Arquitetura de Computadores

LEI – 2021/22

DI-FCT/NOVA

Linguagem de Programação C

Sumário

- A linguagem de programação C
  - Introdução à linguagem C
  - Compilação
  - Ler e escrever dados
  - Apontadores
  - Arrays

Bibliografia:

- Brian W. Kernighan, Dennis M. Ritchie. The C Programming Language. Caps 1 a 5.
- Mike Banahan, Declan Brady and Mark Doran, *The C Book*. Caps 1 a 5. (https://publications.gbdirect.co.uk/c_book/)

Introdução ao C

- A linguagem C marcou a sintaxe de várias linguagens posteriores, como o Java
  - Muito é semelhante
  - Também há muitas diferenças
    - No C não há objetos/classes
    - Tem funções (são como métodos estáticos do Java — sem ligação a qualquer objeto)
    - A biblioteca *standard* é completamente diferente

C vs Java

- Construções de controlo:

```
if (exp) {
        inst;
} else {
        inst;
}
```

```
while ( exp ) {
  inst;
}
```

```
for ( inst; exp; inst ) {
  inst;
}
```

```
switch (expr) {
  case val: inst; break;
  ....
  default: inst;
}
```

```
do {
   inst;
} while (exp);
```

C vs Java

- Comentários:

```
/* bloco
 */

// linha
```

- Blocos de código e escopo das declarações:

```
{ declarações;
  instruções;
}
```

C tem pre-processador

- Diretivas iniciadas por #

  `#define XPTO 234` - no ficheiro, substitui XPTO por 234

  - Permite pseudo-constantes, e mais...

  `#include "file.h"` - inclui o file.h a partir de um caminho relativo à diretoria corrente

  `#include <file.h>` - inclui o file.h a partir de um caminho relativo à diretoria standard

  - Permite introduzir as definições de funções e tipos fora do nosso ficheiro como das bibliotecas (interfaces)
  - Também chamados ficheiros de *headers*

Funções C vs métodos Java

- As funções são semelhante aos métodos Java, mas:
  - Não pertencem a objetos. Não existe `this`
  - São como os métodos `static`
  - Não há polimorfismo (um nome = uma função)

Exemplo *Hello World*

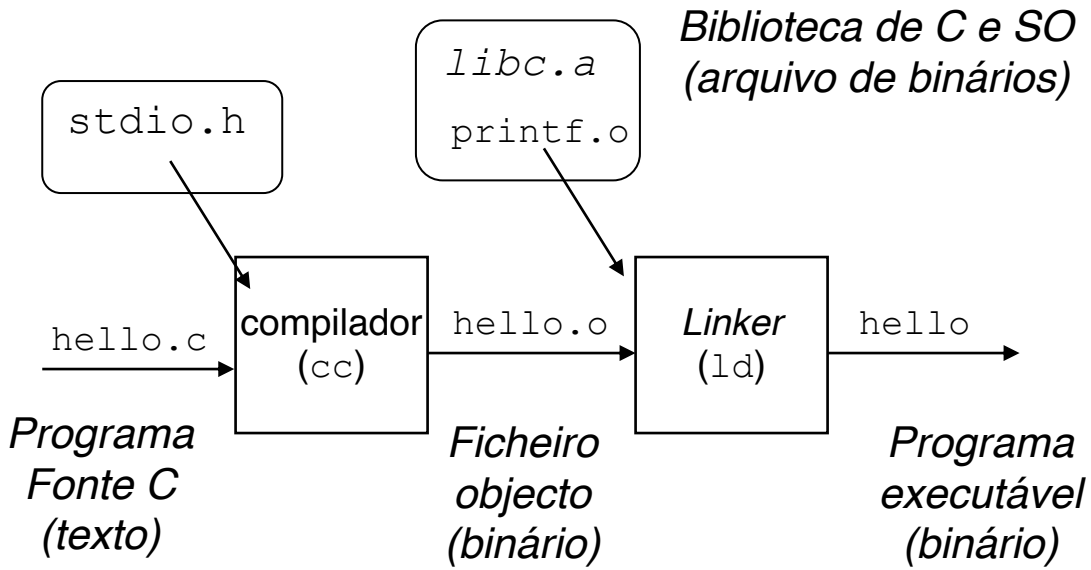
- Escrever 5 linhas com “hello world”:

```
#include <stdio.h>
#define NVEZES  5

int main( ) {
    // escreve NVEZES
    for (int i=0; i<NVEZES; i++)
        printf("hello world\n");
    return 0;
}
```

Processo de compilação de C

- Compilação do programa:

```
cc –o hello hello.c
```



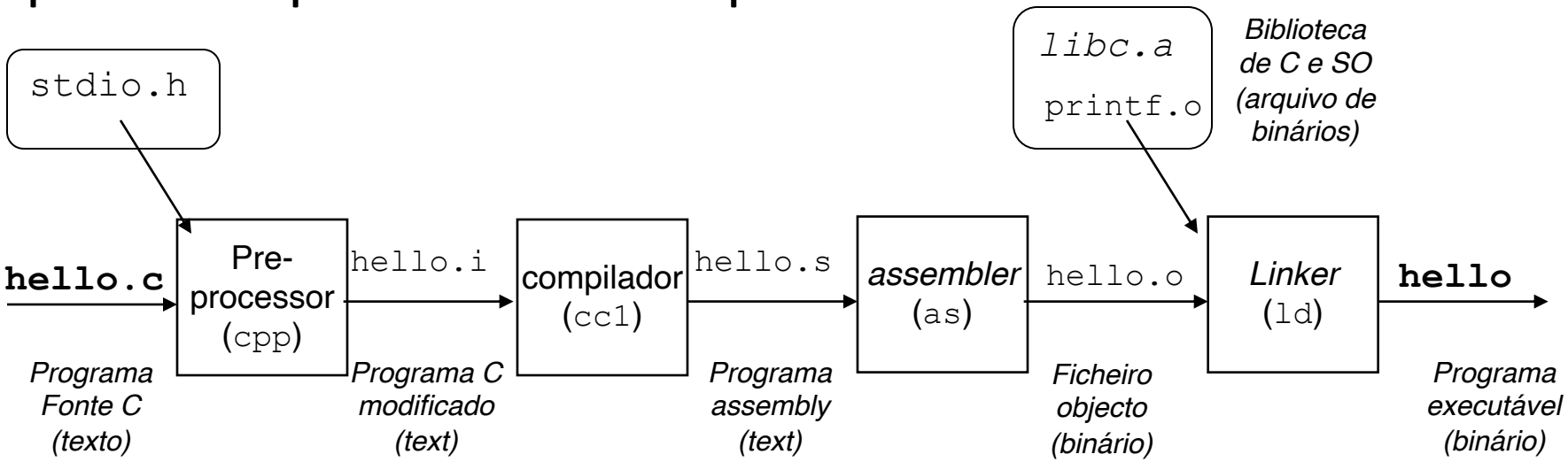
- Existem erros e warnings de compilação
- Existem erros ligação

Processo de compilação de C

- Compilação do programa:

```
cc –o hello hello.c
```

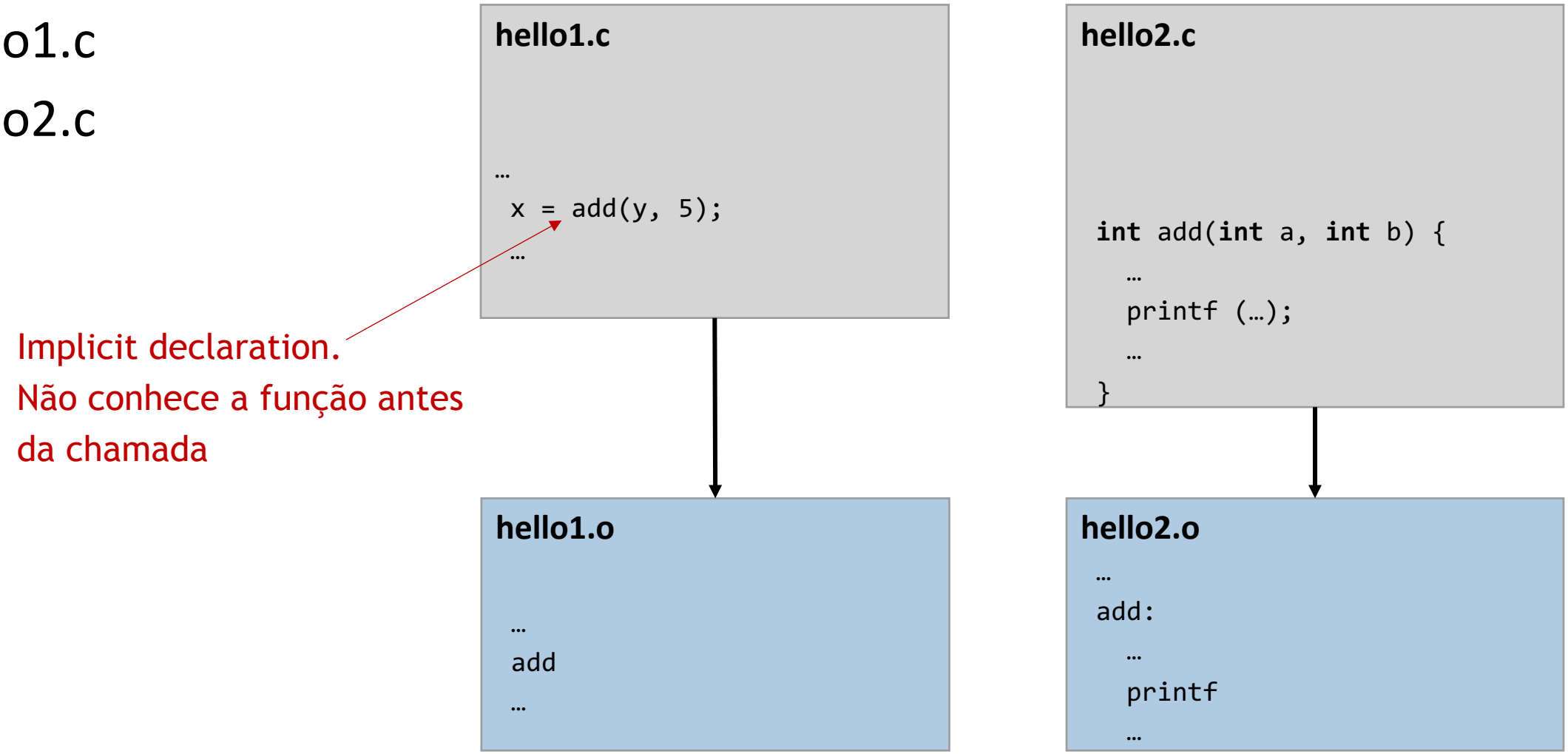
- **Nota**: os ficheiros de cabeçalho não entram na linha de compilação. São incluídos pelo pré-processador
- Os compiladores podem ter mais passos intermédios:



Passo de Compilação

cc1 hello1.c

cc1 hello2.c

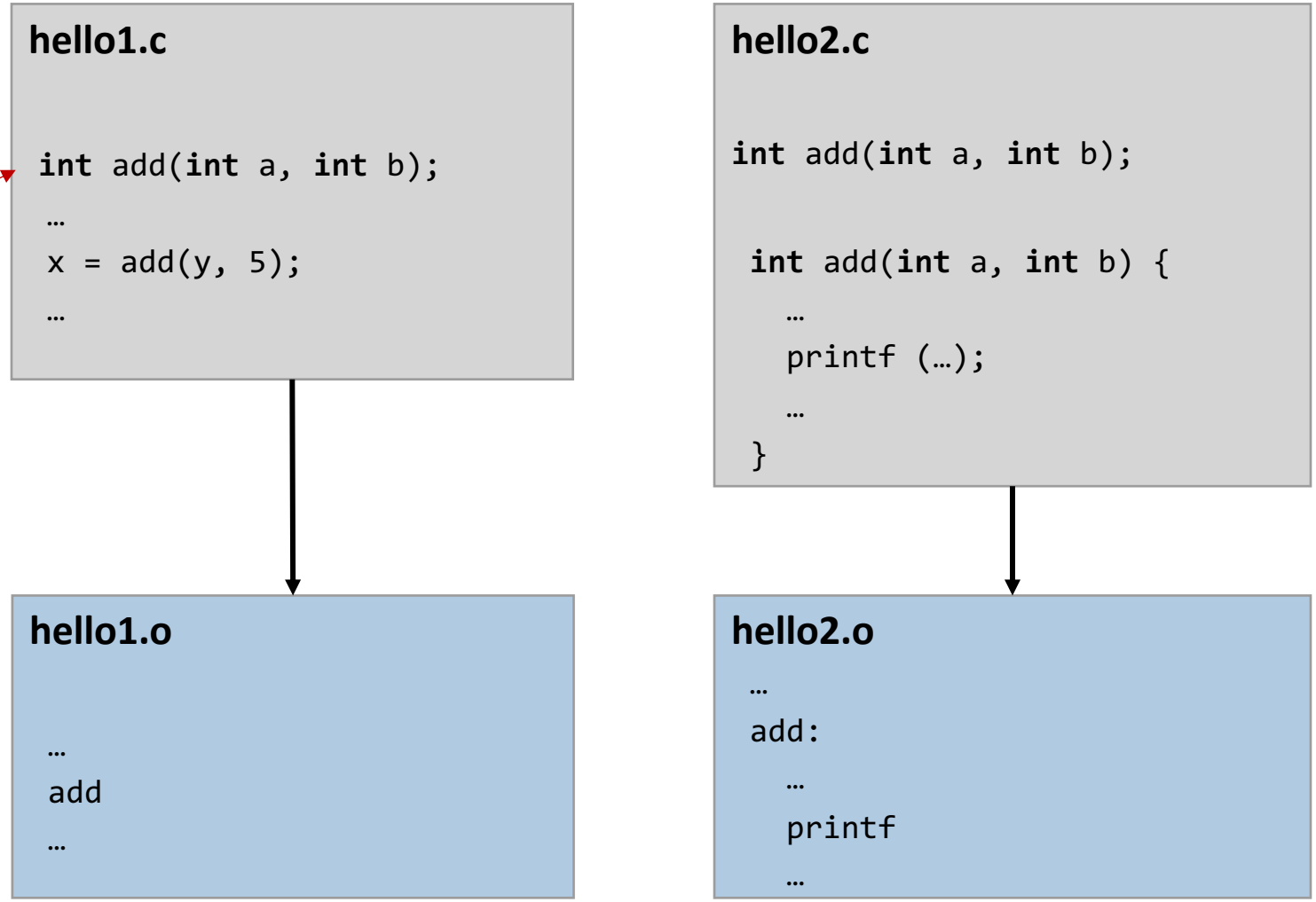


Passo de Compilação

cc1 hello1.c

cc1 hello2.c

Normalmente colocado num ficheiro cabeçalho, por exemplo hello.h, cujo *include* já foi processado pré-processador.



Passo de Compilação

cc1 hello1.c

cc1 hello2.c

hello1.c

```
int add(int a, int b);
…
x = add(y, “5”);
…
```

Erro de compilação

hello2.c

```
int add(int a, int b);

 int add(int a, int b) {
   …
   printf (…);
   …
 }
```

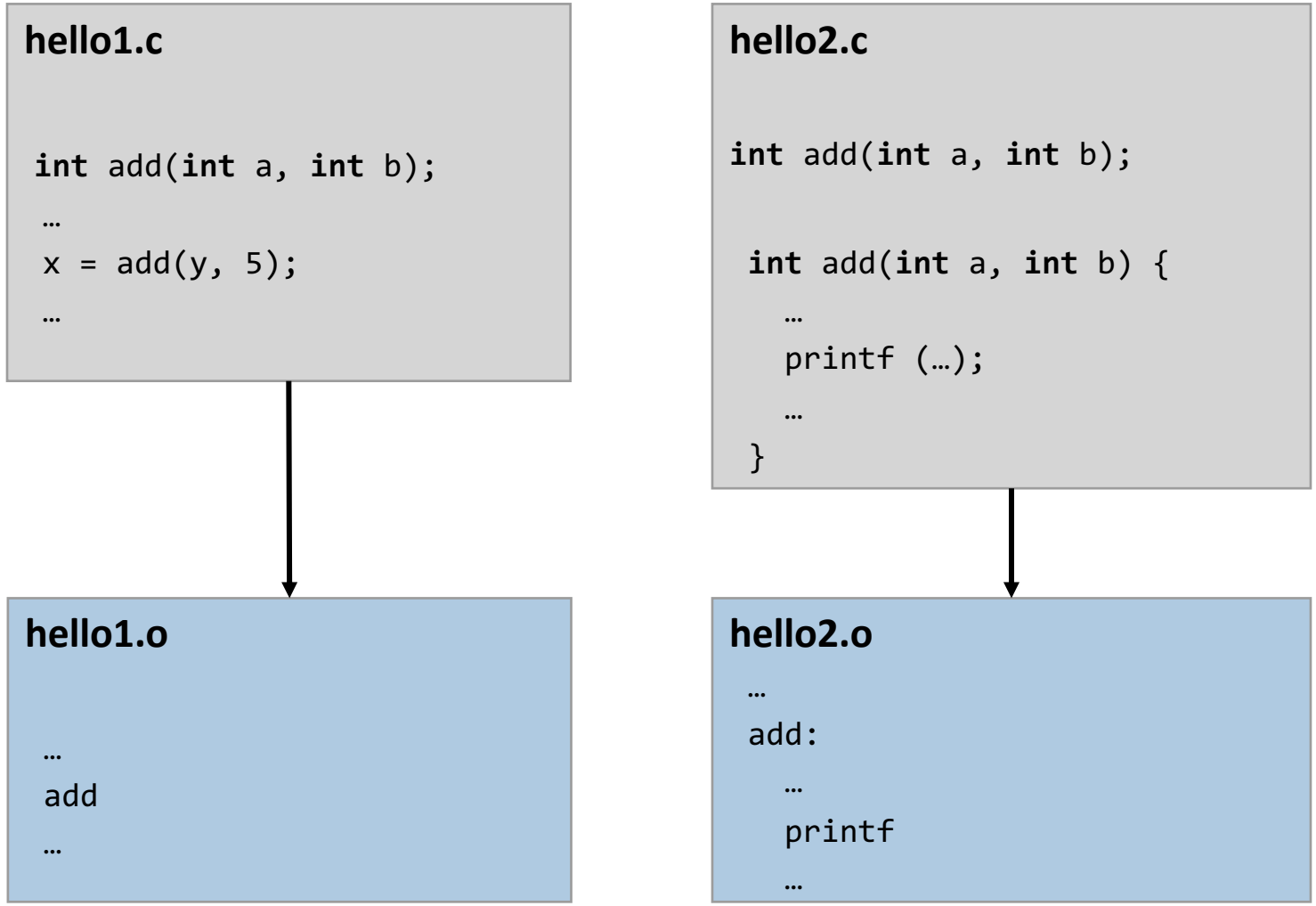
hello2.o

```
…
add:
   …
   printf
   …
```

Passo de Compilação

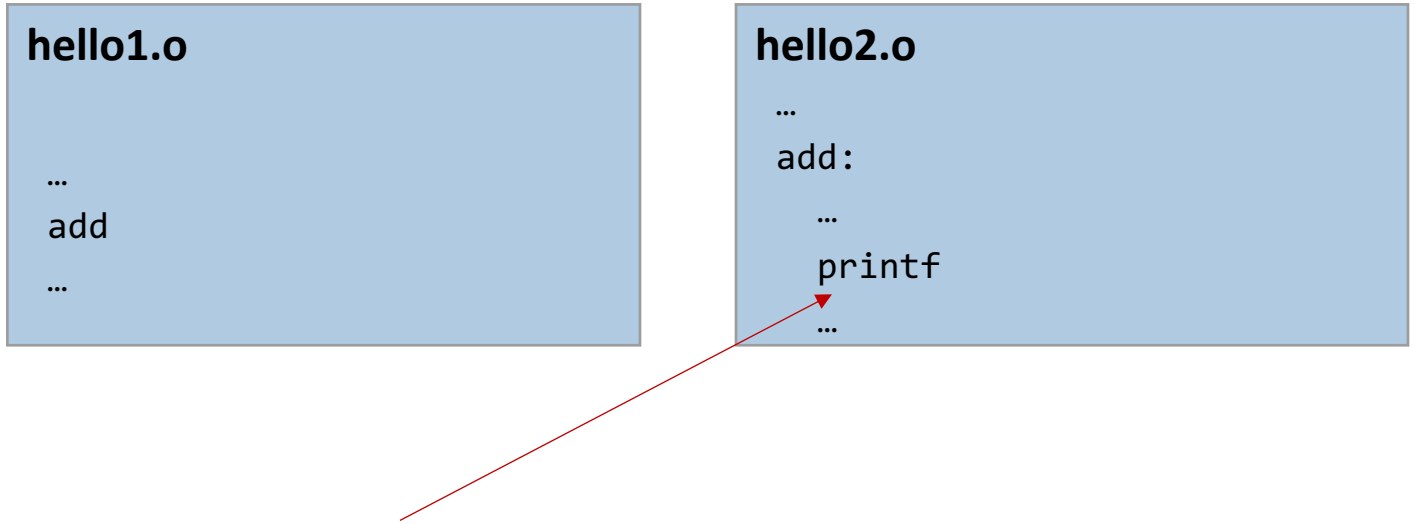
cc1 hello1.c

cc1 hello2.c



Passo de ligação

ld hello1.o hello2.o -o hello



Erro de ligação.

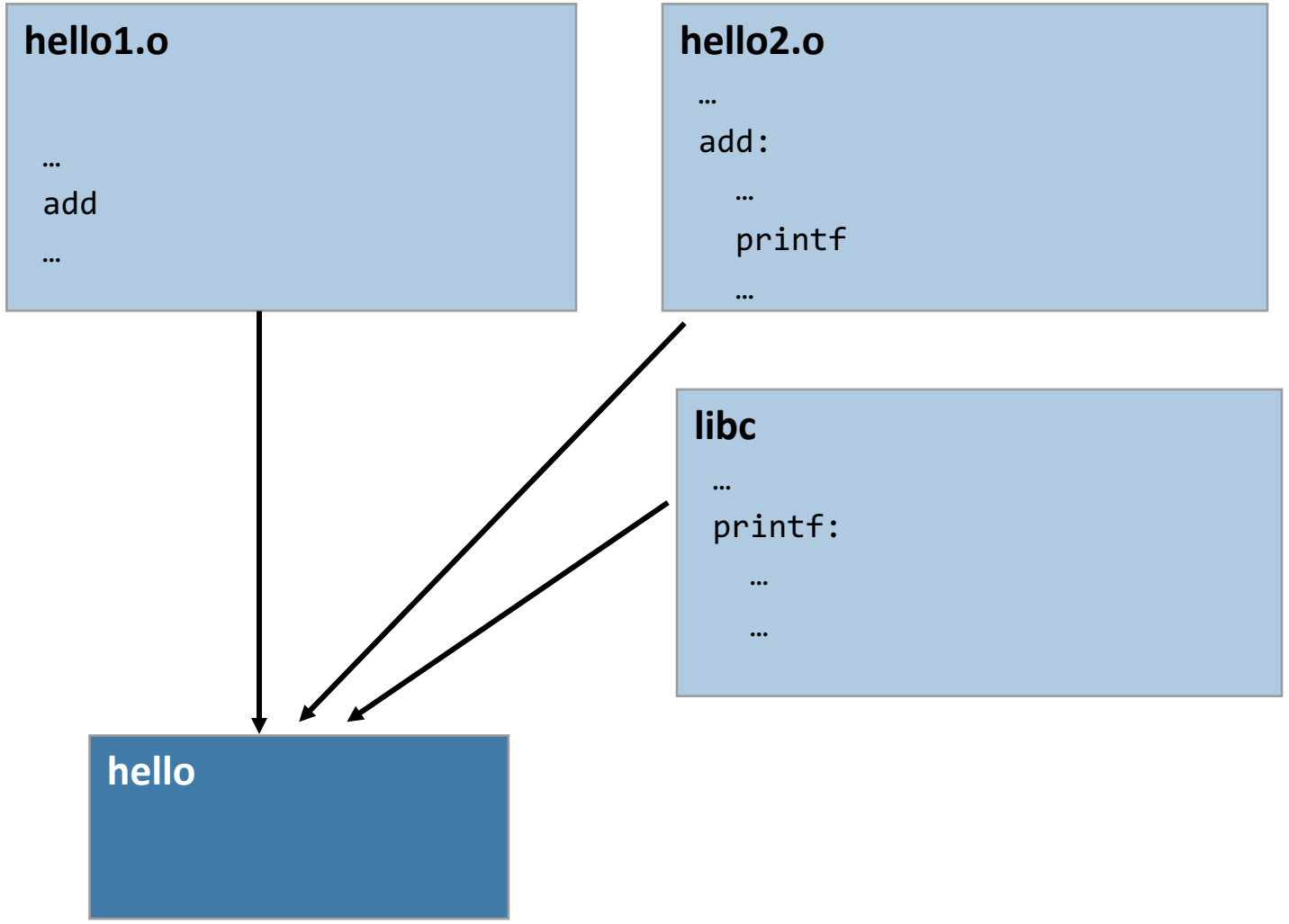
Onde está a implementação do printf?

Passo de ligação

ld hello1.o hello2.o

-lc -o hello

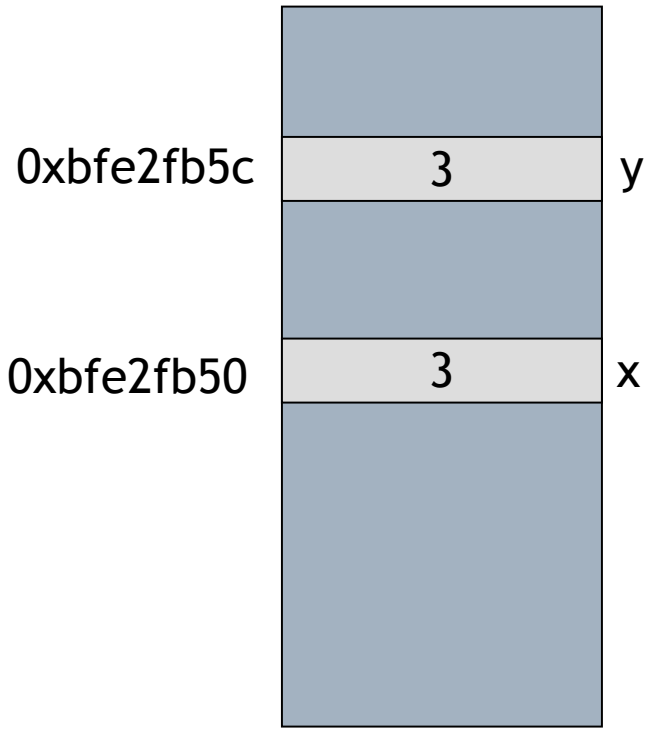
O cc, quando chama o ld, junta a opção -lc que liga a biblioteca lib**c**.



Parâmetros de funções (Java e C)

- Os argumentos são passados por cópia do valor (**passagem por valor**)

```
int fun1( int x ) {
    x=x+1;
    return x;
}
...
int y=3;
int z=fun1( y );
```



- Uma cópia do valor em y é passada para x de fun1
  - y não é alterado!

Exemplo de Programa

```
#include <stdio.h>

int proxpar( int x ) {
    if ( x%2 == 0 ) return x+2;
    else return x+1;
}

int main( ) {
    int par = proxpar( 3 );
    printf("Proximo par: %d\n", par);
    return 0;
}
```

Exemplo de Programa (versão 2)

```
#include <stdio.h>

int proxpar( int x ) {
    return ( x%2 == 0 ) ? x+2 : x+1;
}


int main( ) {
    int par = proxpar( 3 );
    printf("Proximo par: %d\n", par);
    return 0;
}
```

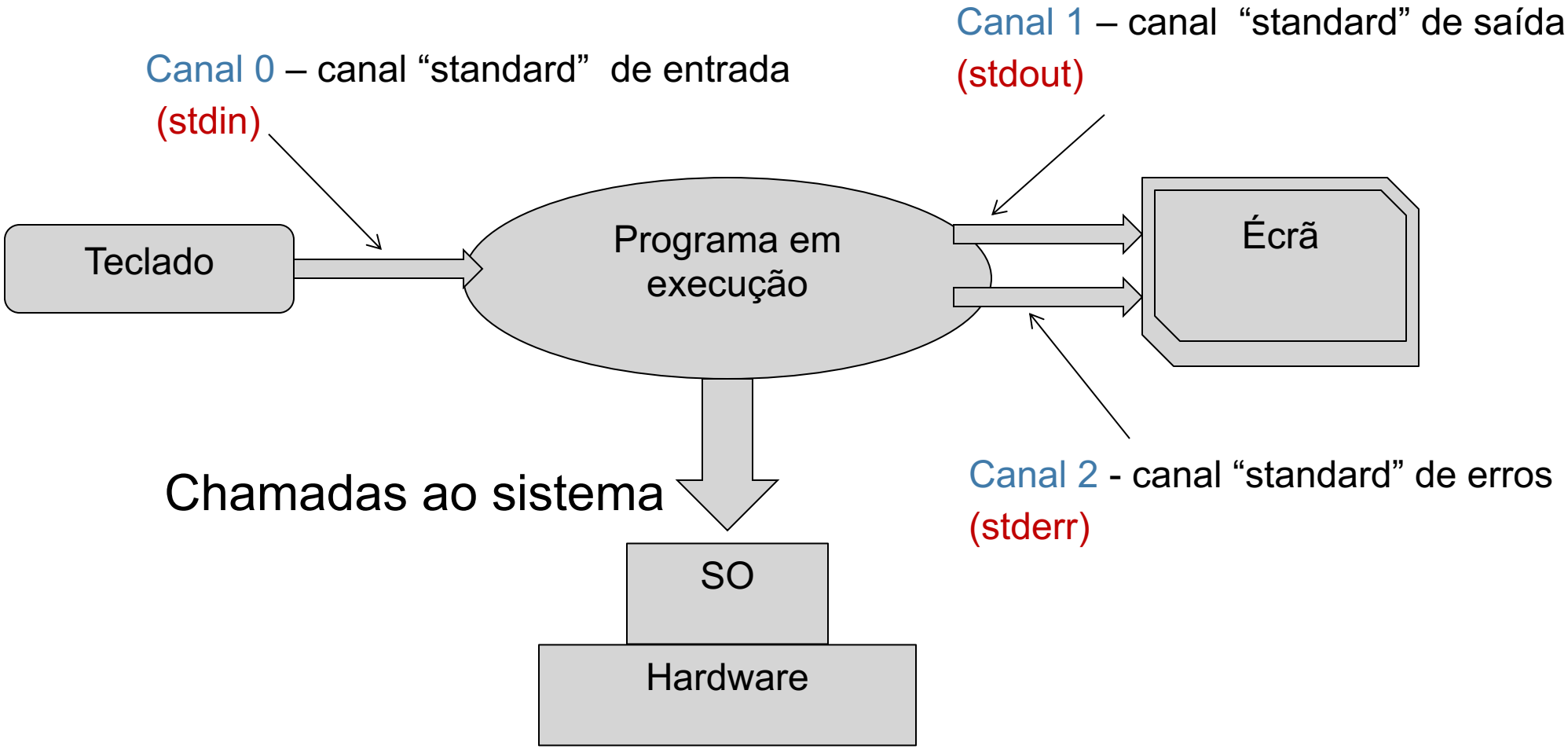
Em C, qualquer valor inteiro pode ser interpretado como um booleano:

- == 0 → **falso**
- != 0 → **verdadeiro**

Exemplo de Programa (versão 3)

```
#include <stdio.h>

int proxpar( int x ) {
    return x%2 ? x+1 : x+2;
}


int main( ) {
    int par = proxpar( 3 );
    printf("Proximo par: %d\n", par);
    return 0;
}
```

Programa em execução



Escrevendo dados (Java)

- Exemplo `System.out.print(X)` / `println(X)`
  - X é uma String ou então um novo objecto String é construído com uma representação do valor de X
  - String.valueOf(X)
  - Integer.toString(X), Long.toString(X), Float.toString(X), etc.

  - *System.out* é um Stream (canal) para onde é enviada a representação da String como uma sequência de bytes que codifica os caracteres passados ao SO para que este faça aparecer o texto no écran (ou noutro local)

Em C

- Exemplo `printf(`*`format, X,...`*`)`
  - *`format`* – **string** que define o que escrever e como apresentar os restantes argumentos

  Exemplo: `printf("result: %d, %x, %f\n", 12, 12, 12.0);`

  escreve: **`result: 12, c, 12.00000`**↵
  - Alguns formatos:

    %d ou %i → inteiro em decimal
    %x → inteiro em hexadecimal
    %c → caracter
    %f → float
    %z → size
  - Exemplos:

    `%ld` = long em base 10
    `%f` = float
    `%e` = double em notação científica
    `%hd` = short em base 10
    `%lu` = unsigned long base 10

Modificadores:

%h → half
%l → long
%ll → long long
%u → unsigned

- Em Java também existe **`printf()`**

Printf faz o indicado pela formatação

- Exemplo:

```
short i = -1;
unsigned short j = i;
int x = j+1;
printf("i=%hd, j=%hu, x=%d", i, j, x);
printf("i=%u j=%d x=%f ?=%d", i, j, x);
```

- O compilador pode avisar, mas produz um executável
- Resultados? (pode variar entre arquiteturas....)

```
i=-1, j=65535, x=65536
i=4294967295 j=65535 x=0.00000 ?=134518384
```

Ler dados do Standard Input em C

- Java → Podemos usar a classe Scanner
- Em C:
  - Ler texto do *standard input* convertendo para o tipo indicado:

```
int scanf( char *format, …vars…)
```

    - Devolve nº de variáveis que leu/converteu

  - Exemplo:

```
int v1, v2;
int n = scanf("%d%d",&v1,&v2);
if (n!=2) ... // houve erro
```

De texto para números em C

- De *string* para int?
- Exemplo do "ascii to int":
  - Em Java → `Integer.parseInt(S), Long.parseLong(S), ...`
  - Em C:

```
int atoi( char* str );
```

    - Obtém o número representado numa string:

```
int i = atoi("135");
```

    - Existem outras funções mais gerais, como `strtol` ou `sscanf`
- Exemplo:

```
float num;
sscanf("135.5", "%f", &num );
```

Apontadores

- *Pointer* = apontador = ponteiro = referência

- Declaração de variável apontador:

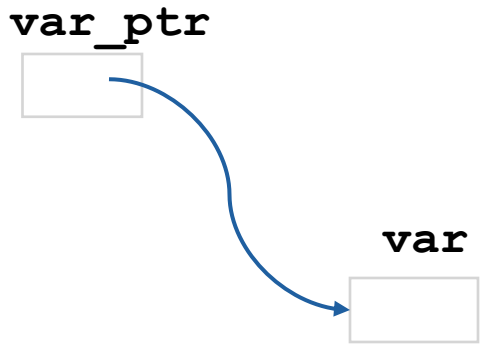
  Tipo_apontado* var_ptr;

- Operações:
  - Obter apontador para uma variável:

  var_ptr = &var

  - Obter valor apontado (valor na variável original):

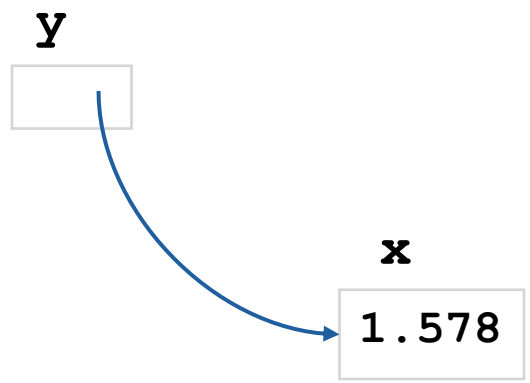
  x = *var_ptr



Apontadores

- Exemplo:

```
double x = 1.578;
double *y;

y = &x; // y aponta x
*y = 5.4;
printf("valor em x=%f\n", x );
x = x+1.5;
printf("valor apontado por y=%f\n", *y );
```

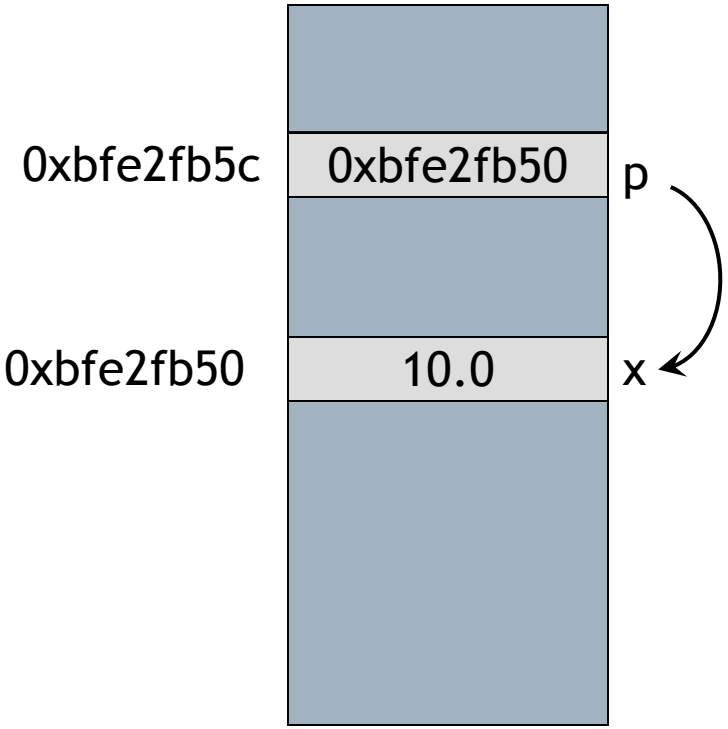



```
valor apontado por y=6.90000
```

Apontadores: Exercício

float x = 10.0;

float* p = &x;



- Formato %p permite imprimir o endereço em hexadecimal de uma variável
  - printf("Valor de x: %f", x); → 10.0
  - printf("Endereco de x: %p", &x); → 0xbfe2fb50
  - printf("Valor de p: %p", p); → 0xbfe2fb50
  - printf("Valor apontado por p: %f", *p); → 10.0
  - printf("Endereco de p: %p", &p); → 0xbfe2fb5c

Referências inválidas

- Um apontador C não inicializado contém uma referência inválida (lixo)
- Java e C têm a referência para “nada”
  - Java: constante **null**
  - C: constante **NULL**
- Estas não podem ser acedidas.
  - Java: não podemos aceder a um objecto que não foi criado (variável de instância a **null**)
  - C: não podemos aceder ao apontado por **NULL**

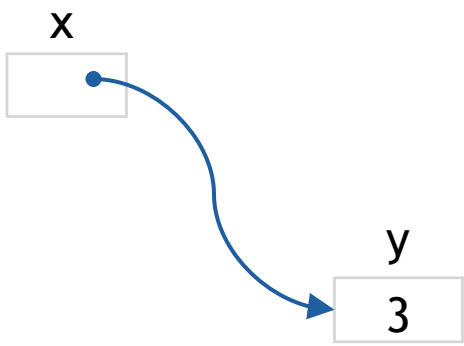
```
String s;
int z = s.length();
// null pointer exception!!
```

```
int *v=NULL;
printf( "%d", *v );
// segmentation fault!!
```

Parâmetros apontadores em C

- Em C podemos passar referências (apontadores):

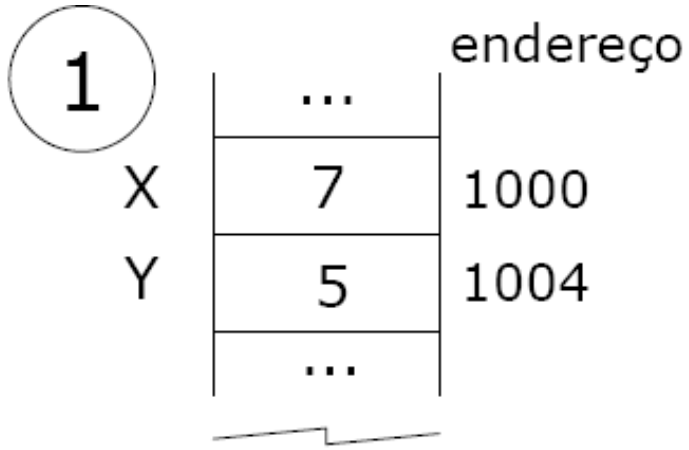
```
     int fun1( int *x ) {
          *x=*x+1;
          return *x;
     }
     ...
int y=3;
int z=fun1( &y );
                    //passamos a referência de y
```



- O endereço de y é passado para x de fun1
- y é alterado via apontador x!

Passagem da referência (por valor)

```
int main() {
  int x = 7, y = 5;
  printf("Valor de X: %d\n", x);
  printf("Valor de Y: %d\n", y);
  troca(&x, &y);
  printf("Valor de X: %d\n", x);
  printf("Valor de Y: %d\n", y);
  return 0;
}

void troca(int *px, int *py) {
  int temp = *px;
  *px = *py;
  *py = temp;
}
```

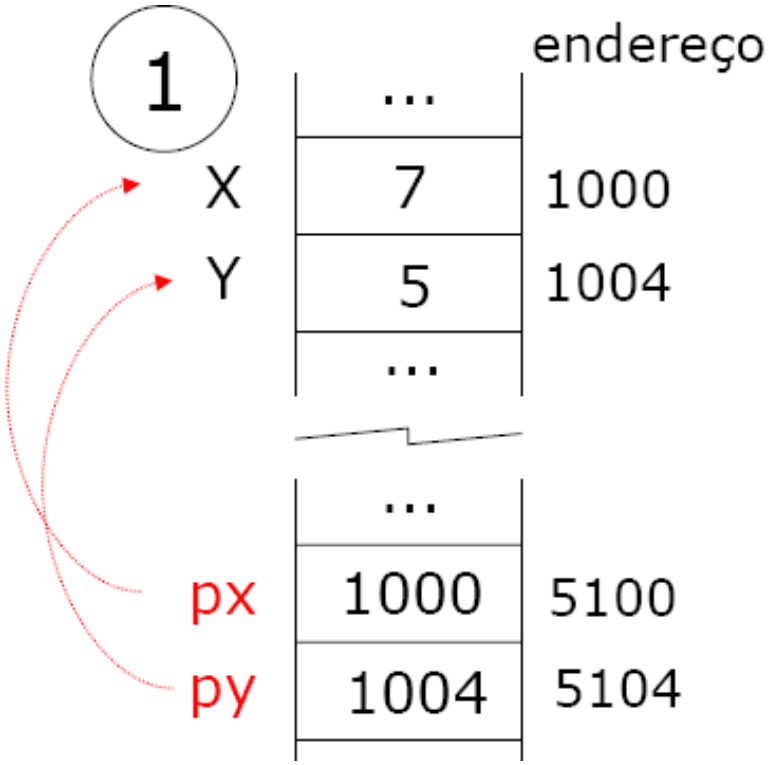


Passagem da referência por valor.

Não confundir com passagem de argumento por referência (que é outra coisa – VER C++)

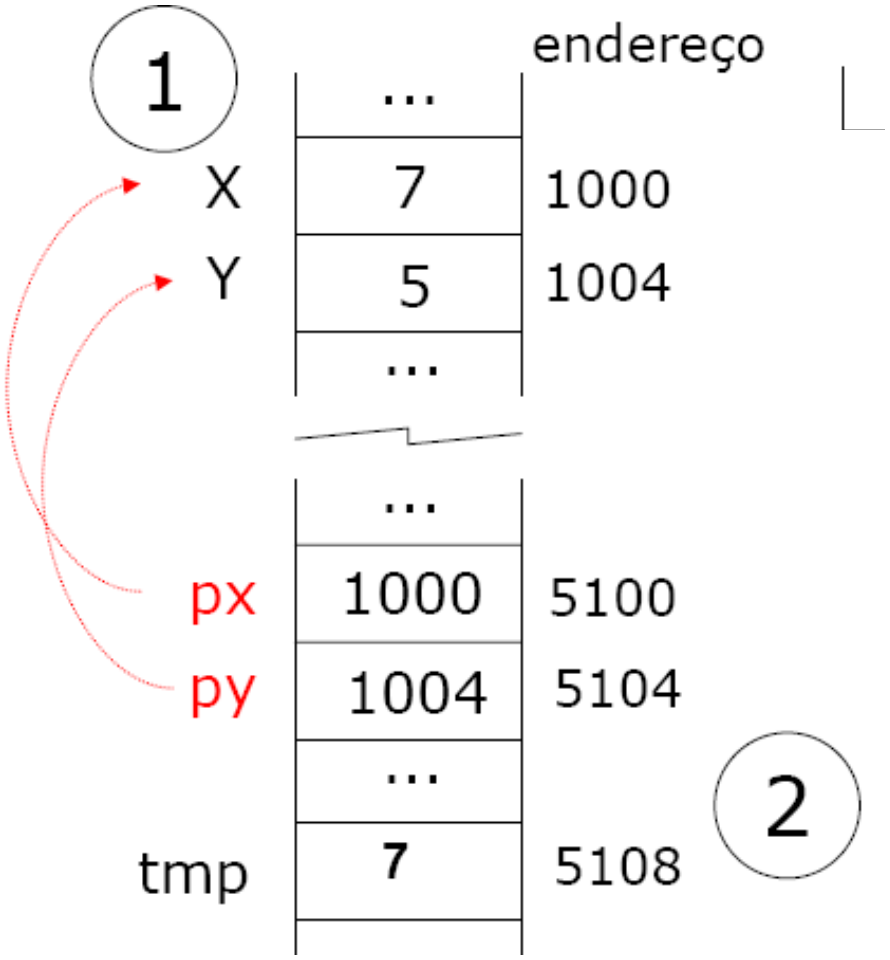
Passagem da referência (por valor)

```
int main() {
  int x = 7, y = 5;
  printf("Valor de X: %d\n", x);
  printf("Valor de Y: %d\n", y);
  troca(&x, &y);
  printf("Valor de X: %d\n", x);
  printf("Valor de Y: %d\n", y);
  return 0;
}

void troca(int *px, int *py) {
  int temp = *px;
  *px = *py;
  *py = temp;
}
```



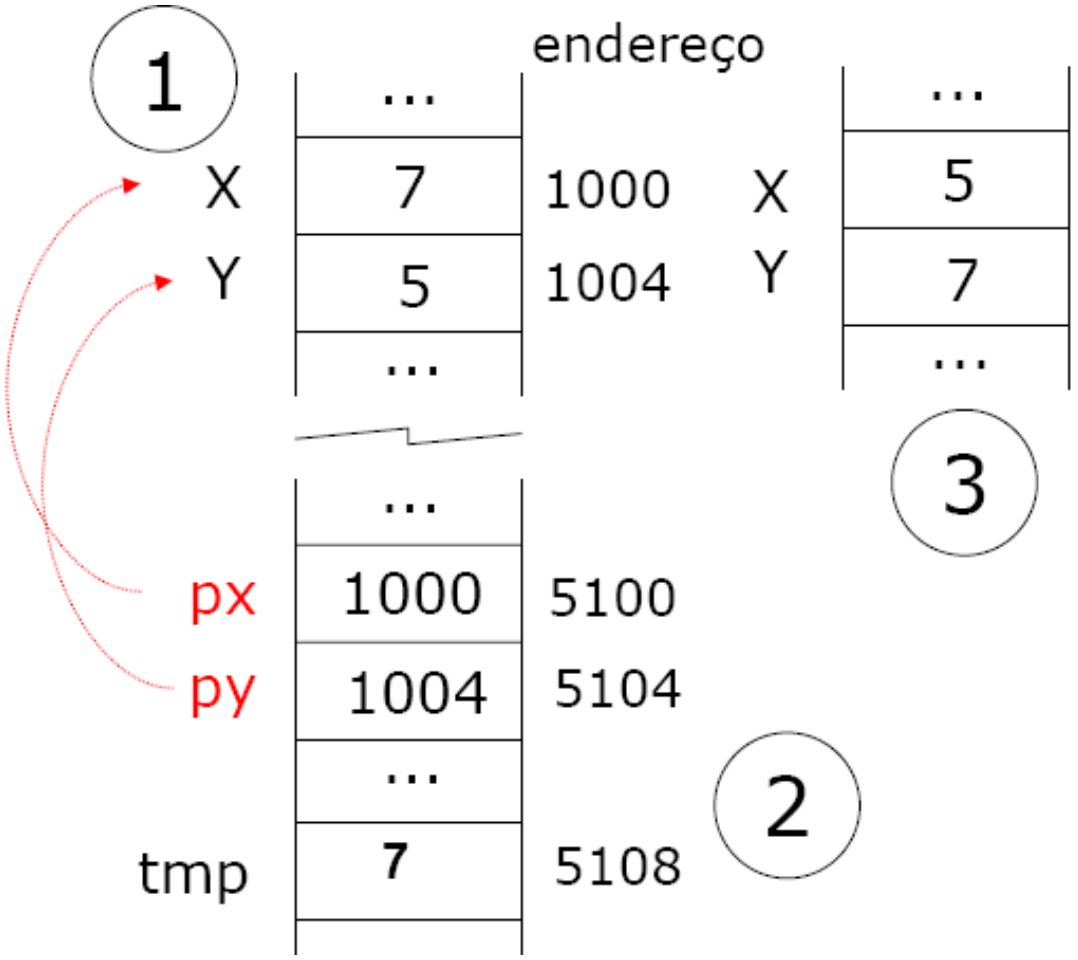
Passagem da referência (por valor)

```
int main() {
  int x = 7, y = 5;
  printf("Valor de X: %d\n", x);
  printf("Valor de Y: %d\n", y);
  troca(&x, &y);
  printf("Valor de X: %d\n", x);
  printf("Valor de Y: %d\n", y);
  return 0;
}

void troca(int *px, int *py) {
  int temp = *px;
  *px = *py;
  *py = temp;
}
```



Passagem da referência (por valor)

```
int main() {
  int x = 7, y = 5;
  printf("Valor de X: %d\n", x);
  printf("Valor de Y: %d\n", y);
  troca(&x, &y);
  printf("Valor de X: %d\n", x);
  printf("Valor de Y: %d\n", y);
  return 0;
}

void troca(int *px, int *py) {
  int temp = *px;
  *px = *py;
  *py = temp;
}
```



Objetos como parâmetros em Java

- O mesmo se passa com os objetos em:

```
class A { public int i; }
int fun1( int x, A z ) {
    x=x+1;
    z.i = z.i+1;
    return x;
}
```

```
...
int y1 = 1;
A y2 = new A(); // em y2 fica o endereço
onde está o objeto criado na memória
y2.i=1;
int w = fun1(y1, y2);
```

  - A passagem de argumentos é por cópia
    - uma cópia do valor em y1 é passada para x de fun1
    - uma cópia do valor em y2 (referência para o objeto) é passada para z de fun1
  - y1 não é alterado por fun1!
  - y2.i é alterado por fun1!

Arrays em C - vetores

- Os arrays consistem em sequências contíguas de elementos do mesmo tipo.
- Sintaxe (variantes):

```
int v[3];   // criação, não iniciado
v[0]=1; v[1]=2; v[2]=3;

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

- Tabela de símbolos do compilador:

  *'v' - **constante** que representa o início do array (seu endereço)*

  v = &v[0]

Arrays em C e apontadores

- O nome do *array* vale por um apontador constante:

```
int v[3];     // v tem tipo int*

v[0] = 1;   ⇔   *v = 1     ⇔ *(v+0)=1

v[2] = 5;    ⇔   *(v+2) = 5;

int w[3];
v = w;     → Erro de compilação: v é uma constante
```

Arrays em C e funções

- Um parâmetro que é um *array* de dimensão pré-definida:

```
void fun( int vet[3] ){ ... }
```

Uso:

```
int myvet[3];
fun( myvet );
```

← O que é passado é o endereço!

- Podemos ter parâmetros *arrays* sem dimensão pré-definida:

```
void fun( int vet[] );
```

ou

```
void fun( int *vet );
```

Arrays em C e funções

- Se um argumento é um array, o valor que é passado é o do endereço!

```
int myvet[3];
fun( myvet );
```

independentemente da declaração de fun:

`void fun(int v[3])` *vs.* `void fun(int *v)`

- Porquê?
  - Consequências se passasse todo o array? Memória ocupada? Tempo de execução?

Parâmetros *arrays* e sua dimensão

- Um array não é um objeto → não tem campo *length*!

```
void fun( int* vet );
```

  - `vet` será um array de int, ou aponta só um int?
  - Fun deve ser chamada com um array? De que dimensão?

- Uma solução: acompanhar o array com a dimensão:

```
void fun(int vsize, int vet[vsize]);
```

ou

```
void fun(int vsize, int* vet);
```

Java ... Referencias de objetos

- Recorde:

```
class A { public int i; ......}
A a = new A();
A b, c;
a.i = 1;  b = a;  b.i++;
c = new A(); c.i = 2;
```

Valores em a, b e c?

`a == b` *vs.* **`b == c`**

C ... Comparando apontadores

```
int x = 1;
int *xp = &x;
(*xp)++;
int y = 2;
int *yp = &y;
```

Comparando:

`x == y` *vs.* **`xp == yp`**

Java ... Arrays

- Recorde:

```
int a[] = {1, 2, 3}; // ou int[] a ...
int b[] = a;
int c[] = {1, 2, 6};
b[2]=6;
```

Valores em a, b e c ?

`a == b` *vs.* **`b == c`**

vs. **`Arrays.equals(b, c)`**

C ... Arrays e apontadores: Comparação

```
int a[] = {1, 2, 3};
int *b = a;
int c[] = {1, 2, 6};
b[2]=6;
```

Comparações:

`a == b` *vs.* **`b == c`**

vs. **`memcmp(b, c, 3*sizeof(int))`**

vs. **`memcmp(a, c, 3*sizeof(int))`**

Exemplo: comprimento de uma string

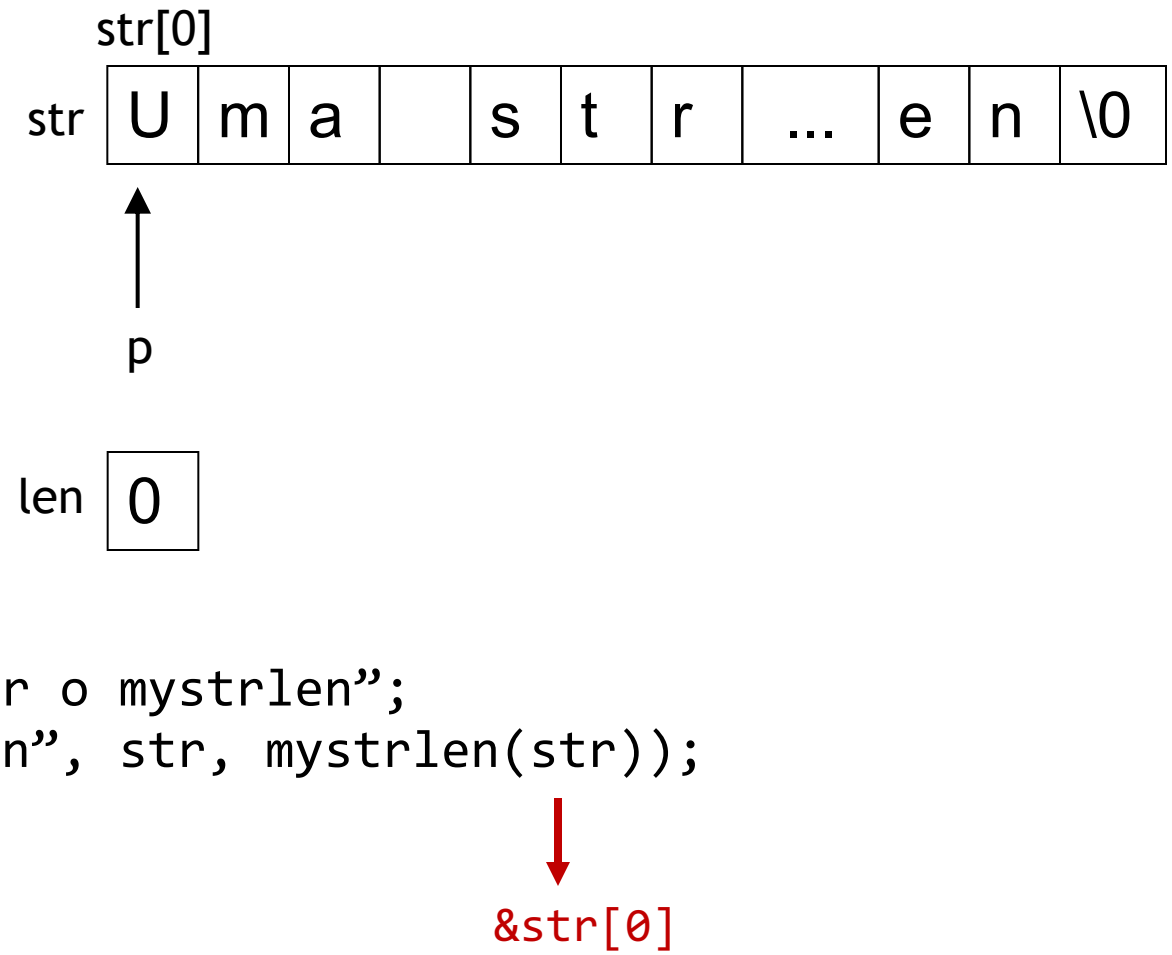
```
#include <stdio.h>

int mystrlen(char *p) {
  int len = 0;
  while (*p != '\0') {
        p++;
        len++;
  }
  return len;
}

int main() {
  char  str[] = "Uma string para testar o mystrlen";
  printf("O comprimento de \"%s\": %d\n", str, mystrlen(&str[0]));
  return 0;
}
```

Há formas mais compactas de implementar esta função.

Vamos utilizar esta versão que é mais fácil de ler e seguir.

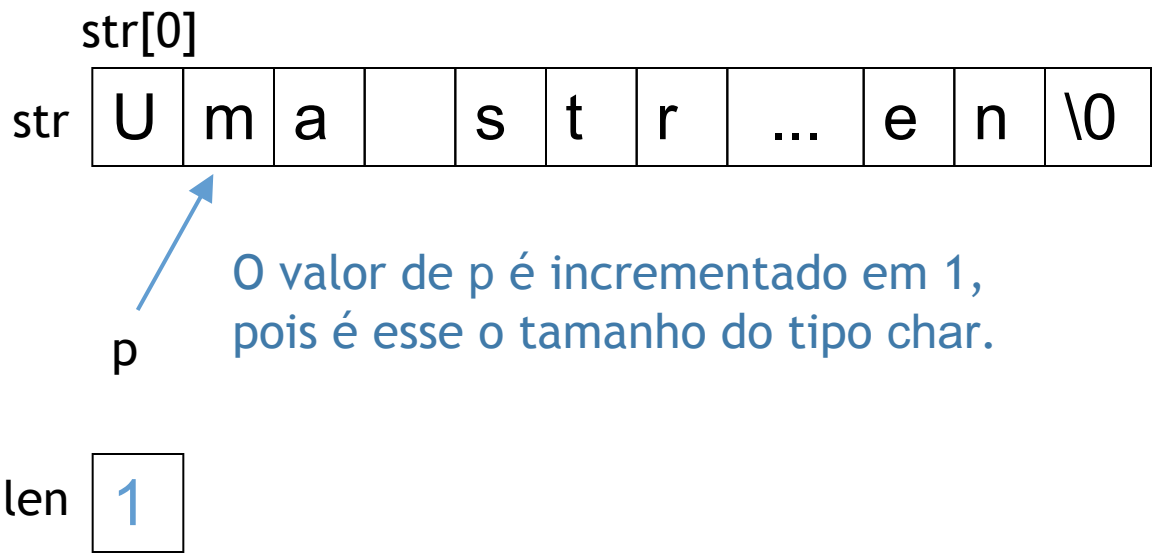
Exemplo: comprimento de uma string

```
#include <stdio.h>

int mystrlen(char *p) {
  int len = 0;
  while (*p != '\0') {
        p++;
        len++;
  }
  return len;
}
int main() {
  char  str[] = "Uma string para testar o mystrlen";
  printf("O comprimento de \"%s\": %d\n", str, mystrlen(str));
  return 0;
}
```



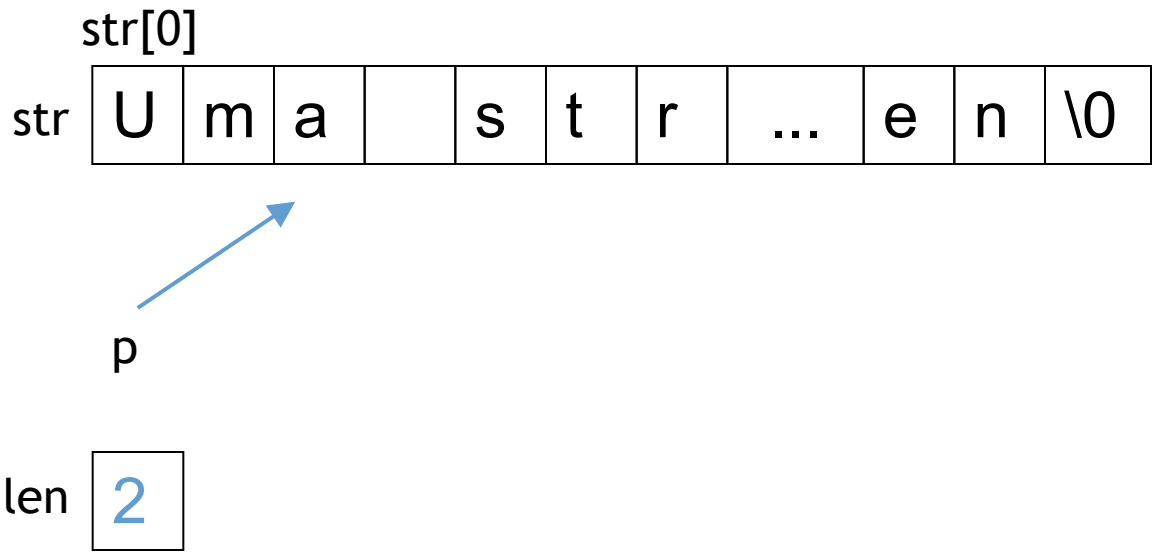
Exemplo: comprimento de uma string

```
#include <stdio.h>

int mystrlen(char *p) {
  int len = 0;
  while (*p != '\0') {
        p++;
        len++;
  }
  return len;
}
int main() {
  char  str[] = "Uma string para testar o mystrlen";
  printf("O comprimento de \"%s\": %d\n", str, mystrlen(str));
  return 0;
}
```



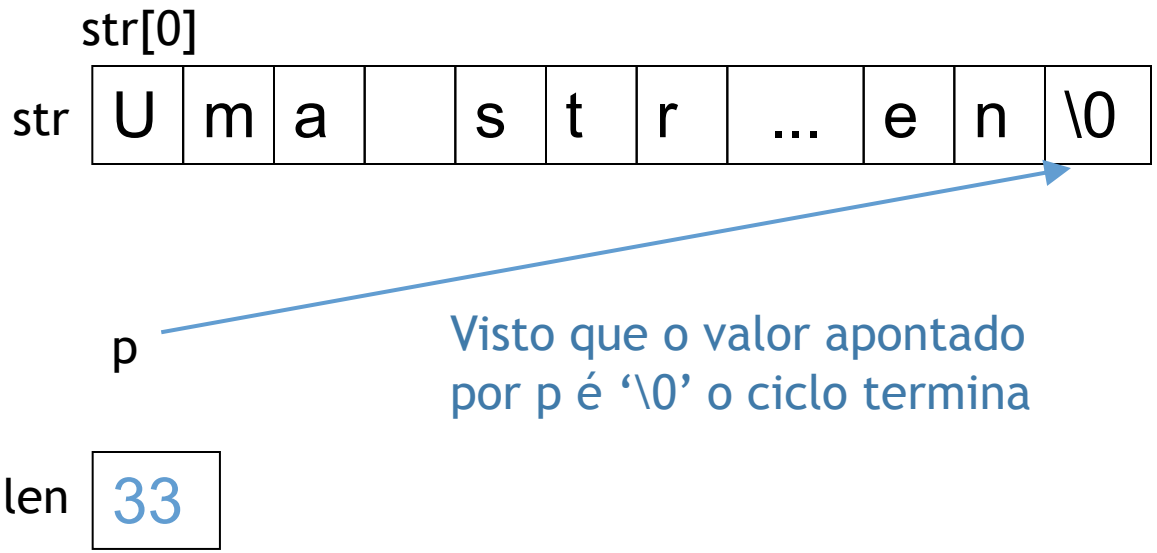
Exemplo: comprimento de uma string

```
#include <stdio.h>

int mystrlen(char *p) {
  int len = 0;
  while (*p != '\0') {
        p++;
        len++;
  }
  return len;
}
int main() {
  char  str[] = "Uma string para testar o mystrlen";
  printf("O comprimento de \"%s\": %d\n", str, mystrlen(str));
  return 0;
}
```



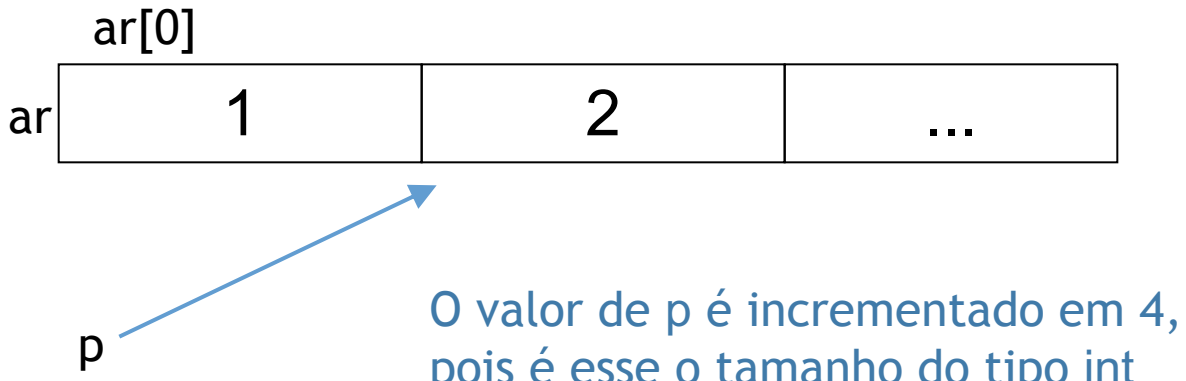
Exemplo: comprimento de uma string

```
#include <stdio.h>

int mystrlen(char *p) {
  int len = 0;
  while (*p != '\0') {
        p++;
        len++;
  }
  return len;
}
int main() {
  char  str[] = "Uma string para testar o mystrlen";
  printf("O comprimento de \"%s\": %d\n", str, mystrlen(str));
  return 0;
}
```



Exemplo: somar elementos de um array

```
#include <stdio.h>

int sumarray(int *p, int n) {
  int i, sum = 0;
  for (i = 0; i <n; i++) {
       sum += *p;
       p++;
  }
  return sum;
}

int main() {
  int  ar[] = {1, 2, 3, 4, 5, 6, 7};
  printf("Soma = %d\n", sumarray(ar, 7));
  return 0;
}
```



O valor de p é incrementado em 4, pois é esse o tamanho do tipo int

Criar variáveis dinamicamente

- Criando dinamicamente variáveis:

```
void * malloc( int size )
```

- Criar um vetor de 3 inteiros:

```
int *v = (int*)malloc(3*sizeof(int));
```

- Não há “garbage collection”. Têm de ser libertadas (destruídas) com:

```
void free( void *ptr )
```

- Exemplo:

```
free( v );
```

Criar variáveis dinamicamente

- Usando um array criado dinamicamente:

```
int *v = (int*) malloc(3*sizeof(int));

v[2] = 5;
v[1]++;
...etc...

free(v);
```

Exemplo

```
int v1[MAX] = { ..........};
```

- Criar uma copia só com os elementos >=0:

```
int npos = 0;
for (int i=0; i<MAX; i++ )
    if ( v1[i]>=0 ) npos++;

int *vpos = (int*) malloc(npos*sizeof(int));
for (int i=0, j=0; i<MAX; i++ )
    if ( v1[i]>=0 ) vpos[j++] = v1[i];
```

Strings em C

- Não existe tipo específico para *strings*.
- Convenção: uma sequência de *chars* terminada com '\0' (código ASCII 0)

```
char msg[]={'H','e','l','l','o','\0'};
char msg[]= "Hello";
```

- Mas há constantes! Exemplo: **"Hello"**

```
char *msg2 = "Hello";
```

  - msg2 é um apontador para a string **constante** "Hello"

Strings em C – string.h

- A biblioteca do C dispõe de várias funções para manipular *strings*, declaradas em `string.h`
- Exemplos:

```
int strnlen(char *str, int max);
char *strncpy(char *dst, char *src, int len);
char *strncat(char *dst, char *src, int len);
int strncmp(char *s1, char *s2, int len);
```

- E versões sem dimensão (sem `max` ou `len`):

  `strlen`, `strcpy`, `strcat`, `strcmp`

Exemplos de utilização

```
char* str1;
char str2[] = "Hello";
char str3[100];

str1 = str2;
str1[strlen(str1)-1] = ‘u’;

strcat( strcpy(str3,str1),str2 );

str1 == str2    vs.    strcmp(str1, str2)
```

Java … String

- Recorde:

```
String s1 = “Hello”;
String s2;

s2 = s1;    vs.    s2 = new String(s1);

s2 == s1    vs.    s2.equals( s1 )
```

Nota: as String são imutáveis (não podem ser alteradas)

C ... Cadeias de caracteres (strings)

```
char* s1 = "Hello";
char* s2;

s2 = s1;    vs.     s2 = strdup(s1);

        s2[1] = 'E';
        // o que altero? É sempre válido?
```

Nota: as constantes strings são imutáveis; mas os arrays de char podem ser alterados

C ... Cadeias de caracteres (strings)

```
char *s1 = "Hello";
char *s2;

s2 = s1;     vs.  s2 = strdup(s1);

s2 == s1     vs.  strcmp(s1, s2)
```

É diferente comparar os apontadores ou comparar as strings apontadas

Erro comum usando apontadores

- Usar apontadores não iniciados:

```
char *str;
strcpy( str, "Hello" );
```

- Devia ser, por exemplo:

```
char str[100];
strcpy( str, "Hello" );
```

ou

```
char *str = strdup("Hello");
```

- em caso de dúvida, inicie sempre os apontadores com NULL ...

```
char *str = NULL;
```

Mais erros comuns

- Copiar apontadores em vez dos *arrays*:

```
char *str1; char str2[10];

str2 = "hello";     str1 = str2;
```

- Comparar apontadores em vez dos *arrays*:

```
str1 == str2
```

em vez de `strcmp( str1, str2 )`

nota: strcmp devolve 0 se iguais

valor>0 se str1>str2 e valor<0 se str1<str2

Mais erros usando *strings* em C

- Esquecer a terminação '\0':

```
char str[5]="Hello";
   // sizeof("Hello") é 6!  →
       {'H','e','l','l','o','\0'}
```

- Aceder fora do *array*:

```
char str[6];
strcat( strcpy(str, "Hello"), "World");
```

- Julgar que o tamanho do *array* é o da *string* ou *vice versa*:

```
char str2[10];
strcpy(str2, "Hello");
strlen(str2)
```

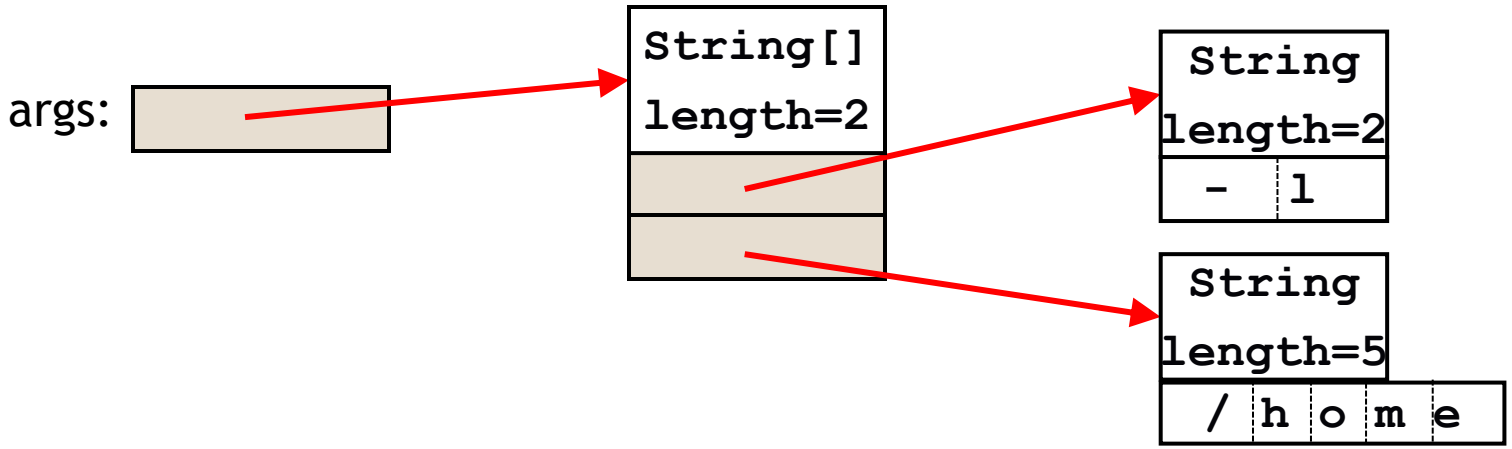
não é 10. É 5!

Exemplo: `main` em Java

`public static void main(String[] args)`

- Args contém uma cópia dos argumentos na linha de comando.
- Args representa um vetor de `String`. Exemplo:

```
$ java Ls  -l  /home
```



Escrever todos os argumentos:

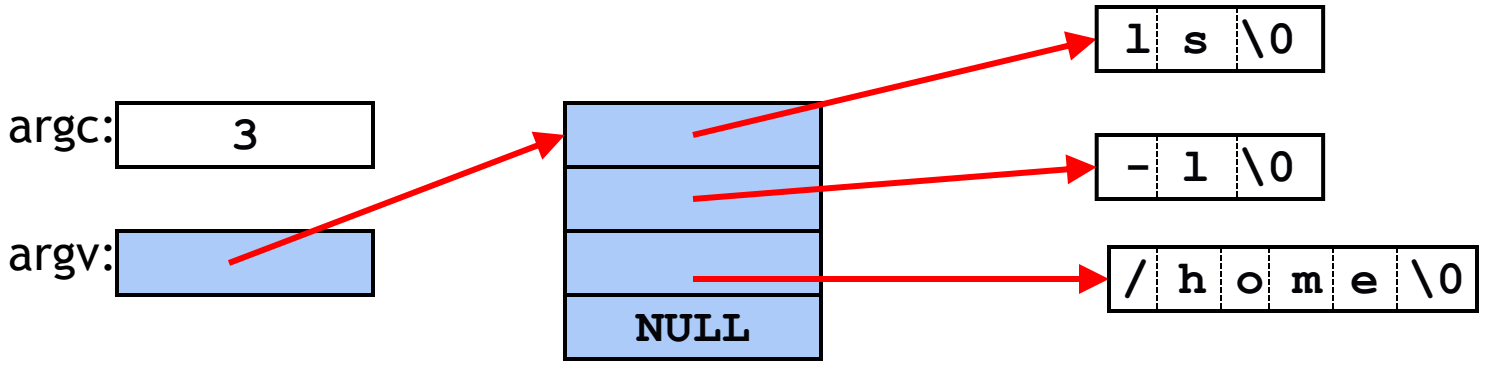
```
class Prog {

 public static
 void main( String[] args ) {
   System.out.print("num args: ");
   System.out.println( args.length );
   for ( int i=0; i<args.length; i++ ) {
     System.out.printf("%d: %s\n", i, args[i] );
   }
 }
}
```

main completo em C

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

- Argv contém uma cópia da linha de comando.
- Argv é um vetor de strings. Argc o número de strings. Exemplo dando o comando indicado:

```
$ ls  -l  /home
```



Escrever todos os argumentos:

```c
#include <stdio.h>

int main( int argc, char *argv[] ) {
    printf("num args: %d\n", argc );
    for ( int i=0; i<argc; i++ ) {
        printf("%d: %s\n", i, argv[i] );
    }
    return 0;
}
```

O que fica por falar (por agora)

- Dados estruturados: **struct**, **union**, **enum**
- Apontadores para funções
- Nomeação de tipos: **typedef**
- e muitas outras coisas…

- Pre-processador:
  - Macros com #define
  - Compilação condicional: #ifdef, …