DEI
DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
TÉCNICO LISBOA

Ponteiros e Tabelas

K&R: Capítulo 5

IAED

Cenas dos próximos capítulos

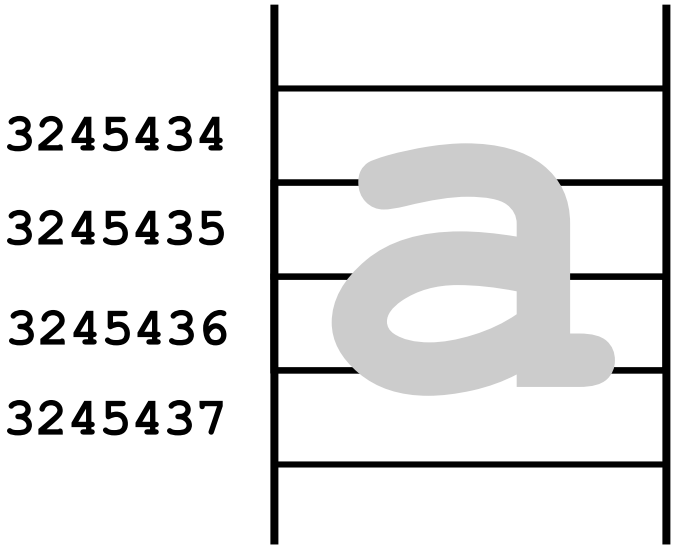
- Ponteiros e Tabelas
- Alocação Dinâmica de Memória
- Estruturas, Funções e Apontadores
- Estruturas Auto-Referenciadas
- Exemplo de aplicação:
  - Listas ligadas
  - Pilhas

Ponteiros e Tabelas

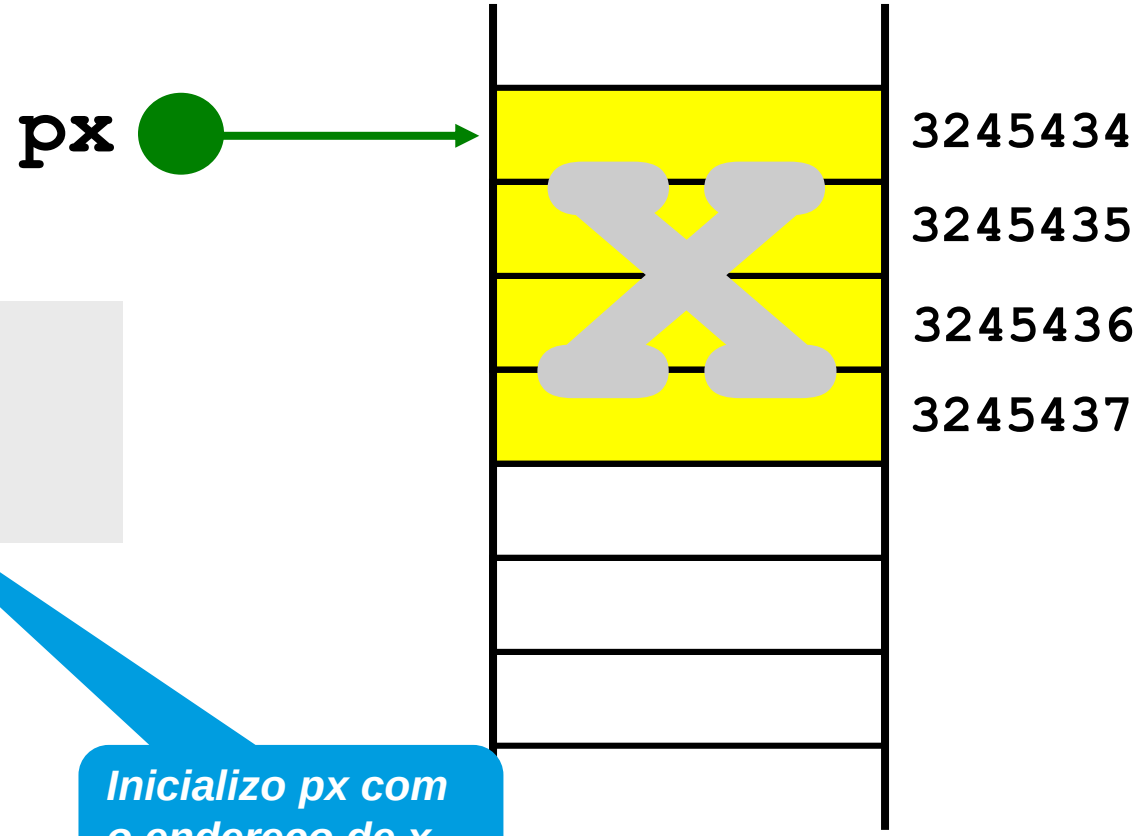
- Ponteiros e endereços
- Ponteiros e argumentos de funções
- Ponteiros e tabelas
- Alocação dinâmica de memória
- Aritmética de ponteiros
- Tabelas de ponteiros e ponteiros para ponteiros
- Tabelas multi-dimensionais
- Inicialização de tabelas de ponteiros
- Argumentos da linha de comandos

Ponteiros e Endereços

- Na memória do computador cada posição é referenciada por um endereço, atribuído de forma sequencial
- Posições adjacentes têm endereços consecutivos
- Um ponteiro é uma variável que contém um endereço de outra variável.



Ponteiros e Endereços: exemplo




```
int x = 10;
int *px = &x;
```

Declaração de um ponteiro

Inicializo px com o endereço de x (operador &)

O que é
um ponteiro em C ?

Um ponteiro em C

é

um endereço de memória

Declaração de ponteiros

- Na sua declaração temos de indicar ao compilador para que tipo de variável estamos a endereçar.

- Declaração de um ponteiro para tipo `<tipo>`

```
<tipo> *<variável>;
```

- Exemplos

```
char *cptr;      /* ponteiro para caracter */

int *iptr;       /* ponteiro para inteiro */

double *dptr;    /* ponteiro para double */
```

Declaração de ponteiros

- Na sua declaração temos de indicar ao compilador para que tipo de variável estamos a endereçar.

- Declaração de um ponteiro para tipo `<tipo>`

```
<tipo> *<variável>;
```

- Exemplos

```
/* apenas a é um ponteiro*/
int* a, b;

/* c e d são ponteiros para floats */
float *c, *d;
```

Operador &

- O endereço de uma variável é obtido através do **operador &**.

- Exemplos

```
int a = 43;   /* um inteiro inicializado a 43 */
int* iptr;    /* ponteiro para inteiro */
iptr = &a;    /* iptr passa a guardar o endereço de a */
```

Operador &

- O endereço de uma variável é obtido através do **operador &**.

- Exemplos

```
int a = 43;   /* um inteiro inicializado a 43 */
int *iptr;    /* ponteiro para inteiro */
iptr = &a;    /* iptr passa a guardar o endereço de a */
```

Operador &

- O endereço de uma variável é obtido através do **operador &**.

- Exemplos (também poderia inicializar `iptr` na mesma linha):

```
int a = 43;       /* um inteiro inicializado a 43 */
int *iptr = &a;
/* declaro um ponteiro para inteiro e esse ponteiro passa
   a guardar o endereço de a */
```

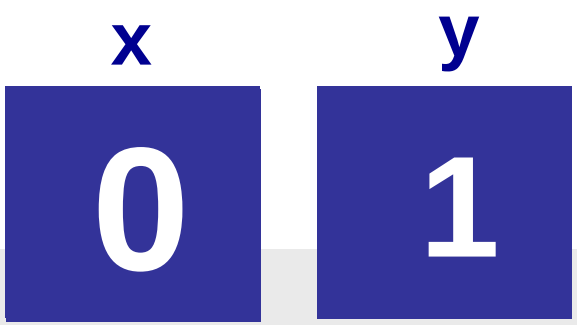
Operador *

- O **operador** * permite aceder ao conteúdo de uma posição de memória endereçada pelo ponteiro
  (i.e., o conteúdo para onde um ponteiro "aponta").
- **O valor guardado num determinado endereço é dado pelo operador** *

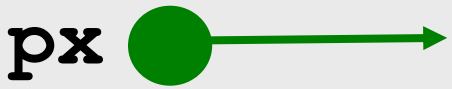
```
int a = 43;   /* um inteiro inicializado a 43 */
int *iptr;    /* ponteiro para inteiro */
int b;

iptr = &a;    /* iptr passa a guardar o endereço de a */
b = *iptr;    /* b passa a guardar o valor apontado por
                 iptr */
```

Operadores & e *

- Outro exemplo:




```
#include <stdio.h>

int main() {
  int y, x = 1;
  int *px;

  px = &x;
  y = *px;
  *px = 0;
  printf("%d %d\n", x, y);

  return 0;
}
```



Operadores & e *

- Outro exemplo:

```
#include <stdio.h>

int main() {
  int y, x = 1;
  int *px;

  px = &x;
  y = *px;
  *px = 0;
  printf("%d %d\n", x, y);

  return 0;
}
```

Declaro dois inteiros, sendo x=1

px é um ponteiro para inteiros... Mas ainda não guarda o endereço de nenhum

Agora sim... px fica a guardar o endereço de x

y toma o valor guardado no endereço de memória guardado em px, i.e., o valor 1

Alteramos o conteúdo da posição de memória para onde px aponta, ou seja, vamos alterar o valor de x

Output: 0 1

Confusão habitual: utilizações do * (asterisco)

- **Declaração** do ponteiro

```
int *x;
```

  - `x` é um ponteiro para um inteiro

- **Conteúdo** da posição de memória apontada pelo ponteiro

```
*x = 4;
```

  - o valor `4` é atribuído ao conteúdo da posição de memória apontada por `x`

Utilização de Ponteiros

- O valor de retorno de uma função pode ser um ponteiro

```
int* xpto();
```

- O argumento de uma função pode ser um ponteiro

```
int abcd(char *a, int *b);
```

Passagem de Parâmetros para Funções

- Em C os parâmetros são passados por valor

```
void swap(int a, int b) {
  int aux;

  aux = a;
  a = b;
  b = aux;
}
```

- Chamada `swap(x, y);`
- A troca não é efectuada: Não funciona como necessário !

Passagem de Parâmetros para Funções

- Passagem por referência consegue-se enviando ponteiros

```
void swap(int *a, int *b) {
  int aux;

  aux = *a;
  *a = *b;
  *b = aux;
}
```

- Chamada deverá ser `swap(&x, &y)`

Passagem de Parâmetros para Funções

- Outro exemplo: Qual o output deste código?

```
void incrementa(int t){
   t++;
}

int main(){
   int k=0;
   incrementa(k);
   printf("%d\n",k);
   return 0;
}
```

Passagem de Parâmetros para Funções

- Outro exemplo: Qual o output deste código? E deste?

```
void incrementa(int* t){
   (*t)++;
}

int main(){
   int k=0;
   incrementa(&k);
   printf("%d\n",k);
   return 0;
}
```

Ponteiro Nulo / Endereço Zero

- Ponteiro especial para representar o endereço `0`

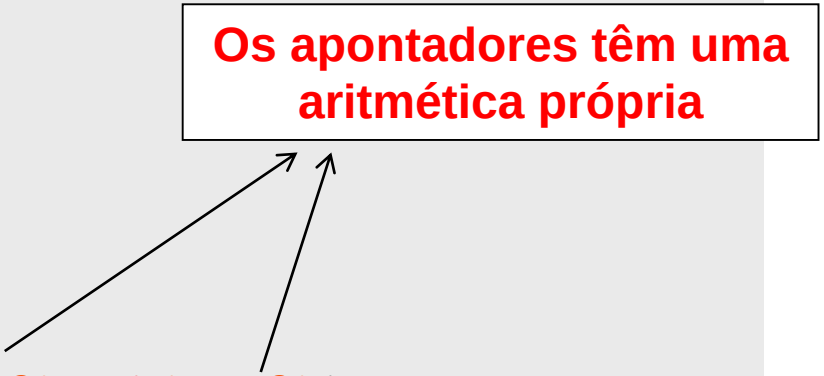
```
int *ptr = NULL;
```

- Definido em `stdlib.h`
  - Necessário `#include <stdlib.h>`
- Utilizado para indicar situações especiais
- Na realidade `NULL == 0`

Ponteiros e Tabelas

- Em C existe uma relação entre ponteiros e tabelas

```
#include <stdio.h>

int main() {
  int a[6] = {1, 2, 7, 0, 11, 6};
  int *pa = a;

  printf("%d %d %d\n", a[2], *(a+2), *(pa+2));

  return 0;
}
```

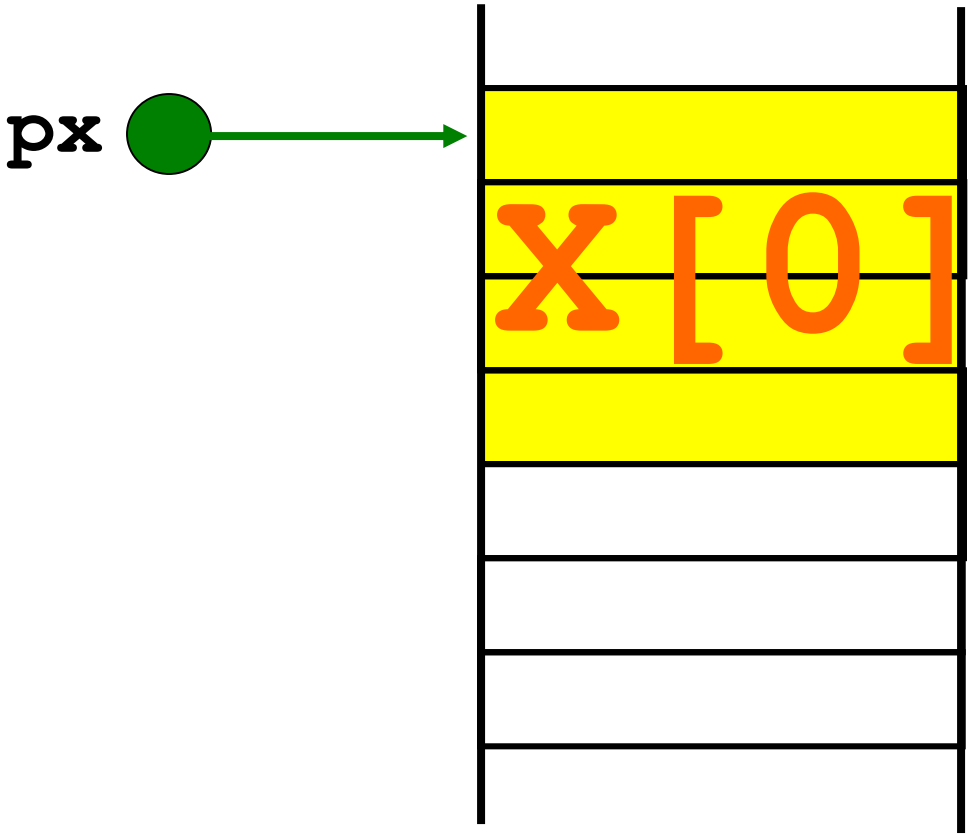


- `a` é um ponteiro para a primeira posição da tabela

Ponteiros e Tabelas

- É possível efectuar + e – com ponteiros

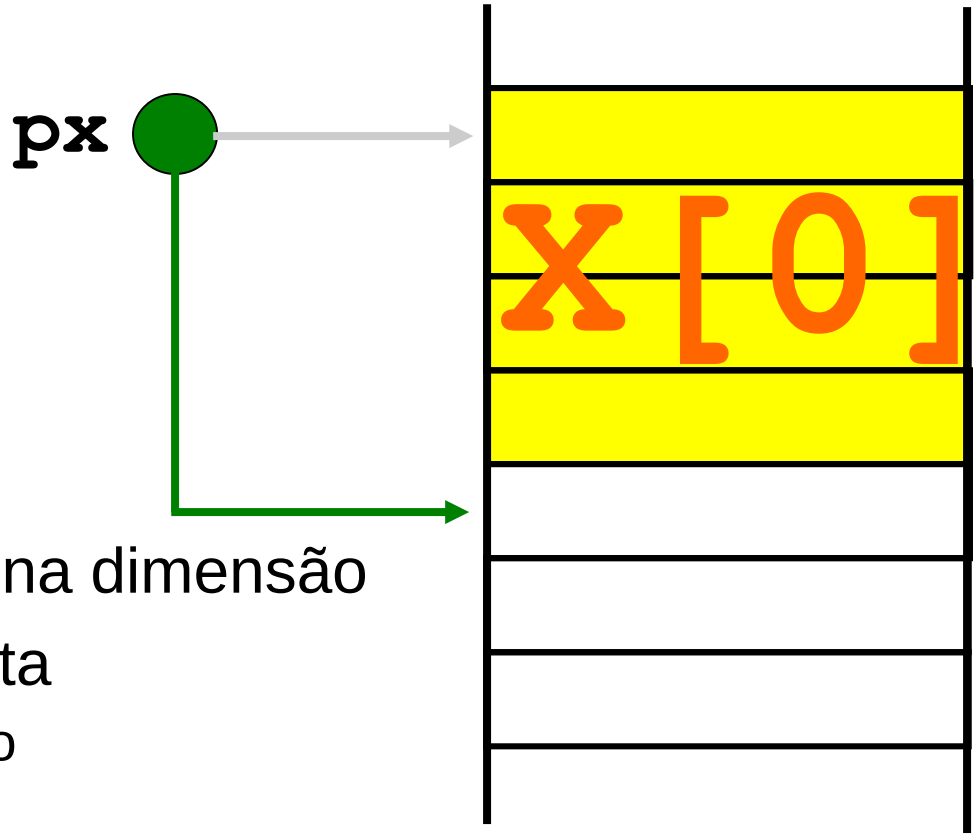
```
int x[10];
int *px = x;
```



Ponteiros e Tabelas

- É possível efectuar + e – com ponteiros

```
int x[10];
int *px = x;

px++;
```



- Incrementa/decrementa na dimensão do tipo para o qual aponta
  - `sizeof(int)` neste caso

Ponteiros e Tabelas

- Em C existe uma relação entre ponteiros e tabelas

```
#include <stdio.h>

int main() {
  int a[6] = {1, 2, 7, 0, 11, 6};
  int *pa = a;

  printf("%d %d %d\n", a[2], *(a+2), *(pa+2));

  return 0;
}
```

- `a` é um ponteiro para a primeira posição da tabela

Exercício

- Seja

```
float a[100];
float *p=a;
```

- então

`a[i]`	é equivalente a	`*(a+i)`	V
`&a[i]`	é equivalente a	`a+i`	V
`a[i]`	é equivalente a	`p[i]`	V
`p[i]`	é equivalente a	`*(p+i)`	V
`*a ?`	`a`	é equivalente a	`p[0]`	F
`a[0] ?`				

Ponteiros e Tabelas

- A declaração `int *p1;` declara o mesmo que `int p2[];`
  - `p1` pode ser alterado
  - `p2` não pode ser alterado
  - `int p2[];` só pode ser utilizado em certos casos

- A declaração `int p3[100];` declara uma tabela com 100 inteiros e aloca memória na quantidade necessária
  - `p3` não pode ser alterado

- A declaração `char *text;` não aloca qualquer memória
  - no entanto `char *text = "ola";` aloca;

Ponteiros e Tabelas

- Qual a diferença entre as duas declarações seguintes ?

```
char t1[] = "ola";
char *t2 = "ola";
```

- Ambas alocam 4 bytes e copiam para essa posição de memória a sequência de caracteres `'o','l','a','\0'`
- No caso **`t1`** é possível modificar o conteúdo da memória alocada
- Não é possível alterar o valor de `t1`, ou seja não é possível pôr `t1` a endereçar outra posição de memória
- É possível alterar o valor de `t2`

Passagem de Parâmetros para Funções II

- Quando fazemos

```
int a;
scanf("%d",&a);
```

o que estamos a passar ao scanf ?

```
char s[100];
scanf("%s",s);
```

- Porque não precisamos do **&** ?

Passagem de Parâmetros para Funções II

- Passagem por referência consegue-se enviando ponteiros

```
void leVector(int v[], int tamanho) {
  int i;
  for (i=0 ; i<tamanho ; i++)
      scanf("%d",&v[i])
}
```

- Podemos escrever o argumento como `int* v` ou `int v[]`

Passagem de Parâmetros para Funções II

- Passagem por referência consegue-se enviando ponteiros

```
void leVector(int *v, int tamanho) {
  int i;
  for (i=0 ; i<tamanho ; i++)
      scanf("%d",&v[i])
}
```

- Podemos escrever o argumento como `int* v` ou `int v[]`

Passagem de Parâmetros para Funções II

- Passagem por referência consegue-se enviando ponteiros

```
void leVector(int *v, int tamanho) {
  int i;
  for (i=0 ; i<tamanho ; i++)
      scanf("%d",v+i}
}
```

- Podemos escrever o argumento como **`int* v`** ou **`int v[]`**
- Como **`v`** já é um endereço podemos alterar o **`v`** dentro da função.

Passagem de Parâmetros para Funções II

- Passagem por referência consegue-se enviando ponteiros

```
void leVector(int *v, int tamanho) {
  int i;
  for (i=0 ; i<tamanho ; i++)
      scanf("%d",v++}
}
```

- Podemos escrever o argumento como `int* v` ou `int v[]`
- Como `v` já é um endereço podemos alterar o `v` dentro da função.

Ponteiros e Tabelas

- Exemplo: cópia de *strings*

```
void strcpy(char s[], char t[]) {
  int i = 0;
  while ((s[i] = t[i]) != ´\0´)
    i++;
}
```

Ponteiros e Tabelas

- Exemplo: cópia de *strings*

```
void strcpy(char s[], char t[]) {
  int i = 0;
  while ((s[i] = t[i]) != '\0')
    i++;
}
```

```
void strcpy(char *s, char *t) {
  while ((*s++ = *t++));
}
```

Porquê?

Ponteiros e Tabelas

```
void strcpy(char *s, char *t) {
  int i = 0;
  while ((s[i] = t[i]) != ´\0´)
    i++;
}
```

Ponteiros e Tabelas

```
void strcpy(char *s, char *t) {
  int i = 0;
  while ((s[i] = t[i]) != ´\0´)
    i++;
}
```

```
void strcpy(char *s, char *t) {
  int i = 0;
  while ((*(s+i) = *(t+i)) != ´\0´)
    i++;
}
```

Ponteiros e Tabelas

```
void strcpy(char *s, char *t) {
  int i = 0;
  while ((s[i] = t[i]) != '\0')
    i++;
}
```

```
void strcpy(char *s, char *t) {
  int i = 0;
  while ((*(s+i) = *(t+i)) != '\0')
    i++;
}
```

```
void strcpy(char *s, char *t) {
  while ((*s = *t) != '\0') {
    s++;
    t++;
  }
}
```

Ponteiros e Tabelas

```
void strcpy(char *s, char *t) {
  while ((*s = *t) != '\0') {
    s++;
    t++;
  }
}
```

Ponteiros e Tabelas

```
void strcpy(char *s, char *t) {
  while ((*s = *t) != '\0') {
    s++;
    t++;
  }
}
```

```
void strcpy(char *s, char *t) {
  while ((*s++ = *t++) != '\0');
}
```

Ponteiros e Tabelas

```
void strcpy(char *s, char *t) {
  while ((*s = *t) != '\0') {
    s++;
    t++;
  }
}
```

```
void strcpy(char *s, char *t) {
  while ((*s++ = *t++) != '\0');
}
```

```
void strcpy(char *s, char *t) {
  while ((*s++ = *t++));
}
```

Endereços de ponteiros

- É possível declarar um ***ponteiro para um ponteiro***

```
#include <stdio.h>

int main() {
  int x = 10;
  int *px = &x;
  int **ppx = &px;

  printf("%d %d %d\n", x, *px, **ppx);

  return 0;
}
```

1ª nota

- Ao fazer

```
int *a;
```

apenas estamos a reservar memória para 1 endereço de memória e não para um inteiro.

- Por esta razão, não devemos inicializar o conteúdo de um ponteiro sem que saibamos exactamente onde ele está a escrever. Ex.:

```
int *a;
*a=12; /* A evitar!!! */
```

2ª nota: Ponteiros para Funções

- É possível ter ponteiros para funções
- O nome de uma função é um ponteiro para essa função

```
int soma(int a, int b) { return a+b; }

int main() {
  int (*ptr)(int, int);

  ptr = soma;

  printf("%d\n", (*ptr)(3,4));
  return 0;
}
```

2ª nota: Ponteiros para Funções (2º exemplo)

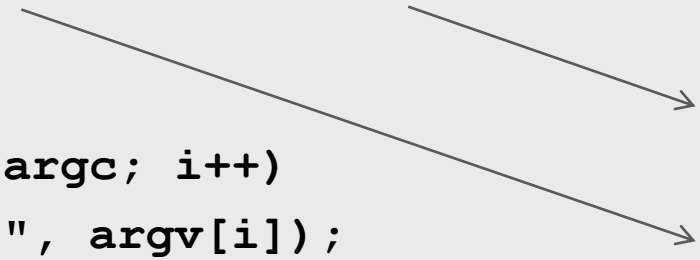
- É possível ter ponteiros para funções
- O nome de uma função é um ponteiro para essa função

```
int modulo(int a) { return a < 0 ? -a : a; }
int dobro(int a) { return a*2; }

void escreve(int (*func)(int), int valor){
   printf("%d\n",(*func)(valor));
}
int main() {
  int x = -10;
  int (*f)(int);
  f = modulo;
  escreve(f,x);
  return 0;
}
```

Argumentos da Linha de Comandos

- `argv[0]` é o nome o programa
- `argv[i]` é `i`-ésimo argumento
- Programa ”escreve"
- `$ escreve hello world` gera `hello world`

```
int main(int argc, char *argv[]) {
  int i;

  for(i=1; i < argc; i++)
    printf("%s ", argv[i]);
  printf("\n");
  return 0;
}
```



array de pointers

Tamanho do array
(número de argumentos introduzidos)