadores relacionados aos ponteiros:
  - O operador **&** retorna o endereço de memória de uma variável:

```
1  int *mema;
2  int a = 90;
3  mema = &a;
```

  - O operador * acessa o conteúdo do endereço indicado pelo ponteiro:

```
1  printf("%d", *mema);
```



Operadores de Ponteiro

```c
#include <stdio.h>

int main(void) {
    int b;
    int *c;

    b = 10;
    c = &b;
    *c = 11;
    printf("%d\n", b);
}
```

O que será impresso?

Operadores de Ponteiro

```c
#include <stdio.h>

int main(void) {
    int num, q = 1;
    int *p;

    num = 100;
    p = &num;
    q = *p;

    printf("%d\n", q);
}
```

O que será impresso?

Cuidado 1!

- Não se pode atribuir um valor para o endereço apontado pelo ponteiro sem antes ter certeza de que o endereço é válido:

```
1  int a, b;
2  int *c;
3
4  b = 10;
5  *c = 13; /* Vai armazenar 13 em qual endereço? */
```

- O correto seria por exemplo:

```
1  int a, b;
2  int *c;
3
4  b = 10;
5  c = &a;
6  *c = 13;
```

Cuidado 2!

- Infelizmente o operador * de ponteiros é igual ao da multiplicação. Portanto preste atenção em como utilizá-lo.

```c
#include <stdio.h>

int main() {
    int b, a;
    int *c;
    b = 10;
    c = &a;
    *c = 11;
    a = b * c;
    printf("%d\n", a);
    return 0;
}
```

- Ocorre um erro de compilação pois o * é interpretado como operador de ponteiro sobre **c** e não de multiplicação.

Cuidado 2!

- O correto seria algo como:

```c
#include <stdio.h>

int main(void) {
    int b, a;
    int *c;

    b = 10;
    c = &a;
    *c = 11;
    a = b * (*c);
    printf("%d\n", a);
}
```

Cuidado 3!

- O endereço que um ponteiro armazena é sempre de um tipo específico.

```c
#include <stdio.h>

int main() {
    double b, a;
    int *c;
    b = 10.89;
    c = &b; /* Ops! c é ponteiro para int e não double */
    a = *c;
    printf("%lf\n", a);
    return 0;
}
```

- Além do compilador alertar que a atribuição pode causar problemas, é impresso um valor totalmente diferente de 10.89.

Operações com ponteiros

Podemos comparar ponteiros ou o conteúdo apontado por eles:

```
1  int main() {
2      double *a, *b, c, d;
3      b = &c;
4      a = &d;
5
6      if (b < a)
7          printf("O endereco apontado por b eh menor: %p e %p\n", b, a);
8      else if (a < b)
9          printf("O endereco apontado por a eh menor: %p e %p\n", a, b);
10     else if (a == b)
11         printf("Mesmo endereco");
12
13     if (*a == *b)
14         printf("Mesmo conteudo: %lf\n", *a);
15     return 0;
16 }
```

Notem que para imprimir um ponteiro usamos %p.

O valor NULL

- Quando um ponteiro não está associado com nenhum endereço válido é comum atribuir o valor **NULL** para este.
- Isto é usado em comparações com ponteiros para saber se um determinado ponteiro possui valor válido ou não.
- O que é impresso se fizermos: `printf("%p\n", NULL);` ?

```
1  int main() {
2      double *a = NULL, *b = NULL, c = 5;
3      a = &c;
4      if (a != NULL) {
5          b = a;
6          printf("Numero: %lf\n", *b);
7      }
8      return 0;
9  }
```

Passagem de Parâmetros por Valor e por Referência

Passagem de parâmetros

- Quando passamos argumentos para uma função, os valores fornecidos são copiados para as variáveis parâmetros da função. Este processo é idêntico a uma atribuição. **Este processo é chamado de passagem por valor.**
- Desta forma, alterações nos parâmetros dentro da função não alteram os valores que foram passados na chamada da função:

```
1  int main() {
2      int a = 4, b = 5;
3      nao_troca(a, b);
4      return 0;
5  }
6  void nao_troca(int x, int y) {
7      int aux;
8      aux = x;
9      x = y;
10     y = aux;
11 }
```

Passagem de argumentos por referência

- Em C só existe passagem de parâmetros por valor.
- Em algumas linguagens existem construções para se **passar parâmetros por referência**.
  - Neste último caso, alterações de um parâmetro passado por referência também ocorrem onde foi feita a chamada da função.
  - No exemplo anterior, se x e y fossem passados por referência, seus conteúdos seriam trocados.

Passagem de argumentos por referência

Podemos obter algo semelhante em C utilizando ponteiros.

- O artifício corresponde em passar como argumento para uma função o endereço da variável, e não o seu valor.
- Desta forma podemos alterar o conteúdo da variável como se fizéssemos passagem por referência.

```
1  #include <stdio.h>
2  void troca(int *a, int *b);
3  int main() {
4      int x = 4, y = 5;
5      troca(&x, &y);
6      printf("x = %d e y = %d\n", x, y);
7      return 0;
8  }
9  void troca(int *a, int *b) {
10     int aux;
11     aux = *a;
12     *a  = *b;
13     *b = aux;
14 }
```

Passagem de argumentos por referência

- O uso de ponteiros para passar parâmetros que devem ser alterados dentro de uma função é útil em certas situações como:
  - Funções que precisam retornar mais do que um valor.
- Suponha que queremos criar uma função que recebe um vetor como parâmetro e precisa retornar o maior e o menor elemento do vetor.
  - Mas uma função só retorna um único valor!
  - Podemos passar ponteiros para variáveis que "receberão" o maior e menor elemento.

Passagem de argumentos por referência

```c
#include <stdio.h>
void maxAndMin(int vet[], int tam, int *min, int *max);
int main() {
    int v[] = {10, 80, 5, -10, 45, -20, 100, 200, 10};
    int min, max;
    maxAndMin(v, 9, &min, &max);
    printf("O menor eh: %d\nO maior eh: %d\n", min, max);
    return 0;
}
void maxAndMin(int vet[], int tam, int *min, int *max) {
    int i;
    *max = vet[0];
    *min = vet[0];
    for (i = 0; i < tam; i++) {
        if (vet[i] < *min)
            *min = vet[i];
        if (vet[i] > *max)
            *max = vet[i];
    }
}
```

Explicando o `scanf`

Finalmente sabemos o porquê do **&** no `scanf`:

```c
#include <stdio.h>

int main() {
    int x;
    int *p;

    scanf("%d", &x);
    scanf("%d", &p); /* ERRO! */

    p = &x;
    scanf("%d", p);
    return 0;
}
```

Ponteiros e Vetores

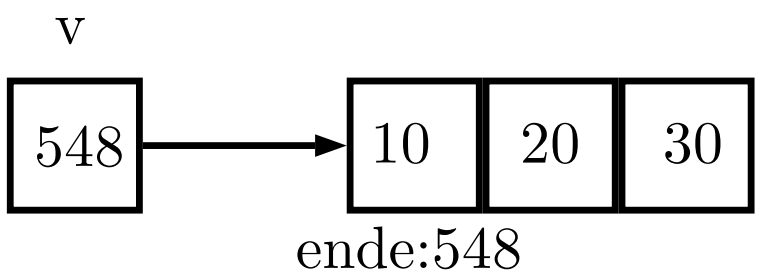
Ponteiros e Vetores

- Quando declaramos uma variável do tipo vetor, é alocada uma quantidade de memória contigua cujo tamanho é especificado na declaração (e também depende do tipo do vetor).

```
1  int v[3]; /* Serão alocados 3*4 bytes de memória. */
```

- Uma variável vetor, assim como um ponteiro, armazena um endereço de memória: **o endereço de início do vetor.**

```
1  int v[3] = {10, 20, 30};
```



- Quando passamos um vetor como argumento para uma função, seu conteúdo pode ser alterado dentro da função pois estamos passando na realidade o endereço inicial do espaço alocado para o vetor.

Ponteiros e Vetores

```c
#include <stdio.h>

void zeraVet(int vet[], int tam) {
    int i;
    for (i = 0; i < tam; i++)
        vet[i] = 0;
}

int main() {
    int vetor[] = {1, 2, 3, 4, 5};
    int i;
    zeraVet(vetor, 5);
    for (i = 0; i < 5; i++)
        printf("%d, ", vetor[i]);
    return 0;
}
```

Ponteiros e Vetores

- Tanto é verdade que uma variável vetor possui um endereço, que podemos atribuí-la para uma variável ponteiro:

```
1  int a[] = {1, 2, 3, 4, 5};
2  int *p;
3  p = a;
```

- E podemos então usar **p** como se fosse um vetor:

```
1  for (i = 0; i < 5; i++)
2      printf("%d\n", p[i]);
```

Ponteiros e Vetores: Diferenças!

- Uma variável vetor, diferentemente de um ponteiro, possui um endereço fixo.
- Isto significa que você não pode tentar atribuir um endereço para uma variável do tipo vetor.

```c
#include <stdio.h>
int main() {
    int a[] = {1, 2, 3, 4, 5};
    int b[5], i;
    b = a;
    for (i = 0; i < 5; i++)
        printf("%d", b[i]);
    return 0;
}
```

Ocorre erro de compilação!

Ponteiros e Vetores: Diferenças!

- Mas se **b** for declarado como ponteiro não há problemas:

```c
#include <stdio.h>
int main() {
    int a[] = {1, 2, 3, 4, 5};
    int *b, i;
    b = a;
    for (i = 0 ; i < 5; i++)
        printf("%d, ", b[i]);
    return 0;
}
```

Ponteiros e Vetores: Diferenças!

O que será impresso pelo programa abaixo?

```c
#include <stdio.h>
int main() {
    int a[] = {1, 2, 3, 4, 5};
    int *b, i;

    b = a;
    printf("Conteudo de b: ");
    for (i = 0; i < 5; i++) {
        printf("%d, ", b[i]);
        b[i] = i*i;
    }
    printf("\nConteudo de a: ");
    for (i = 0; i < 5; i++) {
        printf("%d, ", a[i]);
    }
    printf("\n");
    return 0;
}
```

Extra — Os argumentos do programa

- Muitos programas recebem argumentos/parâmetros diretamente na linha de comando
  - Exemplos: `ls -ah`, `unzip NomeDoArquivo`, `cp ArqOrigem ArqDestino`, ...
- Os argumentos de um programa são recebidos como parâmetros da função `main`.

```
1  int main(int argc, char *argv[]) {
2      ...
3  }
```

- **argc** – *Argument count* – Número de argumentos do seu programa
  - Sempre há pelo menos 1, que é o nome do seu programa
- **argv** – *Argument values* – Vetor de tamanho `argc` de ponteiros para os argumentos do programa (strings)

Extra — Os argumentos do programa — Exemplo

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int i;
    printf("O programa recebeu %d argumentos\n", argc);
    for (i = 0; i < argc; i++)
        printf("\tArgumento %d: %s\n", i, argv[i]);
    return 0;
}
```

Exercícios

Exercício 1

O que será impresso?

```c
#include <stdio.h>

int main() {
    int a = 3, b = 2, *p = NULL, *q = NULL;

    p = &a;
    q = p;
    *q = *q + 1;
    q = &b;
    b = b + 1;

    printf("%d\n", *q);
    printf("%d\n", *p);
    return 0;
}
```

Exercício 2

Escreva uma função **strcat** que recebe como parâmetro 3 strings: **s1**, **s2**, e **sres**. A função deve retornar em **sres** a concatenação de **s1** e **s2**.

Obs: O usuário desta função deve tomar cuidado para declarar **sres** com espaço suficiente para armazenar a concatenação de **s1** e **s2**!