Aula 07 - Ponteiros

Assuntos

1. Ponteiros

Lembrando de vetores...

```c
#include <stdio.h>

int main(){

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

    for(int i=0; i<5; i++)
        printf("%d ", v[i]);

    return 0;
}
```

Na verdade...

Funções em C não recebem vetores e matrizes, só recebem ponteiros

```c
#include <stdio.h>

int main(){

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

    for(int i=0; i<5; i++)
        printf("%d ", *(v+i));

    return 0;
}
```

Ponteiros em C

Quando declaramos uma variável na forma

```
int x = 1;
```

- O compilador sabe onde o valor será armazenado
- Quando usamos a variável, o programa le diretamente o valor na memória
- Neste caso, o acesso é direto

Ponteiros em C

Quando declaramos

```
int *ptr;
```

- Armazenamos para qual local de memória a variável aponta
- O valor de `*ptr` é um endereço de memória e não um valor de inteiro
- Quando usamos a variável, o programa le de forma indireta o valor, pois conhece apenas o endereço de memória

Ponteiros em C

Temos 2 "operadores" de ponteiros:

- `&` : endereço de memória que a variável está armazenada
- `*` : conteúdo da variável apontada pelo ponteiro

Obs: podemos ter ponteiro de ponteiro `**ptr` ...

Obs2: podemos acessar endereços de variáveis que não foram declaradas como ponteiros (por exemplo, `scanf("%d", &i)`)

Resumindo

```c
#include <stdio.h>

int main(){
  int x = 1;
  int *ptr = &x;

  printf("valor de x: %d\n", x);
  printf("valor apontado por ptr: %d\n", *ptr);
  printf("valor armazenado em ptr: %lu\n", ptr);
  printf("endereco de x na memória: %lu\n", &x);
  printf("endereco de ptr na memória: %lu\n", &ptr);
}
```

Mas para que usar ponteiros?

```
void inc(int a){
  printf("valor recebido: %d\n", a);
  a++;
  printf("depois do incremento: %d\n", a);
}

int main(){
  int x = 1;
  printf("valor inicial: %d\n", x);
  inc(x);
  printf("fora da funcao: %d\n", x);
}
```

Mas para que usar ponteiros?

```
void inc(int *a){
  printf("valor recebido: %d\n", *a);
  *a = *a +1;
  printf("depois do incremento: %d\n", *a);
}

int main(){
  int x = 1;
  printf("valor inicial: %d\n", x);
  inc(&x);
  printf("fora da funcao: %d\n", x);
}
```

Mas para que usar ponteiros?

- Funções que precisam alterar valores dos argumentos
- Manipulação de:
  - Strings
  - Structs
  - Matrizes
  - Estruturas de dados mais complexas
- Alocação dinâmica de memória
- Função como argumento de função

Mas para que usar ponteiros?

```
void inc(int *a){
  printf("valor recebido: %lu\n", a);
  a++;
  printf("depois do incremento: %lu\n", a);
}

int main(){
  int x = 1;
  printf("valor inicial: %d\n", x);
  inc(&x);
  printf("fora da funcao: %d\n", x);
}
```

Aritmética de ponteiros

```c
#include <stdio.h>

int main(){

  int *p;
  printf("tamanho de *p: %lu bytes\n", sizeof(p));
  for(int i=0;i<10;i++)
    printf("%lu\n", &p+i);
}
```

Quando incrementamos um ponteiro, incrementamos pelo tamanho da unidade que ele aponta.

Ponteiros e arrays

Voltando para vetores/arrays

- Se vetores são espaços contíguos na memória, então percorrer um vetor é ler endereços de memória de forma sequencial.
- Se um ponteiro armazena um endereço de memória, então ler incrementos de ponteiro é ler endereços de memória de forma sequencial.
- Então qual a diferença entre ponteiros e vetores?

Voltando para vetores/arrays

```c
#include <stdio.h>

int main(){

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

    for(int i=0; i<5; i++){
        printf("%d %d\n", v[i], *(v+i));
    }

    return 0;
}
```

E matrizes?

Uma matriz também é um espaço contíguo na memória, ou seja:

- Podemos ver uma matriz como um ponteiro
- Podemos ver uma matriz como um vetor...

E matrizes?

```
int main(){
  int cs = 3;
  int ls = 3;

    int v[3][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9};

  for(int c = 0; c < cs; c++){
    for(int l = 0; l < ls; l++)
      printf("%d ", v[c][l]);
    printf("\n");
    }
}
```

E matrizes?

```
int main(){
  int cs = 3;
  int ls = 3;

  int v[3][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9};

  for(int c = 0; c < cs; c++){
    for(int l = 0; l < ls; l++)
      printf("%d ", *(*(v+c)+l));
    printf("\n");
    }
}
```

Ou seja

A notação de arrays e matrizes servem para simplificar a escrita do programa e não ter que se preocupar com ponteiros.

Acesso direto	Ponteiro
`v[i]`	`*(v+i)`
`&v[i]`	`v+i`
`m[i][j]`	`*(*(m+i)+j)`
`&m[i][j]`	`*(m+i)+j`

Passagem de parâmetros em funções

```
int soma(int a, int b){
  int s = a + b;
  printf("%d + %d = %d\n", a, b, s);
  a++;
  printf("novo valor de a = %d\n", a);
  return s;
}
```

```
int main(){
  int a = 1;
  int b = 2;
  printf("a = %d, b = %d\n", a, b);

  int s = soma (a, b);
  printf("depois da funcao, a = %d\n", a);
}
```

Passagem de parâmetros por valor

Neste caso:

- as variáveis são passadas por valor e não por referência
- podemos alterar o valor da variável dentro da função que ela não altera o valor fora dela
- a variável `a` dentro da função tem um enderenço de memória diferente da variável `a` fora da função, por isso podemos alterar uma sem alterar a outra

Passagem de parâmetros por referência

```
int soma(int v[], int l){
    int s = 0;

    for(int i = 0; i < l; i++)
        s += v[i];

    return s;
}
```

Passagem de parâmetros por referência

```
int main(){
    int l = 10;
    int v[l];

    for(int i = 0; i < l; i++)
        v[i] = i * i;

    int s = soma(v, l);
    printf("%d\n", s);

    for(int i = 0; i < l; i++)
      printf("%d ", v[i]);
}
```

Passagem de parâmetros por referência

```
int soma(int v[], int l){
    int s = 0;

    for(int i = 0; i < l; i++){
        s += v[i];
        v[i]++;
    }

    return s;
}
```

Qual a diferença entre as execuções?

Passagem de parâmetros por referência

- Quando passamos por valor, podemos alterar a variável sem a preocupação de mudar o valor dela fora da função. Ou seja, só temos acesso ao valor dela.
- Quando passamos por referência, alteramos o valor da variável fora da função quando alteramos o valor dela dentro da função. Ou seja, temos acesso a referência da posição dela na memória

O que fazer?

- **Sempre** procure a documentação da linguagem para saber como os parâmetros são passados!
- Tomar cuidado com a alteração do valor dentro da função
- Sempre verificar se está passando o parâmetro por valor ou referência
- Aproveitar para "retornar" mais do que um valor em uma função