

Aula 07 - Ponteiros, passagem de parâmetros e alocação de memória

Assuntos

1. Ponteiros
2. Passagem por valor vs passagem por referência
3. Alocação de memória

Lembrando de vetores ...

```
#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

```
#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 lê 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 lê 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

```
#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("endereço de x na memória: %lu\n", &x);
    printf("endereço 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

```
#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

```
#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 matrices?

```
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 matrices?

```
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
<code>v[i]</code>	<code>*(v+i)</code>
<code>&v[i]</code>	<code>v+i</code>
<code>m[i][j]</code>	<code>*(*(m+i)+j)</code>
<code>&m[i][j]</code>	<code>*(m+i)+j</code>

Passagem por valor e referência

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 endereço 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

Alocação de memória

De vetores ...

```
#include <stdio.h>

int main() {
    int *p = {1, 2, 3, 4, 5};
}
```

Por que isso não funciona?

De vetores ...

- `int v[] = {1, 2, 3, 4, 5};` aloca o espaço de 5 inteiros na memória;
- `int *p` aloca o espaço de apenas 1 inteiro, então não acessamos as outras posições
- Ou seja, precisamos alocar memória para ter mais do que um ponteiro de inteiro em um espaço contíguo na memória ...

Alocação de memória (em C)

- alocação estática: criada e gerenciada em tempo de execução pelo próprio computador (o que fizemos até agora)
- alocação dinâmica: criada e gerenciada pelo programador (o que vamos fazer agora)

Alocação de memória (em C)

Em C temos duas áreas de memória:

- Stack:
 - variáveis estáticas com gerenciamento automático
 - funciona como uma pilha
- Heap:
 - gerenciado pelo programador

Alocação estática

- Quando uma variável é declarada, seu valor é armazenado na stack
- Quando o programa é encerrado, todos os valores na stack são apagados

```
#include <stdio.h>

int main() {
    int i = 1; // coloca o valor de i na stack
    int j = 2; // coloca o valor de j na stack

    printf("%d %d\n", i, j); // le os valores da stack
} // limpa todos os valores da stack
```

Alocação dinâmica

- Necessário incluir `stdlib.h` e usar ponteiros
- Alocação e liberação de memória por chamada:
 - `malloc` : `void* malloc (sizeof(...))` aloca a memória e retorna um ponteiro para a primeira posição
 - `free` : `void free (void* p)` desaloca a memória do ponteiro `p`

Exemplo

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *p;                                // criamos um ponteiro.
    p = (int *) malloc(5*sizeof(int));     // alocamos memória.
    for (int i = 0; i < 5; i++)            // colocamos valores
        *(p + i) = i;                     // em cada posição.
    for (int i = 0; i < 5; i++)            // lemos os valores
        printf("%d ", *(p + i));          // de cada posição.
    free(p);                               // liberamos a memória.
    for (int i = 0; i < 5; i++)            // tentamos ler valores
        printf("%d ", *(p + i));          // de cada posição.
}
```

Exemplo

Neste caso:

```
int main(){  
    int *p; // *p está na stack e aponta para o heap  
  
    // stack aponta para heap  
    p = (int *) malloc(5*sizeof(int));  
  
    *(p + i) = i; // alteramos o valor no heap  
  
    free(p); // liberamos o conteúdo na stack  
} // liberamos os valores do heap
```