Ponteiros e alocação dinâmica de memória

Disciplina de Programação de Computadores I
Universidade Federal de Ouro Preto

Agenda

- Ponteiros
- Ponteiros e vetores
- Passagem por cópia e por referência
- Alocação dinâmica de memória
- Operações com ponteiros

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.

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.
```

Operador &

- Obtém o endereço de memória da variável à qual é aplicado

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

variável	count	m
conteúdo	5	0x500
endereço	0x500	0x600

Operador *

- 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
```

variável	count	q	m
conteúdo	~~5~~ 10	5	0x500
endereço	0x500	0x600	0x700

Cuidados com Ponteiros (I)

- 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

```
int a, b;
int *c;

b =10;
*c =13;
//Armazena 13 em qual endereço?
```

Correto

```
int a, b;
int *c;

b =10;
c = &a;
*c =13;
```

Cuidados com Ponteiros (II)

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

Errado

```
int a, b;
int *c;

b =10;
c = &a;
*c =13;


a = b * c;
```

Correto

```
int a, b;
int *c;

b =10;
c = &a;
*c =13;


a = b * (*c);
```

Cuidados com Ponteiros (III)

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

Errado

```
float a, b;
int *c;

b =10.80;
c = &b; //c é ponteiro para inteiros
a = *c;
printf(“%f”, a);
```

Correto

```
float a, b;
float *d;

b =10.80;
d = &b;
a = *d;
printf(“%f”, a);
```

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 : %f", *b);
}
```

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!

Exemplo: Vetor como parâmetro de função

```
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: Semelhanças

- 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;
```

Ponteiros e Vetores: Diferenças

- Uma variável vetor armazena um endereço fixo.
- Um ponteiro pode receber por atribuição diferentes endereços.
- Isto significa que não se pode fazer uma atribuição de endereço a uma variável vetor.

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

Passagem por Cópia e por Referência

- **Passagem por Cópia:** Quando passamos uma variável simples para uma função, o valor da variável é copiado para a variável correspondente no corpo da função e o valor original não sofre as alterações do corpo da função.

- **Passagem por Referência:** Uma variável passada por referência para uma função sofre as alterações do corpo da função, ou seja, as alterações realizadas dentro da função afetam o valor original da variável.

Passagem por Cópia em C

```
void nao_troca(int a, int b){
    int temp;
    temp = a;
    a = b;
    b = temp;
}
void main(){
    int x = 1, y = 2;
    nao_troca(x, y);
}
```

variável	x	y	a	b	temp
conteúdo	1	2	1	2	
endereço	500	600	700	800	900

variável	x	y	a	b	temp
conteúdo	1	2	2	1	1
endereço	500	600	700	800	900

variável	x	y
conteúdo	1	2
endereço	500	600

Simulando Passagem por Referência em C

- A **linguagem C** possui apenas **passagem de parâmetros por cópia**.

- Pode-se simular a **passagem por referência** passando-se como **parâmetro da função** o **endereço da variável**.

- Este endereço será copiado para um ponteiro dentro da função, permitindo que as alterações dentro da função afetem o valor original da variável.

Simulando Passagem por Referência em C

```
void troca(int *a, int *b){
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
int main(){
    int x = 1, y = 2;
    troca(&x, &y);
      return 0;
}
```

variável	x	y	a	b	temp
conteúdo	1	2	500	600	
endereço	500	600	700	800	900

variável	x	y	a	b	temp
conteúdo	2	1	500	600	1
endereço	500	600	700	800	900

variável	x	y
conteúdo	2	1
endereço	500	600

Usos de passagem por referência

- Pode-se retornar mais de um valor em uma função através de passagem por referência.

- Cria-se um procedimento e “retorna-se” os diferentes resultados em parâmetros recebidos por referência no procedimento.

- Como as alterações nestes parâmetros são visíveis fora do procedimento, pode-se retornar mais de um valor como resultado da execução do procedimento!

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);
 }
```

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.

Funções para alocação 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.

Função para liberar 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!

Exemplos de alocação e liberação de memória

- O código abaixo aloca 100 inteiros para o ponteiro p e outros 100 inteiros para o ponteiro q. Equivale a declararmos 2 vetores de 100 posições! A memória é liberada no final.

```
int *p=NULL, *q=NULL;
p = (int*) calloc(5, sizeof(int));
q = (int*) malloc(5 * sizeof(int));

for (i = 0; i < 5; i++){
    p[ i ] = 1; q[ i ] = 2;
}

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

Ponteiros e tipos na memória

- Como o computador sabe onde começa e onde termina a região de memória para p[3], por exemplo?
    - O compilador sabe que p é um ponteiro para inteiros.
    - Ele também sabe que p aponta para um endereço de memória em que são armazenados inteiros.
    - Para encontrar o quarto inteiro (p[3]), o compilador gera código para que o ponteiro p[3] aponte para 3 blocos de memória (cada um do tamanho de um inteiro) depois do endereço de p.

Vetores Unidimensionais Dinâmicos na Memória

```
int aI [3];
char aC[4];
int *pi;
char *pc;
pi =
   (int*)malloc(3*sizeof(int));
pc =
  (char*)malloc(4*sizeof(char));
```

Endereço				
700	pi[0]	pi[1]	pi[2]	
600	pc[0]	pc[1]	pc[2]	pc[3]
500				
400				
300	*pc= 600		*pi= 700	
200	aC[0]	aC[1]	aC[2]	aC[3]
100	aI[0]	aI[1]	aI[2]	

Ponteiros para ponteiros

- Como visto, uma variável ponteiro está alocada na memória como qualquer outra variável.
- Pode-se, então, criar um segundo ponteiro que possua o endereço de memória do primeiro ponteiro.
- Isto se chama ponteiro para ponteiro e pode ser declarado assim:

```
tipo **variavel;
```

- Ex:

```
int ** ppi; char **ppc;
```

Exemplo de ponteiro para ponteiro

```
int main(){
   int a=5, *b, **c;
   b = &a;
   c = &b;
   printf("%d\n", a);
   printf("%d\n", *b);
   printf("%d\n", *(*c));
}
```

variável	a	b	c
conteúdo	5	100	200
endereço	100	200	300

Saída:

5

5

5

Resoluções:

a = 5

* b = *(100)

** c = *(*(200)) = *(&100)

Alocação Dinâmica de Matrizes

- Esta é a forma de se criar matrizes dinamicamente:
  - Crie um ponteiro para ponteiro.

    ```
    int **a
    ```

  - Associe um vetor de ponteiros dinamicamente com este ponteiro de ponteiro. O tamanho deste vetor é o número de linhas da matriz.

    ```
    a = (int**) malloc(n * sizeof(int *));
    ```

  - Cada posição do vetor será associada com um outro vetor do tipo a ser armazenado. Cada um destes vetores é uma linha da matriz (portanto possui tamanho igual ao número de colunas).

    ```
    for (i = 0; i < n; i++)
        a[ i ] = (int*) malloc(m * sizeof(int));
    ```

- Deve-se liberar toda a memória alocada após o uso!

Vetores Multidimensionais Dinâmicos na Memória

```
int **a, n=2, m=3, i;
a = (int**) malloc(n * sizeof(int *));
for (i = 0; i < n; i++)
      a[ i ] = (int*) malloc(m * sizeof(int));
for(i = 0; i < n; i++)
      free(a[ i ]);
free(a);
```

Endereço			
210	a[1][0]	a[1][1]	a[1][2]
190	a[0][0]	a[0][1]	a[0][2]
160			
130		a[0] = 190	a[1] = 210
100	a = 140		

Operações com 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.

Exemplo de comparação de ponteiros

```
int main(void){
    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);
}
```

Aritmética de Ponteiros

- Ponteiros podem ser utilizados nas seguintes operações aritméticas:

    - Ponteiros podem ser incrementados e decrementados
    - Pode-se somar ou subtrair inteiros a ponteiros
    - Um ponteiro pode ser subtraído de outro (resultando na quantidade de elementos do tipo do ponteiro existente no intervalo entre os dois ponteiros!)

Aritmética de Ponteiros

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

int *ptrA, *ptrAA, x;

ptrA = a; // ptrA = &a[0];

ptrA = ptrA + 2 // ptrA = &a[0+2]

ptrA--; // ptrA = &a[1]

ptrAA = &a[3]

x = ptrAA - ptrA // x = (ptrAA - ptrA) / sizeof(int) = (440-410)/10 = 3
```

variável	a[0]	a[1]	a[2]	a[3]	a[4]
conteúdo	1	2	3	4	5
endereço	400	410	420	430	440

ptrA

ptrAA

Só tem sentido sobre Arranjos !

Ponteiros de mesmo tipo

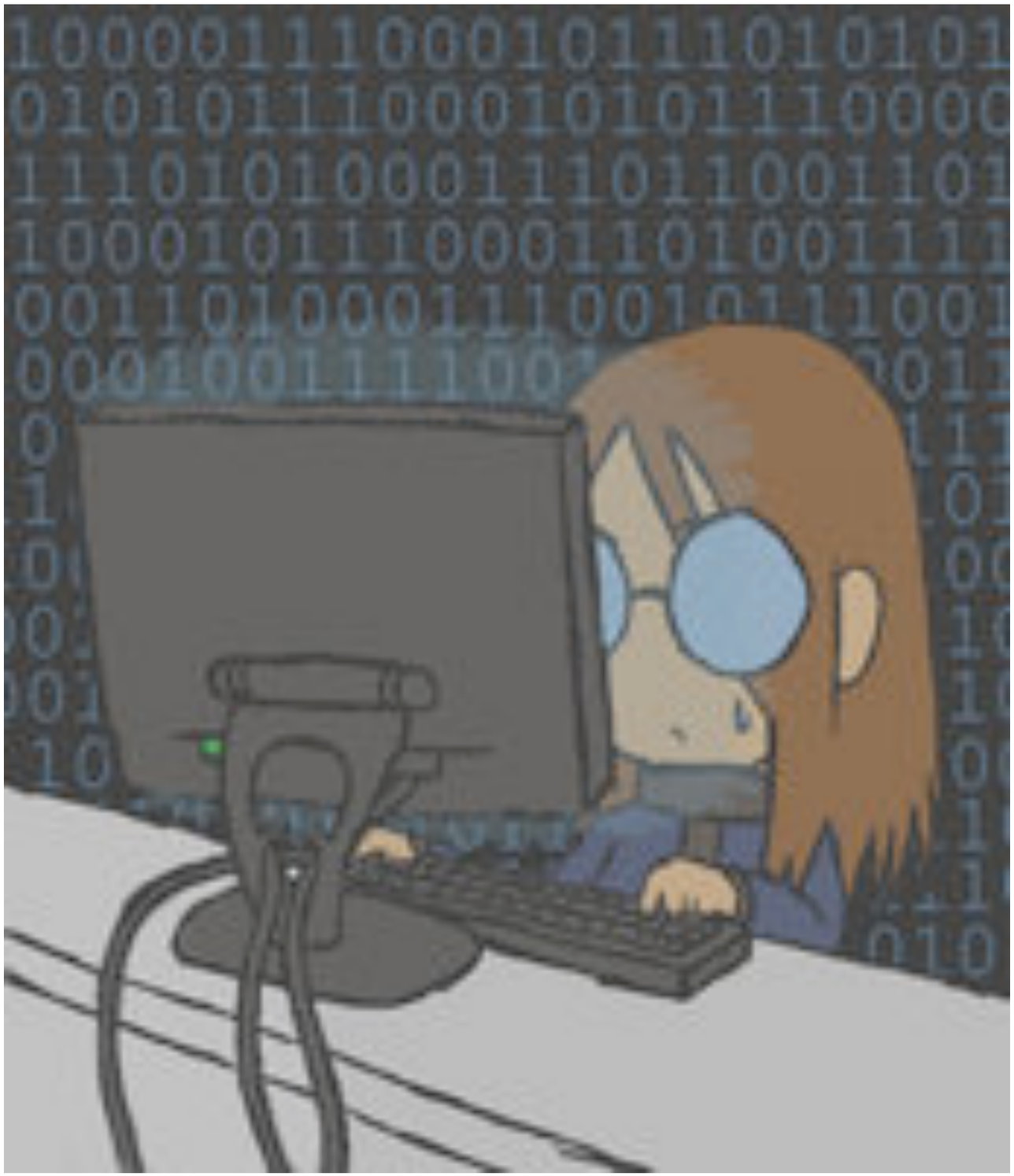
Tipos dos ponteiros

número de elementos entre ponteiros

Relação entre Vetor e Ponteiro

Expressões equivalentes		Valor operado
ptrA = a	ptrA = &a[0]	Endereço
ptrA = &a[3]	ptrA = ptrA + 3	Endereço
x = *&a[3]	x = *(ptrA + 3)	Conteúdo
x = *(ptrA + 3)	x = *(a + 3)	Conteúdo
x = a[3]	x = ptrA[3]	Conteúdo

ptrA += 3 NÃO EQUIVALE A a+=3 porque a é const int *a!



Exercícios

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.