SSC0800 - ICC1 – Teórica

Introdução à Ciência da Computação I

Estruturas Dinâmicas - Ponteiros

Prof. Claudio Fabiano Motta Toledo: claudio@icmc.usp.br

Sumário

- Ponteiros
- Ponteiros e Vetores
- Funções para alocação de memória
- Vetor de ponteiros
- Matrizes e vetores de ponteiros

Ponteiros

- Uma variável é armazenada em um certo número de bytes em uma determinada posição de memória.

- Um ponteiro é uma variável que contém o endereço de outra variável.

- Ponteiros são usados para acessar e manipular conteúdos em determinado endereço de memória.

Ponteiros

- Acesso ao endereço de memória da variável:

  &<nome_var>

- Declaração de um ponteiro:

  <tipo> *<nome_var_ponteiro>

- Exemplos de declaração de ponteiros:

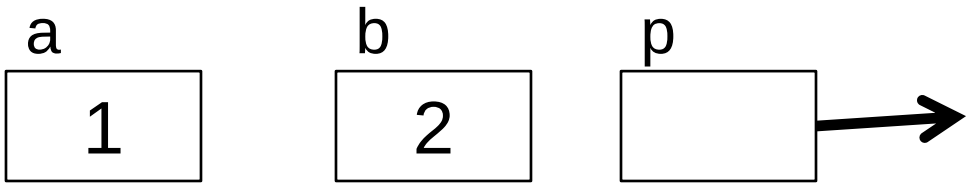
```
int *p;
char *q;
float *r,*s;
```

Ponteiros

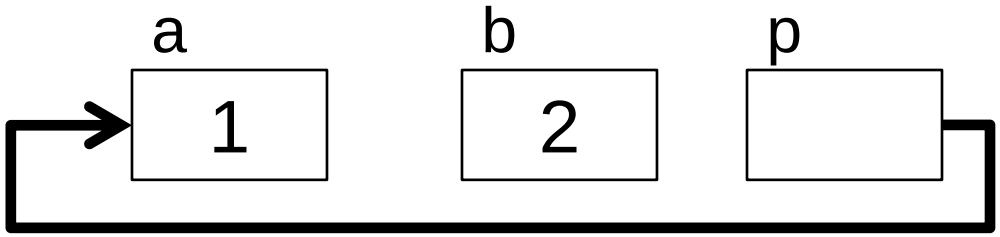
- O operador unário * representa indireção ou deferenciação .
- Se p é um ponteiro, então *p é o valor da variável da qual p é o endereço.
- O valor direto de p é o endereço de memória.
- *p é o valor indireto de p, pois representa o valor armazenado no endereço de memória.

Ponteiros

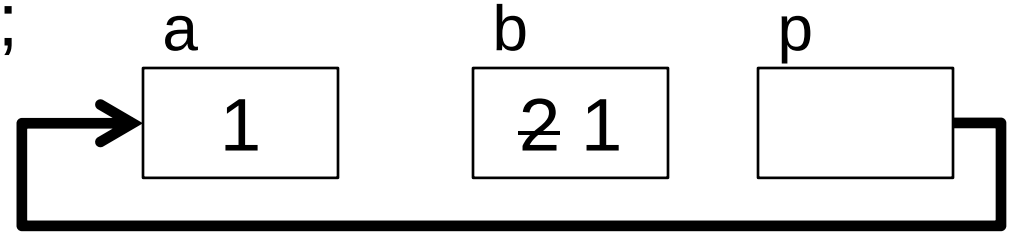
int a=1, b=2, *p;



p=&a



b = *p; ⇔ b=a;



Ponteiros

Exemplo:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int i=7, *p = &i;
    printf("%s%d\n%s%p\n", " Valor de i: ", *p,
           "Endereco de i:", p);
    return 0;
}
```

```
Valor de i: 7
Endereco de i: 0028FF18
```

Ponteiros

Declaração e inicialização		
int i=3, j = 5, *p = &i, *q=&j, *r;		
double x;		
Expressão	**Expressão Equivalente**	**Valor**
p == & i	p == (&i)	1
* * & p	* (* (& p))	3
r = & x	r = (& x)	/* ilegal*/
7 * * p / * q + 7	(((7 * (* p))) / (* q)) +7	11
* (r = & j) *= * p	(* (r = (& j))) *= (* p)	15

Cuidado: (7 * * p / * q + 7) ≠ (7 * * p / * q + 7)

Ponteiros

- Uma declaração void * <nome_ptr> cria um ponteiro do tipo genérico.
- Conversões durante atribuição entre ponteiros diferentes normalmente eram permitidas no C tradicional.
- Porém, conversões de tipo envolvendo ponteiro não são permitidas no padrão ANSI C.
- No ANSIC, a conversão ocorre apenas se um dos tipos é um ponteiro void ou o lado direito é uma constante 0.

Ponteiros

Declaração	
int *p; float *q; void *v;	
Permitido	**Não permitido**
p=0;	p = 1;
p = (int *) 1;	v = 1;
p = v = q;	p = q;
p = (int *) q;	

Ponteiros e Vetores

- Um ponteiro pode acessar diferentes endereços.
- Uma determinada posição em um vetor é um endereço ou um ponteiro que está fixo.
- Considere o vetor **a[ ],** onde **a** aponta para a posição i=0. Temos que:

$$\mathbf{a[\ i\ ]} \Leftrightarrow \mathbf{*(a + i)}$$

- Considere o ponteiro **p**, também temos que:

$$\mathbf{p[\ i\ ]} \Leftrightarrow \mathbf{*(p + i)}$$

Ponteiros e Vetores

- Incrementando/decremento de ponteiros

  int vet[100], *p_vet;

  p_vet = vet;

  p_vet++; *// aponta para o próximo elemento do vetor*

  p_vet--; *// aponta para o elemento anterior do vetor*

  p_vet += 4; *//aponta para a posição atual do vetor + 4;*

Ponteiros e Vetores

```
int vet[] = {10,11,12,13}, *p_vet, cnt;
p_vet =  vet;
for(cnt=0; cnt<3; cnt++){
        printf("\n %d", *p_vet++);
}

p_vet =  vet;
for(cnt=0; cnt<3; cnt++){
        printf("\n %d", *++p_vet);
}
```

Qual a diferença do resultado impresso pelos dois “printf”?

Ponteiros e Vetores

- Considere as declarações abaixo:

```
# define    N     100
int  a[N], i, *p, sum=0;
```

- Temos que:

p = a ⇔ p=&a[0];

p = a+1 ⇔ p=&a[1];

- Suponha que a[N] tenha sido inicializado. As rotinas abaixo são equivalentes

```
for (p=a; p < & a[N]; ++p)
   sum += *p;
```

```
for (i=0; i < N; ++i)
    sum += *(a+i);
```

```
p=a;
for (i=0; i < N; ++i)
    sum += p[ i ];
```

Ponteiros e Vetores

- No exemplo anterior, o vetor **a[N]** tem o identificador **a** como um ponteiro constante.

- Logo, as expressões abaixo são ilegais:

  a = p **++a** **a+=2** **&a**

- Não é possível mudar o valor de **a**.

Ponteiros e Vetores

- Podemos passar parte de um vetor para uma função, passando um apontador para o início do subvetor.

  func(&a[4]) ⇔ func(a+4)

- A declaração dentro da função func() pode ser:

  func(int vet[ ]){...} ou func(int *vet) {....}

- Desde que os limites do vetor sejam obedecidos, pode-se utilizar indexação invertida.

  vet[-1], vet[-2], vet[-3],...

Funções para alocação de memória

- malloc(), calloc(), realloc(), free()
- São funções utilizadas para trabalhar com alocação dinâmica (em tempo de execução) de memória
- A memória é alocada a partir de uma área conhecida como heap

Malloc

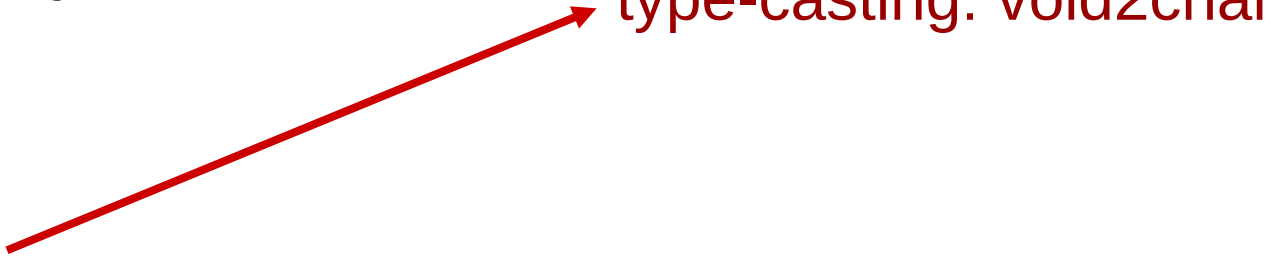
//void *malloc(size_t size);

- size = tamanho do bloco de memória em bytes
- size_t é um tipo pré-definido usado em stdlib.h que faz size_t ser equivalente ao tipo unsigned int.
- Retorna um ponteiro para o bloco de memória alocado
- Quando não conseguir alocar a memória, retorna um ponteiro nulo
- A região alocada contém valores desconhecidos
- Sempre verifique o valor de retorno!

Malloc

```
#include <stdlib.h>


char *str;
if((str = (char *)malloc(100)) == NULL)
{
   printf("Espaco insuficiente para alocar buffer \n");
    exit(1);
}
printf("Espaco alocado para str\n");
```



Malloc

```
#include <stdlib.h>

int *num;
if((num = (int *)malloc(50 * sizeof(int))) ==NULL)
{
    printf("Espaco insuficiente para alocar buffer \n");
    exit(1);
}
printf("Espaco alocado para num\n");
```

Calloc

```
//void * calloc ( size_t  num, size_t  size );
```

- A função calloc() aloca um bloco de memória para um “array” de *num* elementos, sendo cada elemento de tamanho *size*
- A região da memória alocada é inicializada com o valor zero
- A função retorna um ponteiro para o primeiro byte
- Se não houver alocação, retorna um ponteiro nulo

Calloc

```
#include <stdlib.h>
unsigned int num;
int *ptr;
printf("Digite o numero de variaveis do tipo int: ");
scanf("%d", &num);
if((ptr = (int *)calloc(num, sizeof(int))) == NULL)
{
    printf("Espaco insuficiente para alocar \“num\” \n");
    exit(1);
}
printf("Espaco alocado com o calloc\n");
```

Malloc x Calloc

- calloc(n, sizeof(int)); ⇔ malloc(n*sizeof(int));]
- A função malloc() não inicializa o espaço disponibilizado em memória. A função calloc() inicializa com valor zero.
- Em programas extensos, malloc() pode levar menos tempo do que calloc().
- As duas funções retornam um ponteiro do tipo void * em caso de sucesso. Caso contrário, NULL é retornado.

Realloc

//void * realloc (void * ptr, size_t size);

- A função realloc() aumenta ou reduz o tamanho de um bloco de memória previamente alocado com malloc() ou calloc()

- O argumento *ptr* aponta para o bloco original de memória e o *size* indica o novo tamanho desejado em bytes

Realloc

- Possíveis retornos:
  - Se houver espaço para expandir, a memória adicional é alocada e retorna *ptr*
  - Se não houver espaço suficiente para expandir o bloco atual, um novo bloco de tamanho size é alocado numa outra região da memória e o conteúdo do bloco original é copiado para o novo. O espaço de memória do bloco original é liberado e a função retorna um ponteiro para o novo bloco

Realloc

- Possíveis retornos (continuação):
  - Se o argumento *size* for zero, a memória indicada por *ptr* é liberada e a função retorna NULL
  - Se não houver memória suficiente para a realocação (nem para um novo bloco), a função retorna NULL e o bloco original permanece inalterado
  - Se o argumento *ptr* for NULL, a função atua como um malloc()

Exemplo: calloc seguido de realloc

```
unsigned int num; int *ptr;
printf("Digite o numero de variaveis do tipo int: ");
scanf("%d", &num);
if((ptr = (int *)calloc(num, sizeof(int))) == NULL){
      printf("Espaco insuficiente para alocar \"num\" \n");
      exit(1);
 }
//duplica o tamanho da região alocada para ptr
if((ptr = (int *)realloc(ptr, 2*num*sizeof(int))) == NULL){
      printf("Espaco insuficiente para alocar \"num\" \n");
      exit(1);
}
printf("Novo espaço \“realocado\“ com sucesso\n");
```

Free

//void free (void * ptr);

- O espaço alocado dinamicamente com calloc() ou malloc() não retorna ao sistema quando o fluxo de execução deixa uma função.
- A função free() “desaloca”/libera um espaço de memória previamente alocado usando *malloc*, *calloc* ou *realloc*, tornando-o disponível para uso futuro

Free

//void free (void * ptr);

- A função deixa o valor de *ptr* inalterado, porém apontando para uma região inválida (note que o ponteiro não se torna NULL)
- Se for passado um ponteiro nulo, nenhuma ação será realizada.
- Ex: free(ptr);

Vetor de ponteiros

- O exemplo a seguir ilustra a inicialização de um vetor de ponteiros.

- Também é considerada uma função que retorna ponteiro e o uso de um vetor static.

- Suponha que uma função deva retornar um apontador para uma cadeia de caracteres contendo o nome do n-ésimo mês.

Vetor de ponteiros

```
char *nome_mes(int n){

    static char *nome[]={
        "Mes ilegal", "janeiro", "fevereiro", "marco",
        "abril","maio","junho","julho","agosto",
        "setembro","outubro","novembro","dezembro"
    };

    return (n<1 || n>12) ? nome[0]:nome[n];
}
```

Matrizes e vetores de ponteiros

- O identificador **a** é um vetor bidimensional com espaço alocado para 30 caracteres inicializado da seguinte forma:

{ {‘a’, ‘b’, ‘c’,’:’,’\0’} , {‘a’, ‘ ‘, ’i’,’s’,’ ‘, ‘f’, ‘o’, ‘r’, ...} }

a[0] = {‘a’, ‘b’, ‘c’,’:’,’\0’}

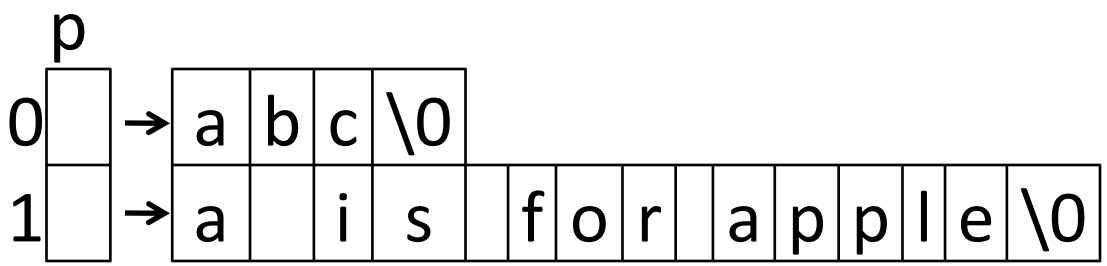
- a[0] e a[1] são cadeias de caracteres (*strings*)

- Apenas 5 elementos estão especificados em a[0], mas há espaço alocado para 15 caracteres. Os demais elementos são iniciados com valor zero (caractere do tipo null).

- Devido à alocação de espaço, qualquer das 30 posições pode ser acessada fazendo a[i][j].

Matrizes e vetores de ponteiros

- O identificador **p** representa um vetor de ponteiros para char.
- A declaração de **p** aloca espaço para dois ponteiros.
- O ponteiro p[0] é iniciado apontando para a cadeia de caracteres “abc:” que requer 5 espaços do tipo char.

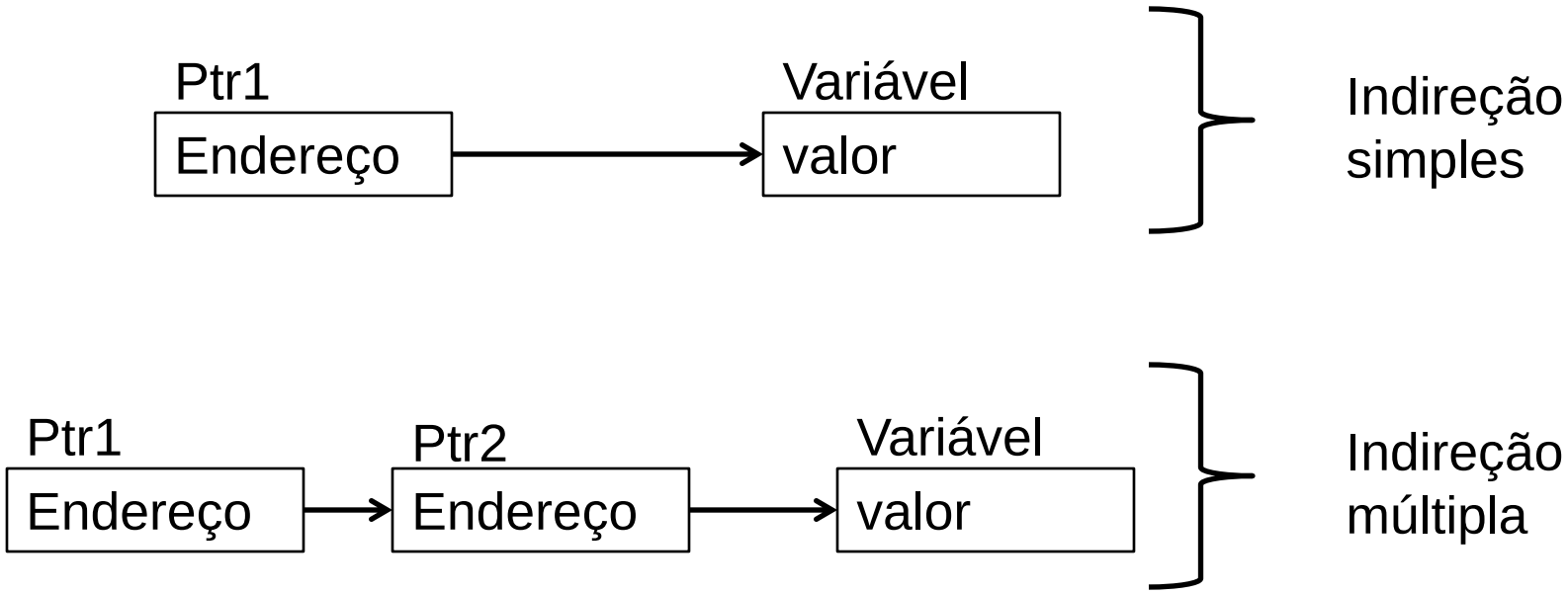


Matrizes e vetores de ponteiros

- **p** trabalha com menos espaço do que **a**.
- Observe que a[0][14] é uma posição válida, mas p[0][14] não.
- As cadeias de caracteres apontadas por p[0] e p[1] não podem ser modificadas, pois p[0] e p[1] são cadeias de caracteres constantes.
- As cadeias de caracteres de a[0] e a[1] podem ser alteradas.

Matrizes e vetores de ponteiros

- **Indireção múltipla** ou **ponteiros para ponteiros** ocorre quando temos ponteiro apontando para outro ponteiro que aponta para determinado valor.



Matrizes e vetores de ponteiros

```
#include <stdio.h>

int main(void){
 int x, *p, **q;
 x=10;
 p=&x;
 q=&p;
 printf("%d",**q);
return 0;
}
```

- Um ponteiro para um ponteiro deve ser declarado com a adição de mais um *.
- int **q;
  - Indica que q é um ponteiro para um ponteiro do tipo int.
- **q
  - acessa o valor apontado pelos ponteiros.

Matrizes e vetores de ponteiros

```
#include <stdio.h>
#include <stdlib.h>
double *alocandoVetor(int *);
double *liberandoVetor(int , float *);

void main (void)
{
  double *vetor;
  int tam;
  vetor = alocandoVetor (&tam);
  printf("tam=%d",tam);
  vetor = liberandoVetor(tam, vetor);
}
```

```
double *alocandoVetor(int *tam)
{
  double *vet;
  do{
     printf ("\nTamanho do vetor:");
     scanf("%d",*tam);
     }while(*tam<1);
  vet = (double *) calloc (*tam,
     sizeof(double));
  if (!vet) {
     printf ("\nEspaço em Memoria
     Insuficiente\n");
     return (NULL);
     }
  return (vet);
}
```

```
double *liberandoVetor(int tam,
     float *vet)
{
  if (!vet) return (NULL);
  free(vet);
  return (NULL);
}
```

Matrizes e vetores de ponteiros

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

double **alocandoMatriz (int *, int *);
float **liberandoMatriz(int, int, float **);

void main (void)
{
  float **matriz;  /* matriz a ser alocada */
  int   ln, cl;   /* numero de linhas e colunas da matriz */
  matriz = alocandoMatriz(&ln, &cl);
  matriz = liberandoMatriz(ln, cl, matriz);
  }
```

```
double **alocandoMatriz (int* linhas, int* colunas)
{
  int   i;
  double **mat;
 //Recebendo num. de linhas e colunas
  do{
      printf("Num. linhas=");
      scanf("%d",linhas);
  }while(linhas<1);

  do{
      printf("Num. colunas=");
      scanf("%d",colunas);
  }while(colunas<1);
```

```
 //Alocação das linhas
  mat = (double **) calloc (*linhas, sizeof(double *));
  if (!mat) {
     printf ("\nEspaço em Memoria Insuficiente\n");
     return (NULL);
  }

//Alocação das colunas
  for ( i = 0; i < *linhas; i++ ) {
       mat[i] = (double*) calloc (*colunas, sizeof(double));
       if (!mat[i]) {
          printf ("Espaço em Memoria Insuficiente");
          return (NULL);
        }
  }
  return (mat);
}
```

```
float **liberandoMatriz(int linhas, int colunas, float **mat)
{
  int  i;

  if (!mat)
     return (NULL);

  //Liberando as linhas da matriz
  for (i=0; i<linhas; i++) free (mat[i]);

  //Liberando a matriz
  free (mat);

  return (NULL);
}
```

Exercício I

- Escreva um programa que:
  - Inicia uma matriz 20x20 com valores aleatórios no intervalo [10,-10].
  - Determina o maior e o menor valor dessa matriz.
  - Determina a quantidade de valores negativos dessa matriz.
- Funções devem ser utilizada para cada tarefa.
- Ponteiros devem ser utilizados sempre que possível.

Referências

Ascencio AFG, Campos EAV. Fundamentos de programação de computadores. São Paulo : Pearson Prentice Hall, 2006. 385 p.

Kelley, A.; Pohl, I*., A Book on C*: programming in C. 4ª Edição. Massachusetts: Pearson, 2010, 726p.

Kernighan, B.W.; Ritchie, D.M. C, *A Linguagem de Programação*: padrão ANSI. 2ª Edição. Rio de Janeiro: Campus, 1989, 290p.

FIM Aula 14