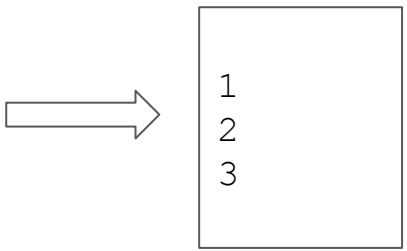
Ponteiros (Parte 2)

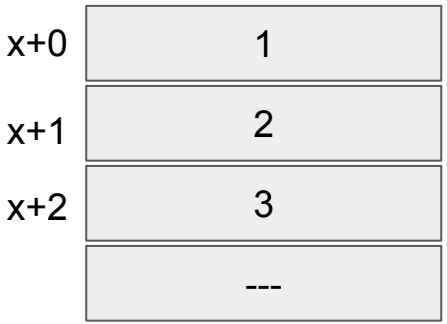
Programação Estruturada - Cris Sato

Vetores vs. ponteiros

- São tipos diferentes mas com algumas compatibilidades.
- `int *x`:
  - `x[0]` é o mesmo que `*(x+0)` : valor guardado na posição apontada por `x`
  - `x[1]` é o mesmo que `*(x+1)` : o valor guardado na posição seguinte à apontada por `x`
  - `x[i]` é o mesmo que `*(x+i)` : o valor guardado na i-ésima posição à apontada por `x`

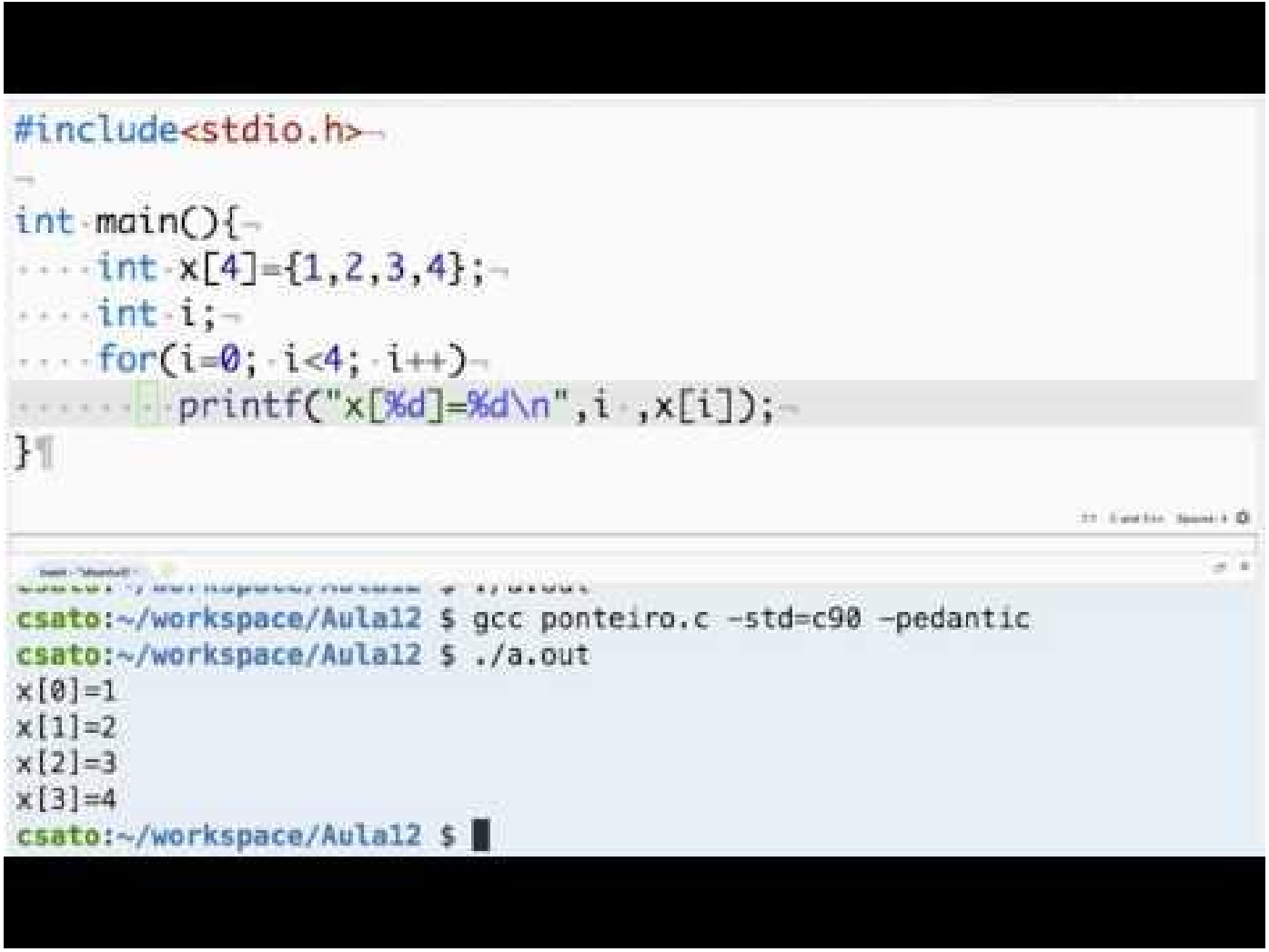
```
int v[3]={1,2,3};
int *x = v;
printf("x[0]=%d\n",x[0]);
printf("x[1]=%d\n",x[1]);
printf("x[2]=%d\n",x[2]);
```





`x[i]` é o mesmo que `i[x]`

- Quando acessamos uma variável `x[i]`, o compilador na verdade traduz isso para `*(x+i)`.
- Mas pensando assim, `i[x]` não deveria ser traduzido para `*(i+x)`? Estranho?
- É exatamente isso que o compilador faz!


#include<stdio.h>
int main(){
int x[4]={1,2,3,4};
int i;
for(i=0; i<4; i++)
printf("x[%d]=%d\n",i ,x[i]);
}
csato:~/workspace/Aula12 $ gcc ponteiro.c -std=c90 -pedantic
csato:~/workspace/Aula12 $ ./a.out
x[0]=1
x[1]=2
x[2]=3
x[3]=4
csato:~/workspace/Aula12 $

Por que funciona?

- `int v[3] = {1,2,3}`
- `int *x = v`
- Por que funciona? Não são tipos diferentes?
- `x[0] = *(x+0)= *x.` Assim, `&(x[0]) = x`
- x é convertido para um ponteiro para o seu primeiro elemento

```
int x[5]={1,2,3,4,5};
printf("%p\n",x);
printf("%p\n",&x);
printf("%p\n", &x[0]);
printf("%d\n", *x);
```

```
0x7ffef1cde6b0
0x7ffef1cde6b0
0x7ffef1cde6b0
1
```

Vetores como parâmetro

```
void muda (int *a)
{
     a[0]=1;
     a[1]=2;
}
```

```
void muda (int a[])
{
     a[0]=1;
     a[1]=2;
}
```

```
void muda (int a[2]) {
     a[0]=1;
     a[1]=2;
}
```

```
-- main --
int a[2] = {0};
printf("a[0]=%d\n",a[0]);
printf("a[1]=%d\n",a[1]);
muda(a);
printf("a[0]=%d\n",a[0]);
printf("a[1]=%d\n",a[1]);
```

```
a[0]=0
a[1]=0
a[0]=1
a[1]=2
```

As três maneiras têm o mesmo efeito! Note que na primeira maneira, tratamos o vetor como um ponteiro.

Matrizes

- Declarando uma matriz de inteiros com 2 linhas e 3 colunas: char m[2][3]
- Como a matriz fica organizada na memória? A memória em si está organizada como um vetor.
- A matriz fica alocada em 2*3 posições **contíguas** da memória. Ou seja, ela parece um vetor.

Matriz

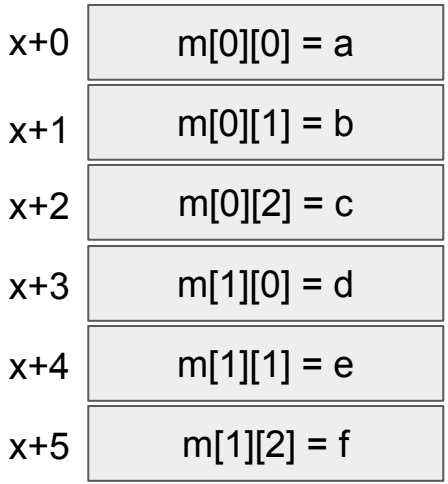
```
char m[2][3];
m[0][0]='a'; m[0][1]='b'; m[0][2]='c';
m[1][0]='d'; m[1][1]='e'; m[1][2]='f';
char *x = &m[0][0];
int i;
for(i=0; i<2*3; i++)
    printf("x[%d] = %c\n", i, x[i]);
```

```
x[0] = a
x[1] = b
x[2] = c
x[3] = d
x[4] = e
x[5] = f
```

x+0	m[0][0] = a
x+1	m[0][1] = b
x+2	m[0][2] = c
x+3	m[1][0] = d
x+4	m[1][1] = e
x+5	m[1][2] = f

Matriz

- Note que se queremos ir de m[0][0] para m[1][0] (linha seguinte), temos que somar 3 a x (e 3 é o número de colunas)



Matriz como parâmetro

```
void muda (int
m[2][2]){   int i, j;
for(i=0; i<2; i++)
  for(j=0; j<2; j++)
    m[i][j]=1;
}
```

OU

```
void muda (int m[][2]){
int i, j;
for(i=0; i<2; i++)
  for(j=0; j<2; j++)
    m[i][j]=1;
}
```

```
--main--
int m[2][2];
int i, j;
for(i=0; i<2; i++)
  for(j=0; j<2; j++)
    m[i][j]=0;
imprime(m);
muda(m);
imprime(m);
```

```
0 0
0 0
1 1
1 1
```

Matriz como parâmetro

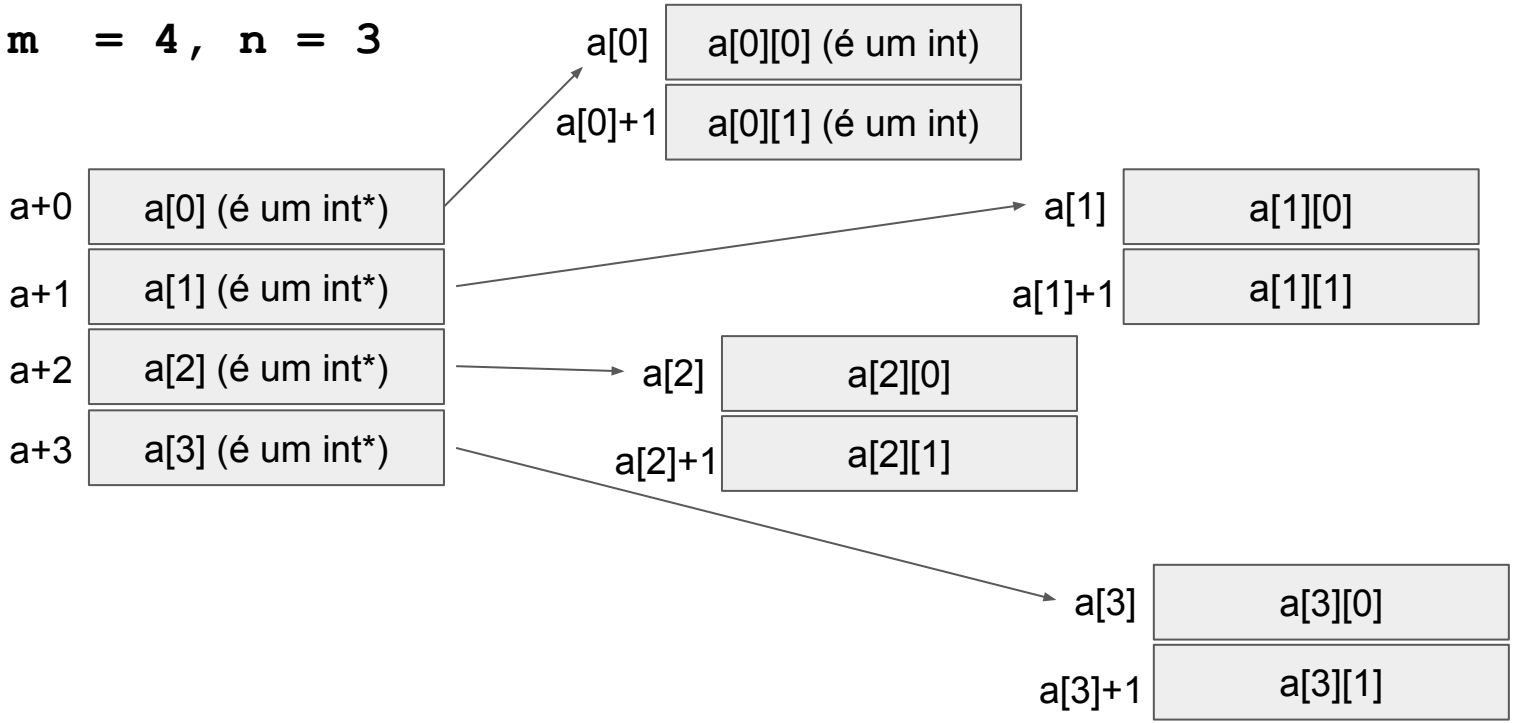
```
void muda (int x, int y,int m[x][y]){
int i, j;
for(i=0; i<x; i++)
 for(j=0; j<y; j++)
   m[i][j]=1;
}
```

Funciona para C99 e para C90 como uma extensão GNU (quer dizer que não está especificado no padrão C90, mas o GCC aceita).

Matrizes alocadas dinamicamente (int **)

- As matrizes que veremos agora não são organizadas do mesmo jeito que as que acabamos de ver. Não são matrizes exatamente. Seria mais correto dizer que são arrays de arrays.
- `int **`: é um ponteiro para um ponteiro de `int`
- Como usar isso para representar algo como uma matriz?
- Suponha que queiramos que a nossa matriz tenha m linhas:
- `int ** a = malloc(m*sizeof(int *))`
- Por que `sizeof(int *)`? Porque cada linha será acessada por um `int *`
- Suponha que queiramos que cada linha tenha n colunas.
- Para cada linha i, fazemos: `a[i] = malloc(n*sizeof(int))`
- Note que se você quisesse, você poderia alocar tamanhos diferentes para cada linha

Matrizes alocadas dinamicamente (int **)



Matrizes alocadas dinamicamente (int **)

- Note que as posições de uma linha para a outra não são necessariamente contíguas (pois fizemos vários mallocs)
- Mas podemos acessá-las usando a mesma notação de matriz, onde `a[i][j]` significa o mesmo que `*(*(a+i)+j)`
- Vamos analisar a expressão `*(*(a+i)+j)`:
- `a+i` é um `int**`
- `*(a+i)=a[i]` é um ponteiro (`int *`) apontando para a linha i
- `a[i]+j` é um ponteiro (`int *`) apontado para a j-ésima posição após `a[i]`
- `*(a[i]+j) = a[i][j]`

Exemplo de int ** como parâmetro

```
-- main --
int **m = malloc(2*sizeof(int *));
int i, j;
for(i=0; i<2; i++) {
  m[i] = malloc(3*sizeof(int));
  for(j=0; j<3; j++)
    m[i][j]=0;
}
imprime(m);
muda(m);
imprime(m);
```

```
void muda (int **m){
  int i, j;
  for(i=0; i<2; i++)
    for(j=0; j<3; j++)
      m[i][j]=1;
}
```

```
0 0 0
0 0 0
1 1 1
1 1 1
```

Outro tipo de matriz

- Suponha que você sabe quantas linhas a matriz vai ter (digamos que 3).
- Mas você não sabe quantas colunas ela vai ter.
- `int *a[3]` é um array de `int *` com 3 posições
- Para alocar cada linha (digamos com tamanhos 1, 2 e 3):
  - `a[0] = malloc(1*sizeof(int))`
  - `a[1] = malloc(2*sizeof(int))`
  - `a[2] = malloc(3*sizeof(int))`

Passagem de parâmetro

```
void muda (int **m){
int i, j;
for(i=0; i<3; i++)
     for(j=0; j<(i+1); j++)
          m[i][j]=1;
}
```

OU

```
void muda (int *m[3]){
int i, j;
for(i=0; i<3; i++)
  for(j=0; j<(i+1); j++)
    m[i][j]=1;
}
```

```
-- main --
int *m[3], i, j;
for(i=0; i<3; i++) {
  m[i] = malloc((i+1)*sizeof(int));
  for(j=0; j<(i+1); j++)
    m[i][j]=0;
}
imprime(m); muda(m); imprime(m);
```

```
0
0 0
0 0 0
1
1 1
1 1 1
```

Curiosidade

- `int *a[3]` é o mesmo que `int *(a[3])` pois colchetes têm precedência sobre o *.
- O que temos nesse caso é um vetor de 3 posições de int *.
- E o que seria `int (*a)[3]`?
- É um ponteiro para `int[3]`!

Ponteiro para funções

- Cada função que escrevemos é traduzida para instruções e alocada na memória.
- Como está na memória, ela tem um endereço! Portanto, podemos passar funções como parâmetro usando o seu endereço!

Funções como parâmetro

```
int comp1 (int x1, int x2) {
  return x1-x2;
}
```

```
int comp2 (int x1, int x2){
  return x2-x1;
}
```

```
-- main --
printf("O endereco de
comp1 é %p\n", comp1);
printf("O endereco de
comp2 é %p\n", comp2);
```

```
O endereco de comp1 é
0x40057d
O endereco de comp2 é
0x400593
```

Funções como parâmetro

```
int comp1 (int x1, int x2) {
  return x1-x2;
}
```

```
int comp2 (int x1, int x2){
  return x2-x1;
}
```

```
void func(int a, int b, int (*comp)(int, int)){
  if(comp(a,b) < 0)  printf("%d<%d\n",a,b);
  else if(comp(a,b)>0)  printf("%d>%d\n",a,b);
  else printf("%d=%d\n",a,b);
```

```
-- main --
printf("Comparando 1 e 2
usando comp1\n");
func(1,2, comp1);

printf("Comparando 1 e 2
usando comp2\n");

func(1,2, comp2);
```

```
Comparando 1 e 2 usando comp1
1<2
Comparando 1 e 2 usando comp2
1>2
```

Exemplo - qsort para ordenar vetores

- O qsort da stdlib pode ser usado para ordernar vetores de quaisquer tipos.
- Ele tem a seguinte assinatura:
- **void qsort(void *base, size_t nitems, size_t size, int (*compar)(const void *, const void*))**
  - **base** -- ponteiro para o primeiro elemento a ser ordenado
  - **nitems** --número de elementos a serem ordenados
  - **size** -- O tamanho em bytes de cada elemento do vetor
  - **compar** -- a função que compara os elementos
- Suponha que queremos ordenar um vetor int nums[10] da posição 2 a 8
- base = nums+2, nitems =7, size = sizeof(int)

Exemplo - qsort para ordenar vetores

- E a função compar?
- **int (*compar)(const void *, const void*)**
- Ela é bastante geral: os parâmetros estão com o tipo `const void*`
- Como fazer as nossas funções comp1 e comp2 ficarem nesse padrão?

```
int comp1 (int x1, int x2)  ⟶  int comp1 (const void* x1, const void* x2)
                                {
```

comp1 passa a receber parâmetros const void *
Mas agora temos que ajustar o corpo da função!

```
int comp1 (int x1,
return x1-x2;
}
```

- x1 é agora um const void*. Então convertemos para int * usando um typecast: `(int *) x1`
- Agora temos que pegar o valor apontado por `(int *) x1: *(int *)x1`
- A função `comp1` fica:

```
int comp1 (const void *x1, const void *x2){
return *(int *)x1 - *(int *)x2;
}
```

- Nossa função de comparação devolve
  - um valor negativo se queremos dizer x1<x2
  - um valor positivo se queremos dizer x1>x2
  - zero se queremos dizer x1=x2
- O qsort usa o mesmo padrão

Vamos usar o qsort!

```
-- main --
int i;
int nums[10]={2,4,6,2,4,3,8,3,9,1};
imprime(2,8);
qsort (nums+2, 7, sizeof(int), comp1);
imprime(2,8);
```

```
6 2 4 3 8 3 9
2 3 3 4 6 8 9
```

```
int comp1 (const void *x1, const void *x2){
return *(int *)x1- *(int *)x2;
}
```

Em ordem crescente!

Trocando a função de comparação!

```
-- main --
int i;
int nums[10]={2,4,6,2,4,3,8,3,9,1};
imprime(2,8);
qsort (nums+2, 7, sizeof(int), comp2);
imprime(2,8);
```

```
6 2 4 3 8 3 9
9 8 6 4 3 3 2
```

Em ordem decrescente!

```
int comp2 (const void *x1, const void *x2){
return *(int *)x2- *(int *)x1;
}
```