Universidade Federal de Ouro Preto
Campus João Monlevade

CSI103 – Algoritmos e Estruturas de Dados I

Revisão de Conteúdo
Ponteiros e Alocação Dinâmica de Memória

Prof. Mateus Ferreira Satler

Índice

1. Ponteiros

- Toda variável possui um endereço de memória.
- Ponteiro (para um tipo) é um tipo de dado especial que armazena endereços de memória (onde cabem valores do tipo apontado).
- Uma variável que é um ponteiro de um tipo A armazena o endereço de uma outra variável também do tipo A.
- Ponteiros permitem alocação dinâmica de memória, ou seja, alocação de memória enquanto o programa já está sendo executado.

1. Ponteiros

- **Declaração de Ponteiros em C**
  - Variáveis do tipo ponteiro pode ser declaradas assim:

```
tipo *variável;
tipo *variável1, *variável2;
```

  - Exemplos:

```
char *pc; //pc armazena endereço de variável do tipo char

int *pi1, *pi2;
//pi1 e pi2 armazenam endereços de variáveis do tipo int
```

1. Ponteiros

- **Operador &**
  - **O endereço de...**
  - Obtém o endereço de memória da variável à qual é aplicado:

```
int count;
int *m;
count = 5;
m = &count;
```

	End.	Cont.
	0x200	
count	0x300	
	0x400	
m	0x500	
	0x600	
	0x700	
	0x800	

1. Ponteiros

- **Operador &**
  - **O endereço de...**
  - Obtém o endereço de memória da variável à qual é aplicado:

```
int count;
int *m;
count = 5;
m = &count;
```

	End.	Cont.
	0x200	
count	0x300	5
	0x400	
m	0x500	**0x300**
	0x600	
	0x700	
	0x800	

1. Ponteiros

- **Operador ***
  - **A área apontada por...**
  - Acessa o conteúdo que está armazenado no endereço indicado pelo ponteiro ao qual é aplicado.

```
int count, q;
int *m;
count = 5;
m = &count;
q = *m;
*m=10
```

	End.	Cont.
	0x200	
count	0x300	
q	0x400	
m	0x500	
	0x600	
	0x700	
	0x800	

1. Ponteiros

- **Operador ***
  - **A área apontada por...**
  - Acessa o conteúdo que está armazenado no endereço indicado pelo ponteiro ao qual é aplicado.

```
int count, q;
int *m;
count = 5;
m = &count;
q = *m;
*m=10
```

	End.	Cont.
	0x200	
count	0x300	~~5~~ 10
q	0x400	5
m	0x500	**0x300**
	0x600	
	0x700	
	0x800	

1. Ponteiros

- **Inicialização de Ponteiros**
  - Na declaração de um ponteiro, é uma boa prática atribuir a constante **NULL**.
  - Isto permite saber se um ponteiro aponta para um endereço válido.

```
float *a = NULL, *b = NULL,  c=5;
a = &c;

if (a != NULL) {
   b = a;
   printf ("Numero : %.2f", *b);
}
```

1. Ponteiros

- Podemos imprimir o endereço apontado por um ponteiro utilizando a sequência **%p**
- Ponteiros podem ser operados com os operadores de igualdade, relacionais e aritméticos.
- Aritmética de ponteiros permite alterarmos os endereços para os quais um ponteiro aponta e outras operações mais avançadas.

1. Ponteiros

	End.	Cont.
a	0x200	
b	0x300	
c	0x400	
d	0x500	

```
int main () {
   float *a,*b, c, d;
   b = &c;
   a = &d;

   if (b < a)
      printf("O endereco apontado por b e menor:%p < %p\n",b,a);
   else if (a < b)
      printf("O endereco apontado por a e menor:%p < %p\n",a,b);
   else if (a == b)
      printf ("Mesmo endereco: %p == %p\n",a,b);

   if (*a == *b)
      printf("Mesmo conteudo: %f == %f\n", *a, *b);
}
```

1.1. Cuidados com Ponteiros

- Não se pode atribuir um valor para o conteúdo de um endereço (utilizando o operador * sobre um ponteiro) sem se ter certeza de que o ponteiro possui um endereço válido!

ERRADO	CORRETO

`int a, b;`
`int *c;`
 	
`b =10;`
`*c =13;`
`/*Armazena 13 em qual endereço?*/`
	`int a, b;`
	`int *c;`
	
	`b =10;`
	`c = &a;`
	`*c =13;`

	End.	Cont.
	0x200	
a	0x300	
b	0x400	
c	0x500	
	0x600	
	0x700	
	0x800	

1.1. Cuidados com Ponteiros

- Como o operador de conteúdo é igual ao operador de multiplicação, é preciso tomar cuidado para não confundi-los:

ERRADO	CORRETO

`int a, b;`
`int *c;`
 	
`b =10;`
`c = &a;`
`*c =13;`
 	
`a = b * c;`
	`int a, b;`
	`int *c;`
	
	`b =10;`
	`c = &a;`
	`*c =13;`
	
	`a = b * (*c);`

	End.	Cont.
	0x200	
a	0x300	
b	0x400	
c	0x500	
	0x600	
	0x700	
	0x800	

1.1. Cuidados com Ponteiros

- Um ponteiro sempre armazena um endereço para um local de memória que pode armazenar um tipo específico.

ERRADO	CORRETO

`float a, b;`
`int *c;`
 	
`b =10.80;`
`c = &b;`
`/* c é ponteiro para inteiros */`
`a = *c;`
`printf("%f", a);`
	`float a, b;`
	`float *d;`
	
	`b =10.80;`
	`d = &b;`
	`a = *d;`
	`printf("%f", a);`

	End.	Cont.
	0x200	
a	0x300	
b	0x400	
c	0x500	
d	0x600	
	0x700	
	0x800	

1.2. Ponteiros e Vetores

- Quanto declaramos uma variável do tipo vetor, é armazenada uma quantidade de memória contígua de tamanho igual ao declarado.
- Uma variável vetor armazena o endereço de início da região de memória destinada ao vetor.
- Assim, uma variável vetor também é um ponteiro!
- Quando passamos um vetor para uma função, estamos passando o endereço da memória onde o vetor começa: por isto podemos alterar o vetor dentro da função!

1.2. Ponteiros e Vetores

```
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;
}
```

vetor

End.	Cont.
0x100	
0x200	
0x300	
0x400	
0x500	
0x600	
0x700	
0x800	
0x900	

1.2. Ponteiros e Vetores

- Como um vetor armazena um endereço, pode-se atribuir um vetor a um ponteiro para o mesmo tipo dos elementos do vetor:

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

- Logo, é possível utilizar um ponteiro como se fosse um vetor:

```
for( i = 0; i < 5; i++)
    p[ i ] = i * i;
```

	End.	Cont.
a	0x100	
p	0x200	
	0x300	
	0x400	
	0x500	
	0x600	
	0x700	
	0x800	
	0x900	

1.2. Ponteiros e Vetores

- Um ponteiro pode receber por atribuição diferentes endereços.

- Uma variável vetor armazena um endereço fixo.
  - Isto significa que não se pode fazer uma atribuição de endereço a uma variável vetor.

```
int a[] = {1, 2, 3};
int b[3], *p;
p = a; /* Ok. O ponteiro p
recebe o endereço do vetor a */
b = a; /* Erro de compilação!
O vetor b não pode receber o
endereço do vetor a */
```

	End.	Cont.
a	0x100	
b	0x200	
p	0x300	
	0x400	
	0x500	
	0x600	
	0x700	
	0x800	
	0x900	

1.3. Retorno Múltiplo usando Ponteiros

```
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];
    }
}

int main( ){
   int v[] = {10, 80, 5, -10, 45,
              -20, 100, 200, 10};
   int min, max;
   maxAndMin (v, 9, &min, &max);
   printf ("O menor é %d e o maior é: %d\n", min, max);
 }
```

	End.	Cont.
v	0x100	
min	0x200	
max	0x300	
vet	0x400	
tam	0x500	
min	0x600	
max	0x700	
i	0x800	
	0x900	

2. Alocação Dinâmica de Memória

- Pode-se alocar dinamicamente (quando o programa está em execução) uma quantidade de memória contígua e associá-la a um ponteiro.

- Isto permite criar programas sem saber, em tempo de codificação, qual o tamanho dos dados a serem armazenados (vetores, matrizes, etc).

- Desta forma, não é necessário armazenar mais memória do que de fato se deseja usar.

2. Alocação Dinâmica de Memória

- A biblioteca **stdlib.h** possui duas funções para fazer alocação de memória:
  - **void* calloc (int blocos, int tamanho):** recebe o número de blocos de memória a serem alocados e o tamanho de cada bloco. Os bits da memória alocada são zerados.
  - **void* malloc (int qtde_bytes):** recebe a quantidade de bytes a serem alocados na memória. Não zera os bits alocados.
- Se não for necessário zerar os bits da memória alocada, a função **malloc** é preferível por ser mais rápida.

2. Alocação Dinâmica de Memória

- A biblioteca **`stdlib.h`** possui a seguinte função para liberar memória:
  - **`free (void* ponteiro)`**: recebe um ponteiro com o endereço da memória a ser desalocada. Como ela pode receber um ponteiro de qualquer tipo, o tipo do parâmetro deve ser **`void *`**.
- Toda memória alocada com **`calloc()`** ou **`malloc()`** deve ser liberada com **`free()`** após seu uso!

2. Alocação Dinâmica de Memória

- O código abaixo aloca 3 inteiros para o ponteiro **p** e outros 3 inteiros para o ponteiro **q**.
  - Equivale a declararmos 2 vetores de 3 posições!
  - A memória é liberada no final.

```
int *p=NULL, *q=NULL;
p = (int*) calloc(3, sizeof(int));
q = (int*) malloc(3 * sizeof(int));

for (i = 0; i < 3; i++){
    p[ i ] = 1; q[ i ] = 2;
}

free(p); free (q);
```

p

q

End.	Cont.
0x100	
0x200	
0x300	
0x400	
0x500	
0x600	
0x700	
0x800	
0x900	

3. Exercícios

1. Declare e inicialize duas variáveis dos tipos **`int`** e **`float`** e utilize ponteiros dos respectivos tipos para triplicar o valor de cada variável, imprimindo seus conteúdos antes e depois da alteração.

2. Programe o procedimento de assinatura

   ```
   void swap2x (int* a, int* b);
   ```

   que troque o conteúdo entre os ponteiros a e b, duplicando os conteúdos durante a troca.

3. Crie um vetor dinâmico de tamanho informado pelo usuário. Solicite os valores do vetor ao usuário. Em seguida, imprima o conteúdo do vetor.

4. Referências Bibliográficas

- Material de aula do Prof. Ricardo Anido, da UNICAMP: http://www.ic.unicamp.br/~ranido/mc102/

- Material de aula da Profa. Virgínia F. Mota: https://sites.google.com/site/virginiaferm/home/disciplinas

- DEITEL, P; DEITEL, H. C How to Program. 6a Ed. Pearson, 2010.