MC-102 — Aula 18
Ponteiros I

Instituto de Computação – Unicamp

25 de Abril de 2016

Roteiro

Ponteiro

- 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;
```

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

Exemplo

```
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**.

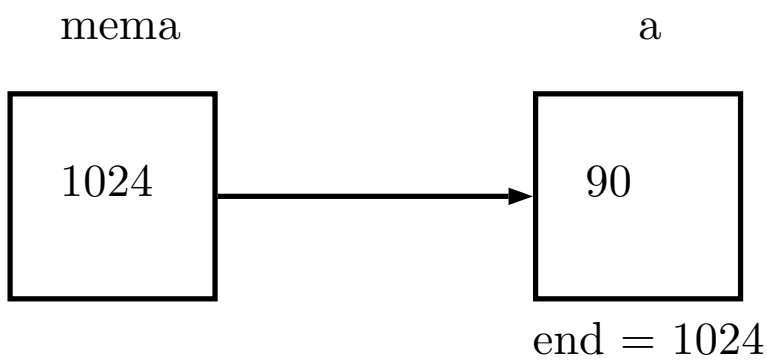
Operadores de Ponteiro

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

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

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

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



Operadores de Ponteiro

```
#include <stdio.h>

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

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

O que será impresso??

Operadores de Ponteiro

```
#include <stdio.h>

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

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

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

O que será impresso??

Cuidado!

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

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

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

- O correto seria por exemplo:

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

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

Cuidado!

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

```
#include <stdio.h>

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

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

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

Cuidado!

- O correto seria algo como:

```
#include <stdio.h>

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

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

Cuidado!

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

```
#include <stdio.h>

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

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

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.

```
int main(void){
  double *a = NULL, *b = NULL,  c=5;
  a=&c;

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

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:

```
int main(){
 int x=4, y=5;
 nao_troca(x,y);
}

void nao_troca(int x, int y) {
  int aux;
  aux = x;
  x = y;
  y = aux;
}
```

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 fizessemos passagem por referência.

```
#include <stdio.h>
void troca(int *a, int *b);

int main(){
 int x=4, y=5;
 troca(&x, &y);
 printf("x = %d  e y = %d\n", x, y);
}
void troca(int *a, int *b) {
  int aux;
  aux = *a;
  *a  = *b;
  *b = aux;
}
```

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

```
#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 e: %d \nO maior e: %d \n",min, max);
}

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];
  }
}
```

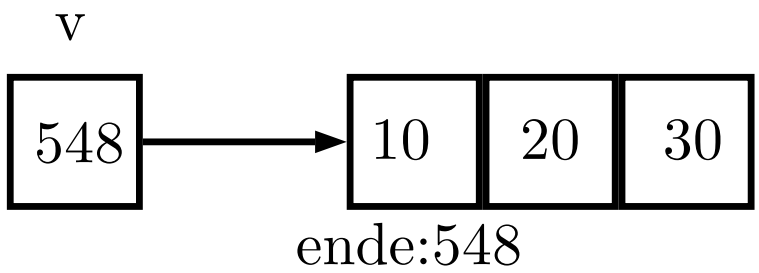
Ponteiros e Vetores

- 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).

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

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

```
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

```
#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]);

}
```

Ponteiros e Vetores

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

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

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

```
for(i = 0; i<5; i++)
  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.

```
#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]);

}
```

Ocorre erro de compilação!

Ponteiros e Vetores: Diferenças!

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

```
#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]);

}
```

Ponteiros e Vetores: Diferenças!

O que será impresso pelo programa abaixo?

```
#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]);
  }


}
```

Exercício

O que será impresso?

```
#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);
}
```

Exercício

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**!