



Linguagem C

Ponteiros e Alocação Dinâmica

MsC. Douglas Santiago Kridi

Programação I - 2018.2

Bacharelado em Ciência da Computação

Universidade Estadual do Piauí

douglaskridi@gmail.com

Introdução

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

tipo * nome_variável ;

Qualquer tipo em C Nome do ponteiro

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

*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.

Introdução

■ Exemplo:

```
float x=-1.0;  
float *p;  
p = &x;
```



Ponteiro `p` no endereço 100
Com conteúdo 500
Aponta para `x`

Fracionário `x` no endereço 500
Com conteúdo -1.0

Operadores

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

```
int *memA;  
int a = 90;  
memA = &a;
```

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

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



Será impresso o valor contido na variável apontada, ou seja, 90.

Operadores

■ Exemplo:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int num, q=1;
    int *p;

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

    printf("%d", q);
    Return 0;
}
```

- O operador * do lado direito da atribuição entrega o *valor* da variável apontada.

■ O que será impresso?

O valor de num: 100

Operadores

■ Exemplo:

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

- O operador * do lado esquerdo da atribuição entrega o *endereço* da variável apontada.

■ O que será impresso?

O novo valor de b: 11

Passagem de parâmetros

- Quando passamos argumentos para uma função, os valores fornecidos são copiados para as variáveis/parâmetros da função.
 - 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 mesma:

```
void nao_troca(int a, int b){  
    int aux;  
    aux = a;  
    a = b;  
    b = aux;  
}
```

```
int main() {  
    int x=4, y=5;  
    nao_troca(x,y);  
    printf("x: %d, y: %d", x, y);  
  
    return 0;  
}
```

Passagem de parâmetros

- 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.

Passagem de parâmetros

- 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.

```
void troca(int *a, int *b){  
    int aux;  
  
    aux = *a;  
    *a = *b;  
    *b = aux;  
}
```

```
int main() {  
    int x=4, y=5;  
    troca(&x, &y);  
    printf("x: %d, y: %d", x, y);  
  
    return 0;  
}
```

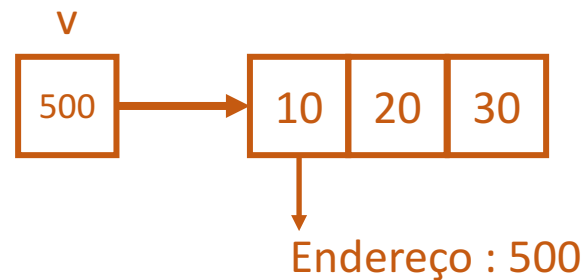
Ponteiros e vetores

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

```
int v[3]; //Serão alocados 3*4 bytes na memória
```

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

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



Ponteiros e vetores

- 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.

```
void zera_vetor(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;  
    zera_vetor(vetor, 5);  
    for(i=0; i<5; i++){  
        printf("%d, ", vetor[i]);  
    }  
    return 0;  
}
```

Ponteiros e vetores

- Como uma variável vetor possui um endereço, podemos atribuí-la para uma variável ponteiro:
 - E podemos então usar a variável como se fosse um ponteiro:

```
int main() {  
    int i, a[] = {1,2,3,4,5};  
    int *p;  
    p = a;  
  
    for(i=0; i<5; i++){  
        printf("%d\t", p[i]);  
    }  
    return 0;  
}
```

Alocação Dinâmica

- A capacidade de armazenamento de um programa esta limitada às variáveis declaradas no código fonte.
 - O espaço destinado à estas variáveis é denominado *memória estática*.
 - Seus tamanhos foram definidos no código fonte e foram *fixados* durante a compilação.
- O tamanho dessa memória *não pode ser modificado* durante a execução.
 - Por exemplo, com memória estática apenas, o programa não pode modificar o tamanho de um vetor já declarado.
- No entanto, é possível aumentar ou diminuir a quantidade de memória em uso, por meio de mecanismos que implementam a chamada *memória dinâmica*.

Alocação Dinâmica

- Para tanto, invocaremos funções especiais para solicitar mais espaço.
 - Uma vez obtido o novo espaço de memória dinâmica, o programa torna-se proprietário do mesmo.
- Ao final de sua execução, o programa deve *devolver* (*liberar*) a memória dinâmica que requisitou.
 - Se o programa requisitar memória dinâmica e não for liberando à medida que não necessitar mais dela, então, poderá *se apropriar de boa parte da memória* disponível na máquina.
 - Levando a situações imprevisíveis, como travamentos.

Alocação Dinâmica

- Para solicitar um trecho contíguo de memória durante a execução do programa, é necessário invocar a função *malloc* (memory allocation).
- Função malloc :
 - O seu único parâmetro é o *número de bytes* que deve ser alocado.
 - A função *devolve o endereço de memória* do início da região que foi alocada ou NULL caso aconteça algum erro.
- É necessário armazenar este endereço em uma variável tipo apontador

Alocação Dinâmica

■ Exemplo:

- Solicitar 1000 bytes e guardar o endereço inicial do trecho de memória obtido em um apontador ap:

```
void *ap;
ap = malloc(1000);
```

O endereço retornado por malloc é totalmente genérico e não possui um tipo especificado.

- Solicitando espaço para um inteiro:


```
int *p;
p = (int*)malloc(4);
```



Quando voce já cria um ponteiro de um tipo específico, o casting (conversão) fica opcional. Tente o exemplo ao lado sem o casting

- Alocação dinâmica de um vetor de 100 inteiros:

```
int *p, i;
p = malloc(100*sizeof(int));
for(i=0; i<100; i++)
    p[i] = i;
```



A função **sizeof()** permite saber o número de bytes ocupado por um determinado tipo de variável.

Alocação Dinâmica

- Função `calloc`:

- Nesta função são passados como parâmetro o número de blocos de memória para ser alocado e o tamanho em bytes de cada bloco.
- A função devolve o endereço de memória do início da região que foi alocada ou `NULL` caso aconteça algum erro.
- A função `calloc` zera todos os bits da memória enquanto que o `malloc` não.

- Exemplo de alocação dinâmica de um vetor de 100 inteiros:

```
int *p, i;  
p = calloc(100, sizeof(int));  
for(i=0; i<100; i++)  
    p[i] = i;
```

Alocação Dinâmica

- É preciso devolver explicitamente o espaço solicitado ao sistema, quando não precisarmos mais dele.
- Para isso, devemos usar a função *free*.
 - Ela recebe como parâmetro um apontador para o espaço de memória que deve ser liberado.
- Uma vez liberado, é impossível acessar novamente este espaço de memória dinâmica.

```
int *p;  
p = calloc(100, sizeof(int));  
free(p);
```

Alocação Dinâmica

■ Exemplo:

```
int *vetor;  
int tamanho;  
scanf("%d", &tamanho);  
  
vetor = (int*)malloc(sizeof(int) * tamanho);  
if (vetor == NULL) {  
    printf("Nao ha memoria suficiente.");  
    return;  
}  
  
vetor[1] = vetor[2] + vetor[3];  
  
free(vetor);
```

Alocação Dinâmica

- A função `realloc` (`realloc` memory) solicita ao sistema *redimensionar um espaço de memória adquirido previamente* com `malloc`.
 - Recebe como parâmetro o apontador para o espaço de memória que desejamos redimensionar e o novo tamanho em bytes.
 - O resultado da solicitação será um apontador para um novo espaço.
- Se o novo tamanho for maior, `realloc` copia os dados já existentes do espaço atual para o novo espaço alocado.
 - Se o novo tamanho for menor que o original, então `realloc` copia tantos bytes quanto possíveis do espaço original para o novo espaço.

Alocação Dinâmica

■ Exemplo:

```
int *v;  
int tamanho, n_tamanho;  
  
scanf("%d", &tamanho);  
v = (int*)malloc(sizeof(int)*tamanho);  
  
scanf("%d", &n_tamanho);  
v = (int*)realloc(v, sizeof(int)*n_tamanho);  
  
free(v);
```

Pratique:

- Lista de Atividades

Bibliografia Básica



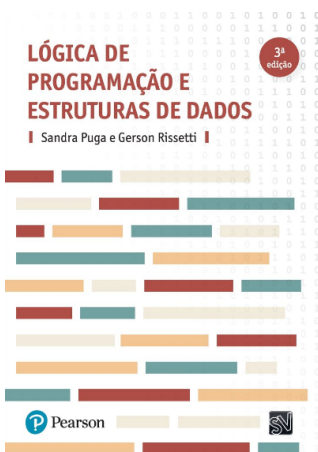
C: COMO PROGRAMAR

DEITEL, Harvey M.; DEITEL, Paul J. Editora Pearson - 6ª ed. 2011

Fundamentos da Programação de Computadores

Ascencio, Ana F. G., Campos, Edilene A. V. de, - Editora Pearson

2012



Lógica de Programação e Estrutura de Dados

Puga, Sandra. Riseti, Gerson. – Ed. Pearson - 2016

Lógica de Programação Algorítmica (Apostila)

Guedes, Sergio. - Editora Pearson/Ser - 2014

