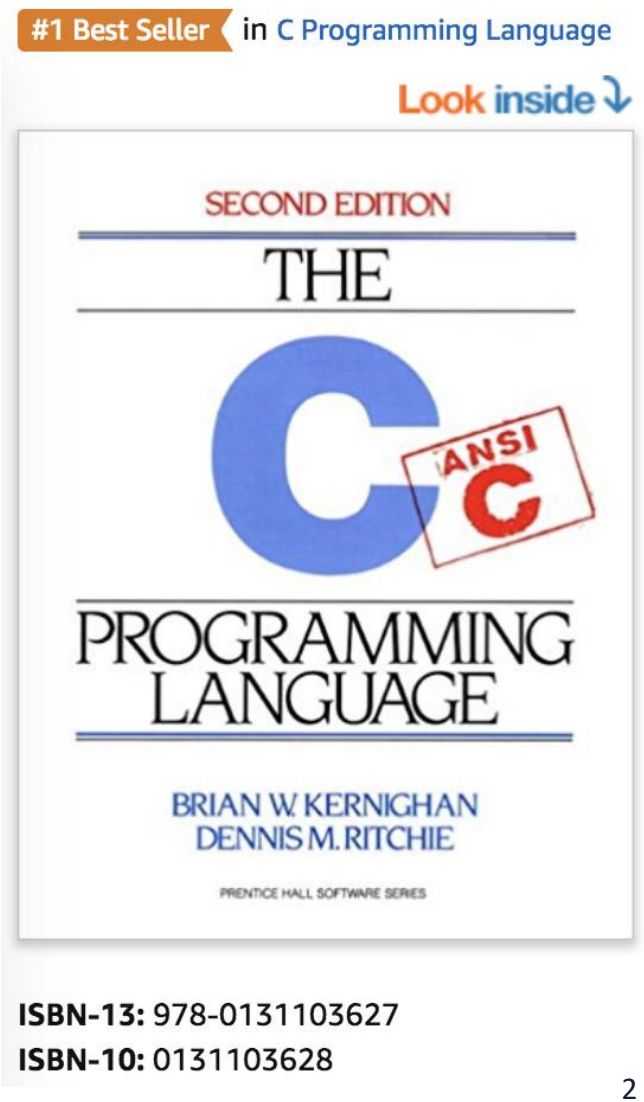
Programação em C

Arquitetura e Organização de Computadores

Conteúdo

- Relâmpago de C
  - Comparação de C com Java
  - “Boa Noite”
  - Preprocessor
  - Argumentos de linha de comando
  - Arrays e structures
  - Pointeiros e memória dinâmica

- Livro do K&R para detalhes
  - Toneladas de tutoriais na internet
  - Apêndice no blog



Se sabe Java, C é “moleza” (C ➔ Java)

1. Mesmos operadores do Java ➔ formam expressões de programações e comandos básicos para cálculos/operações
   - Aritmética
     - `i = i+1; i++; i--; i *= 2;`
     - `+, -, *, /, %,`
   - Relacionais e lógicas
     - `<, >, <=, >=, ==, !=`
     - `&&, ||, &, |, !`

2. Sintaxe basicamente do Java
   - `if ( ) { } else { }`
   - `while ( ) { }`
   - `do { } while ( );`
   - `for(i=1; i <= 100; i++) { }`
   - `switch ( ) {case 1: … }`
   - `continue; break;`

Tipos de Dados

- Primitivos

tipo	size (byte)	valores
char	1	-128 to 127
short	2	-32,768 to 32,767
int	4	-2,147,483,648 to 2,147,483,647
long	4	-2,147,483,648 to 2,147,483,647
float	4	3.4E+/-38 (7 digits)
double	8	1.7E+/-308 (15 digits long)
pointer		

- Complexos
  - Array: int A[100];
  - struct ~= classe

- **Declaração: nome e tipo**

Cuidado! (1)

```
{
   int i;
   for(i = 0; i < 10; i++)
      …
```

ERRADO:

```
{
   for(int i = 0; i < 10; i++)
      …
```

Cuidado! (2)

- Variáveis não inicializadas
  - Use a opção **`-Wall`** do compilador

```
#include <stdio.h>

int main(int argc, char* argv[])
{
   int i;
  factorial(i);
   return 0;
}
```

“Boa Noite!”

```
#include <stdio.h>
int main(int argc, char* argv[])
{
  /* isto é um comentário */
  printf(“Boa Noite!\n");
  return 0;
}
```

```
$ ./boanoite
Boa Noite!
$
```

Breaking down the code

- `#include <stdio.h>` ➔ importação
  - Inclui o CONTEÚDO do arquivo stdio.h
    - Diferencia maiúscula de minúscula – use minúsculas
  - Não tem ; no final da linha

- `int main(…)`
  - Função de entrada do programa.

- `printf(format_string, arg1, …)`
  - Imprime uma string, especificada pela format_string, com os argumentos a seguir.

Argumentos de Linha de Comando (1)

- `int main(int argc, char* argv[])`
- **`argc`**
  - Número de argumentos (incluindo o nome do programa)
- **`argv`**
  - Array de char*s
  - **`argv[0]`** : = nome do programa
  - **`argv[1]`** : = primeiro argumento
  - **…**
  - **`argv[argc-1]`** : último argumento

Argumentos de Linha de Comando (2)

```
#include <stdio.h>

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

Argumentos de Linha de Comando (3)

```
$ ./contargs As turmas SI1N e EE9N estão aqui
8 arguments
  0: ./contargs
  1: As
  2: turmas
  3: SI1N
  4: e
  5: EE9N
  6: estão
  7: aqui
$
```

Structures

- Parecem classes do Java, não têm método

```
#include <stdio.h>

struct pessoa {
   char*      nome;
   int        idade;
}; /* <== NÃO ESQUEÇA o ; */
int main(int argc, char* argv[])
{
   struct pessoa fulano;
   fulano.nome = "Fulano de Tal";
   fulano.idade = 25;

   printf("%s tem %d ano.\n", fulano.nome, fulano.idade);
   return 0;
}
```

Variáveis são localizações na memória

Variáveis são representações simbólicas de endereços de memória

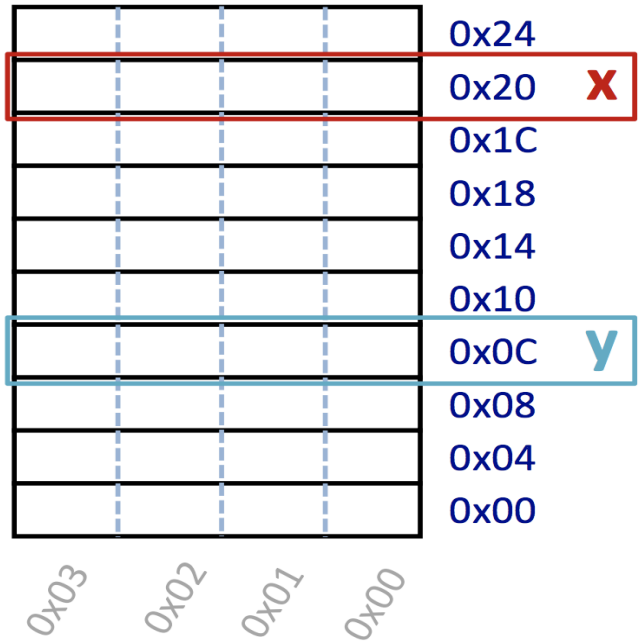
- Se do lado direito: ler/carregar o CONTEÚDO da localização
- Se do lado esquerdo: escrever o CONTEÚDO na localização

```
int x; // x at 0x20
int y; // y at 0x0C

x = 0; // store 0 at 0x20

// store 0x3CD02700 at 0x0C
y = 0x3CD02700;

// load the contents at 0x0C,
// add 3, and store sum at 0x20
x = y + 3;
```

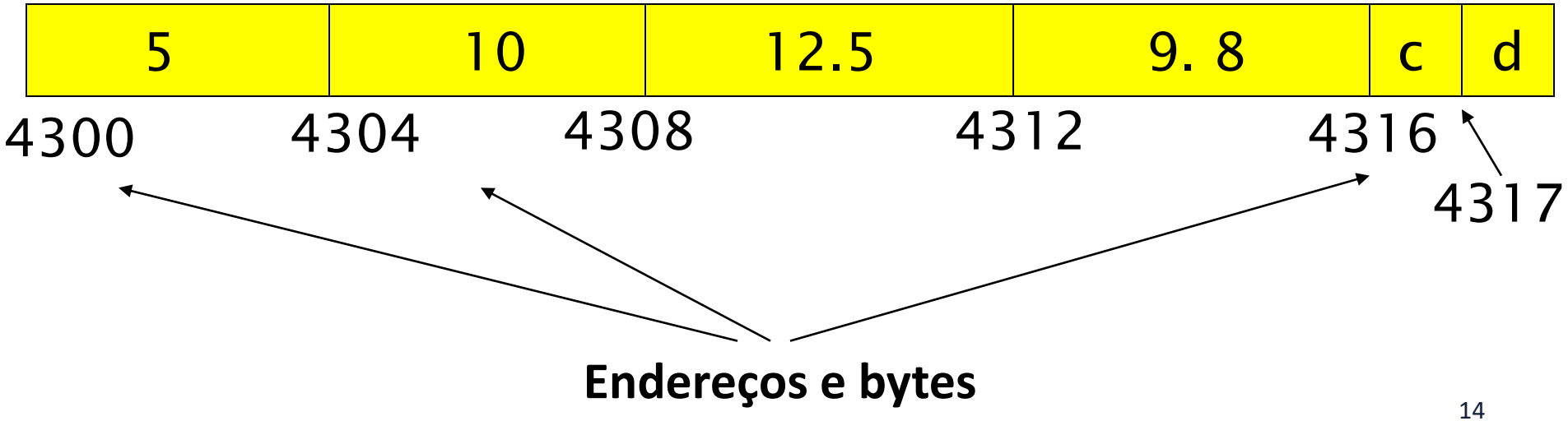


Layout e Endereços de Memória

```
int x = 5, y = 10;
float f = 12.5, g = 9.8;
char c = 'c', d = 'd';
```

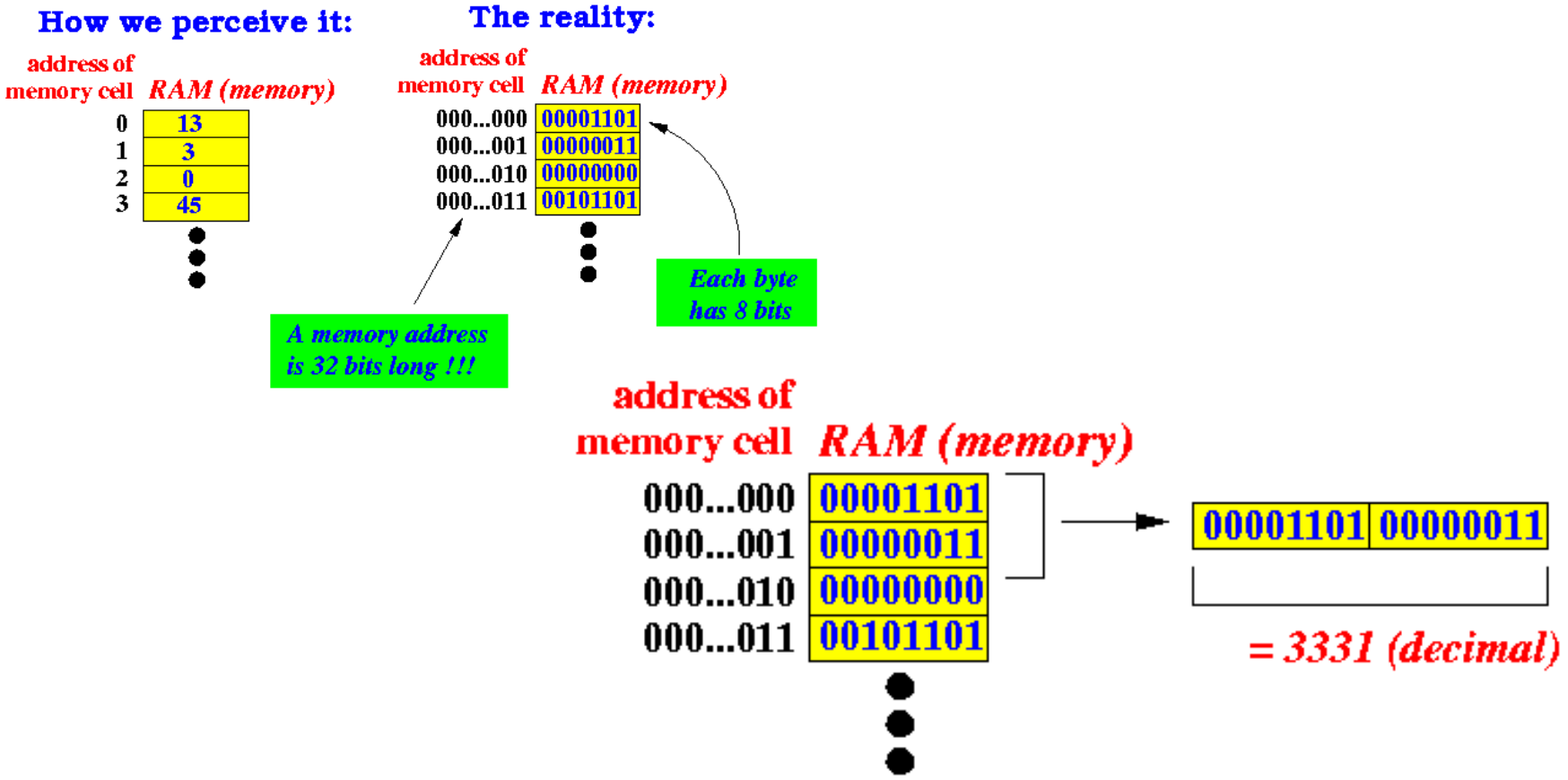
Sizes of data types

int: 4 bytes
float: 4 bytes
char: 1 byte
double: 8 bytes
long: 8 bytes



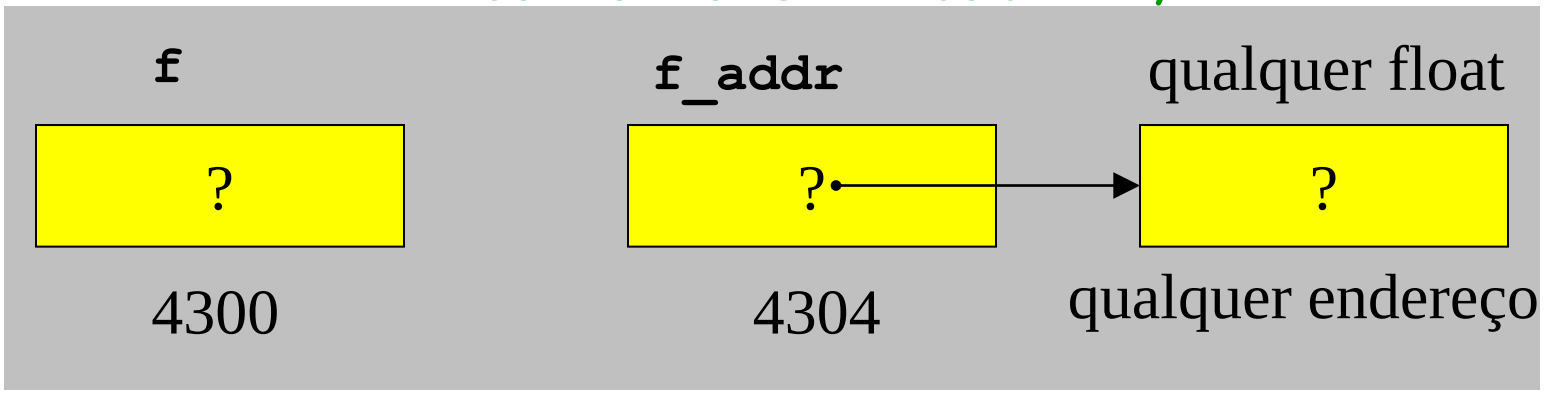
Ponteiros

- São variáveis cujo conteúdo é um ENDEREÇO de memória.
- Esse endereço *APONTA* para outra variável.

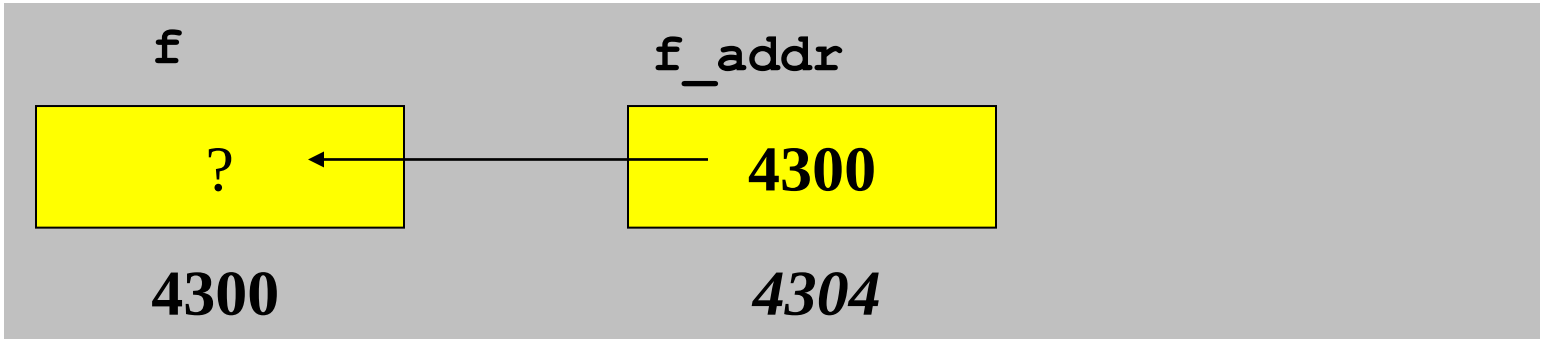


Usando Ponteiros (1)

```
float f;          /* variável float f */
float *f_addr;    /* ponteiro: armazena o endereço */
                   * da variável float f */
```

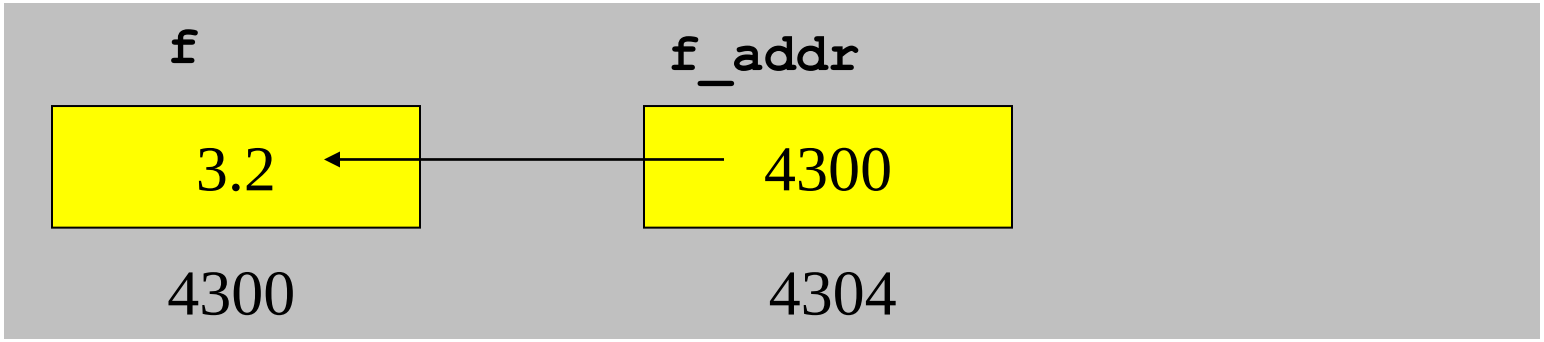



```
f_addr = &f;      /* & = operador para obter o endereço */
```

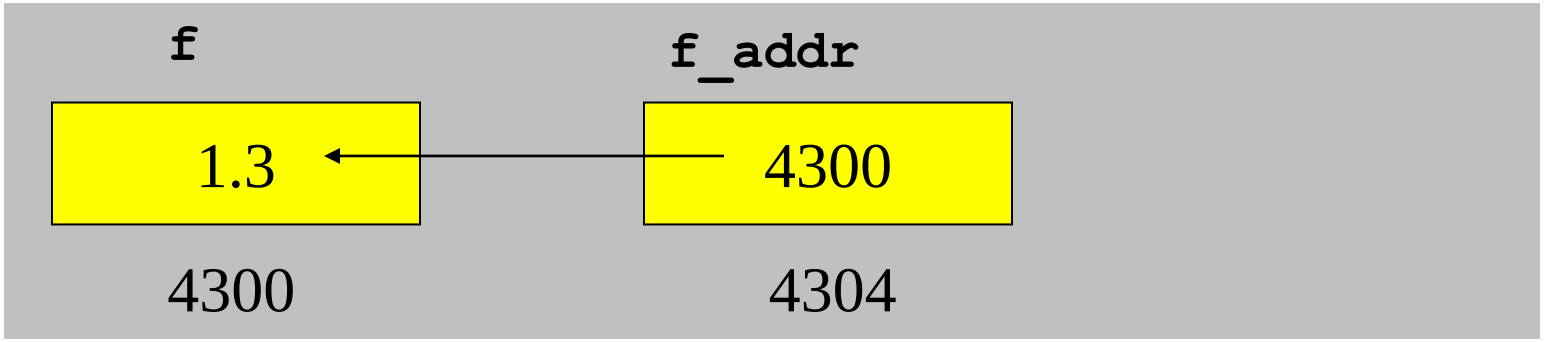


Usando Ponteiros (2)

```
*f_addr = 3.2; /* indirection: aloca o valor 3.2        */
               /* para a memória no endereço f_addr    */
```




```
float g = *f_addr; /* indirection: lê endereço f_addr */
                   /* e aloca em g (g agora é 3.2)     */
f = 1.3;           /* altera f mas g continua 3.2      */
```



Variável e Ponteiro em C

& = endereço de

*** = conteúdo em**

Declare a variable, p

```
int* p;
```

that will hold the address of a memory location holding an int

```
int x = 5;
int y = 2;
```

Declare two variables, x and y, that hold ints, and store 5 and 2 in them, respectively.

Get the address of the memory location representing x

```
p = &x;
```

... and store it in p. Now, "*p points to x.*"

Add 1 to the contents of memory at the address stored in p

```
y = 1 + *p;
```

... and store it in the memory location representing y.

Ponteiros e Memória

C assignment:

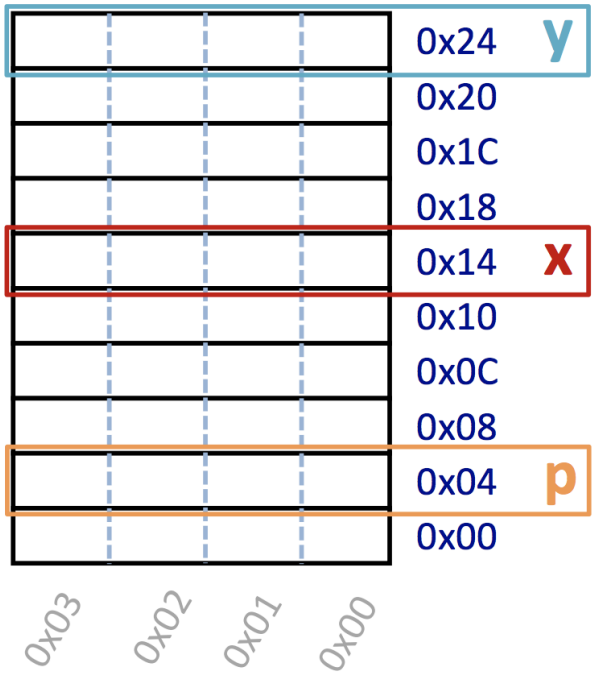
Left-hand-side = right-hand-side;

location value

& = endereço de

*** = conteúdo em**

```
int* p;      // p: 0x04
int x = 5;   // x: 0x14, store 5 at 0x14
int y = 2;   // y: 0x24, store 2 at 0x24
p = &x;      // store 0x14 at 0x04
// load the contents at 0x04  (0x14)
// load the contents at 0x14  (0x5)
// add 1 and store sum at 0x24
y = 1 + *p;
// load the contents at 0x04  (0x14)
// store 0xF0 (240) at 0x14
*p = 240;
```

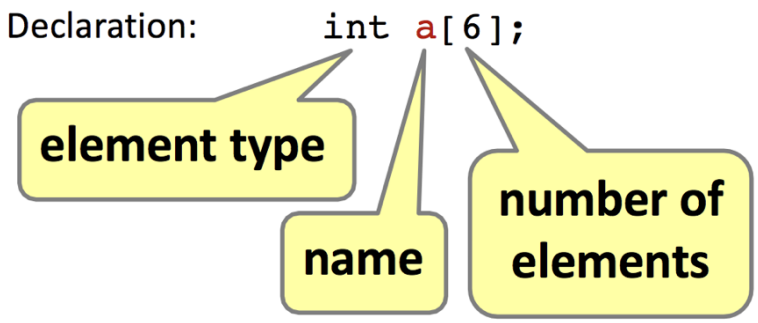


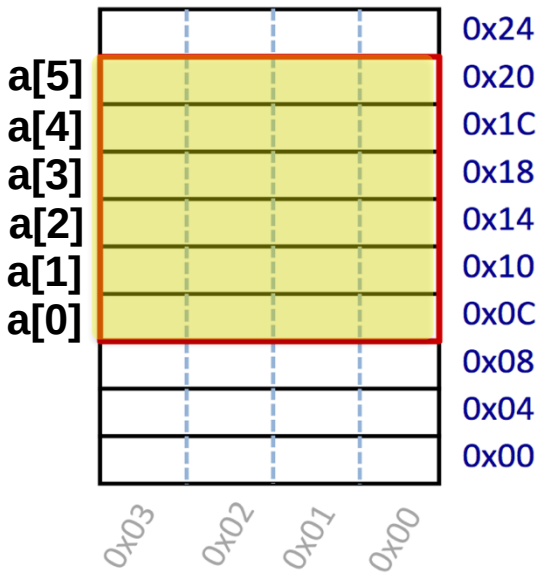
Variáveis do Ponto de Vista da ArqComp

- int a;
- float f;
- int* ap;
- float* fp
- char* str;
- char* argv[];
- Uma variável:
  - O nome é a representação simbólica do primeiro byte da localização na memória alocada para a variável
  - Tipo: tamanho da memória para a variável
    - char: 1 byte, int/float/long: 4 bytes; double: 8 bytes
    - char*, int*, float*, double*, void*: 4 ou 8 bytes, depende se é sistema de 32 ou 64 bits
  - Referência à variável == referência ao endereço
    - À direita do =: carrega o valor de um endereço, o tipo é usado para determinar quantos bytes ler/carregar
    - À esquerda do =: armazena um valor ao endereço, o tipo é usado para determinar quantos byes armazenar
  - &x = endereço de x
  - *p = conteúdo no endereço p

Arrays

- **Localizações adjacentes de memória que armazenam o mesmo tipo de dados**
  - **Elementos são "empilhados" no espaço de memória**
- `int a[6];` espaço para 6 inteiros
  - cada int tem 4 bytes

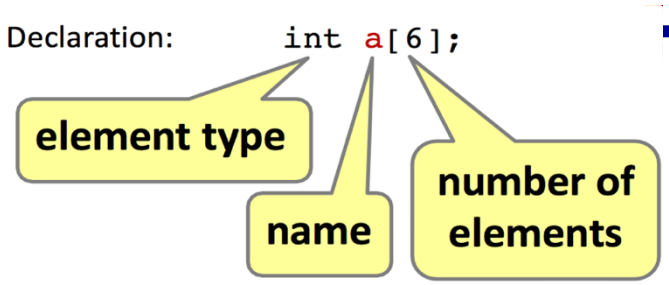


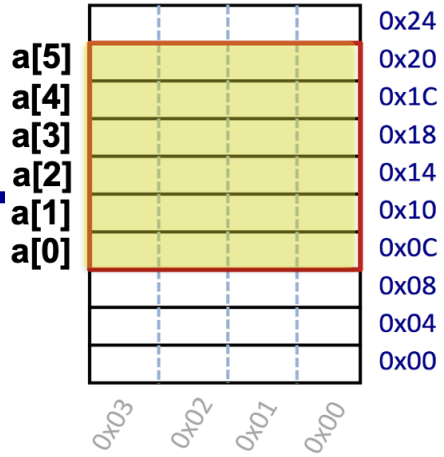


- a é o símbolo (variável) que representa o endereço base do array, que é o endereço do elemento a[0].
  - 0x0C

Endereços dos Elementos no Array

- `int a[6];`



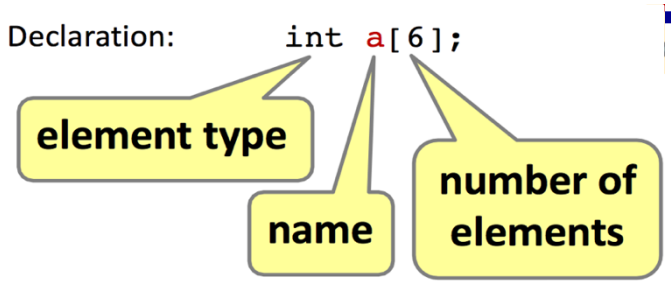


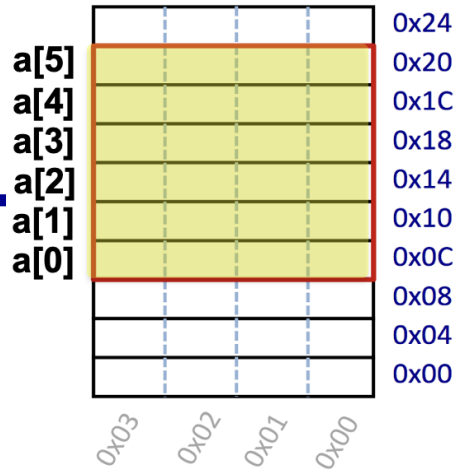
- **Offset de a[i]:** espaço (número de bytes) entre a[0] e a[i]
  - i*sizeof(int)
- **Byte address de a[i] (&a[i]): base + offset**

  &a[i]: (char*)a + i * sizeof(int)
  - Exemplo: &a[2]: 0x0C + 2 * 4 = 0x14
  - (char*)a é o cast do (int*) para (char*), para garantir que o compilador o reconheça como um byte address e assim possa ser somado com o i*sizeof(int)
  - **Em C, &a[i] também é a+i já que o compilador é capaz de fazer aritmética de ponteiro com o tamanho do tipo de dados do array**
    - **Assim &a[i]: a + i, é aritmética de ponteiro, não matemática**
- Por si mesmo, a também é o endereço do primeiro inteiro
  - *a e a[0] significam a mesma coisa

Endereços dos Elementos no Array

- `int a[6];`





- Offset de a[i] a partir de a[j]: espaço (número de byte) de a[j] para a[i]
  - (i-j)*sizeof(int)

&a[i]: (char*)&a[j] + (i-j) * sizeof(int), ou
&a[j] + i-j

  - Exemplo: se &a[3] é 0x18, o que é &a[5]

&a[5]: (char*)&a[3] + (5-3) * sizeof(int), ou
&a[3] + 5-3

  - Exemplo: se &a[4] é 0x1c, o que é &a[2]

&a[2]: (char*)&a[4] + (2-4) * sizeof(int), or
&a[4] + 2-4

C: Arrays

Declaration:

```
int a[6];
```

Indexing:

```
a[0] = 0xf0;
a[5] = a[0];
```

No bounds check:

```
a[6] = 0xBAD;
a[-1] = 0xBAD;
```

Pointers:

```
int* p;
```

equivalent

```
p = a;
p = &a[0];
```

```
*p = 0xA;
```

equivalent

```
p[1] = 0xB;
*(p + 1) = 0xB;
p = p + 2;
```

array indexing = address arithmetic

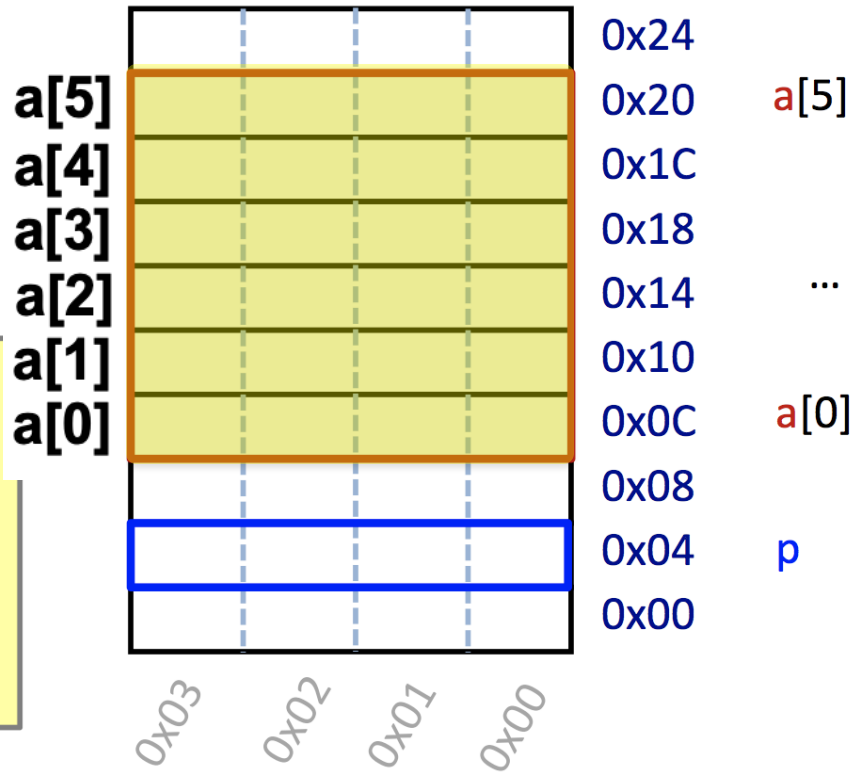
Both are scaled by the size of the type.

```
*p = a[1] + 1;
```

Arrays are adjacent memory locations storing the same type of data.

a is a name for the array's base address, can be used as an *immutable* pointer.

Address of **a[i]** is base address **a** plus **i** times element size in bytes.



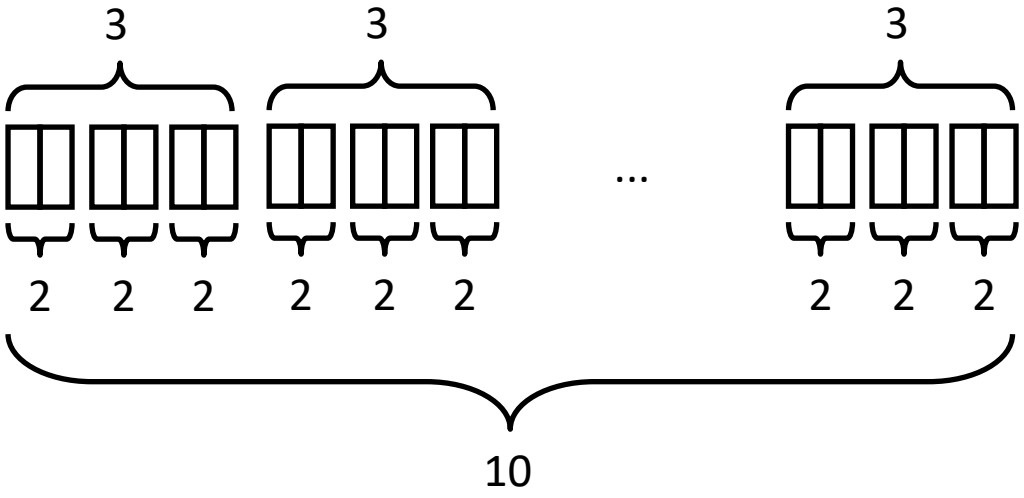
sizeof Arrays

- `int a[6];`
  - `sizeof(a)`
    = 6 × `sizeof(int)`
    = 6 × 4 = 24 bytes

- `char foo[80];`
  - Um array de 80 caracteres
  - `sizeof(foo)`
    = 80 × `sizeof(char)`
    = 80 × 1 = 80 bytes

Arrays Multidimensionais

- Quando declarados, a leitura é da direita para esquerda
- int a[10][3][2];
- “1 array de 10 arrays de 3 arrays de 2 ints”
- Na memória

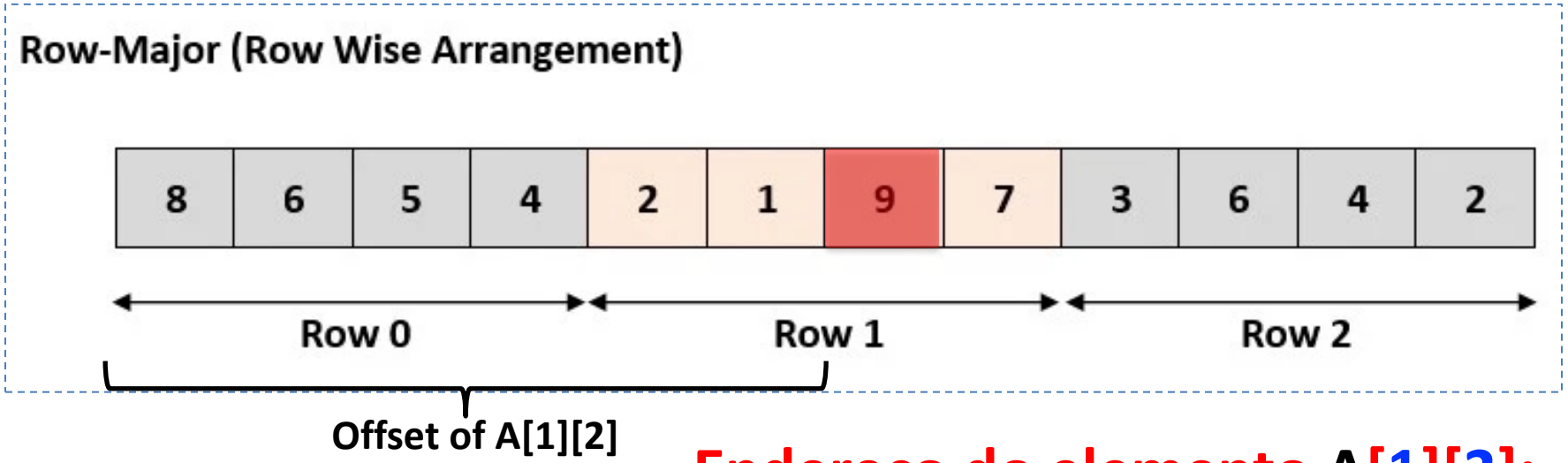


Seagram Building, Ludwig Mies van der Rohe,1957

Arrays na Memória: por LINHA

int A[3][4];

8	**6**	**5**	**4**
2	1	9	7
3	6	4	2



Endereço do elemento A[1][2]:

= (char*) A + offset (de A para A[1][2])

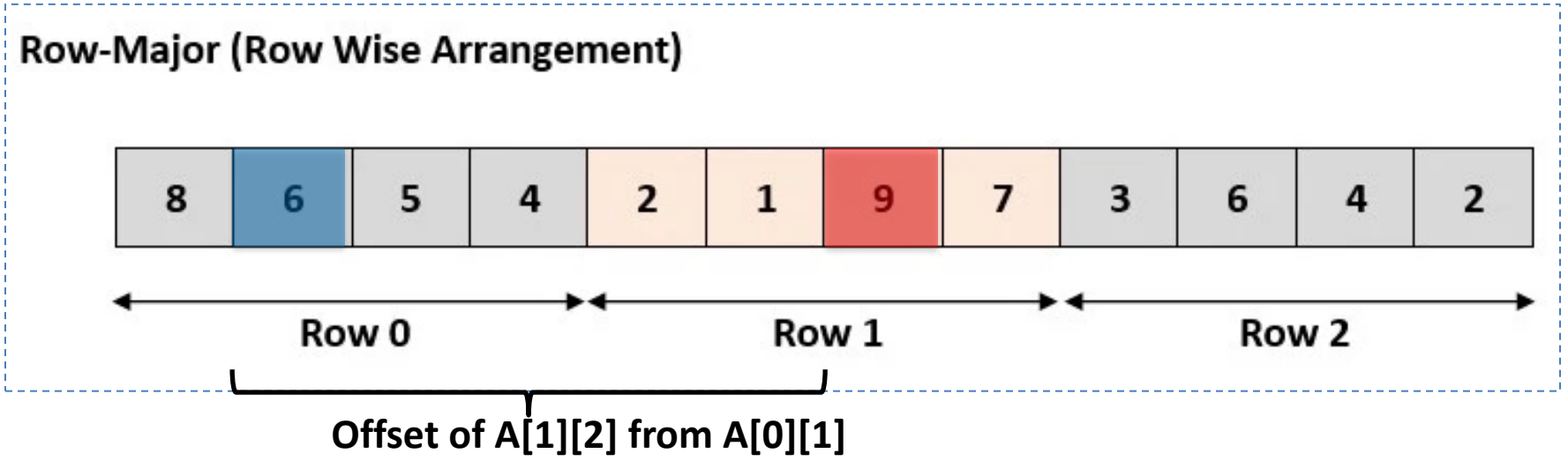
= (char*) A + sizeof (int) * (1 * 4 + 2)

= (char*) A + 4 * 6 = (char *)A + 24

Arrays na Memória: por LINHA

int A[3][4];

8	6	5	4
2	1	9	7
3	6	4	2



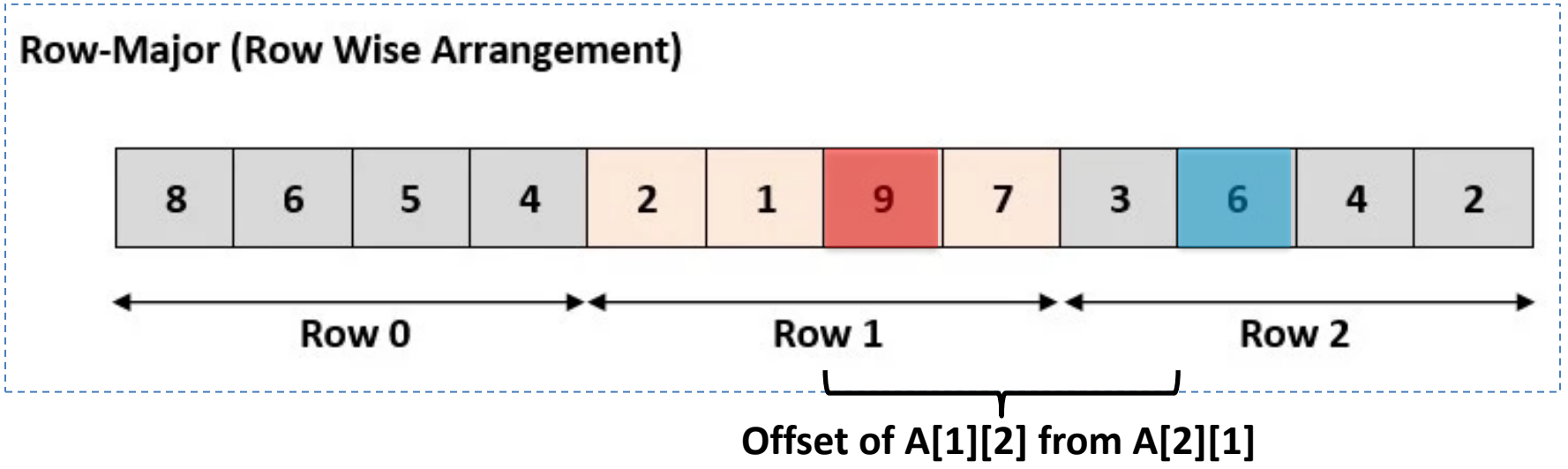
Dado o endereço de A[0][1], encontre o endereço do elemento A[1][2]:

= (char*) A[0][1] + offset (de A[0][1] para A[1][2])
= (char*) A[0][1] + sizeof (int) * ((1-0) * 4 + 2-1)
= (char*) A[0][1] + 4 * 5 = (char*) A[0][1] + 20

Arrays na Memória: por LINHA

int A[3][4];

8	**6**	**5**	**4**
2	1	9	7
3	6	4	2



Dado o endereço de A[2][1], encontre o endereço do elemento A[1][2]:

= (char*) A[2][1] + offset (de A[2][1] para A[1][2])

= (char*) A[2][1] + sizeof (int) * ((1-2) * 4 + 2-1)

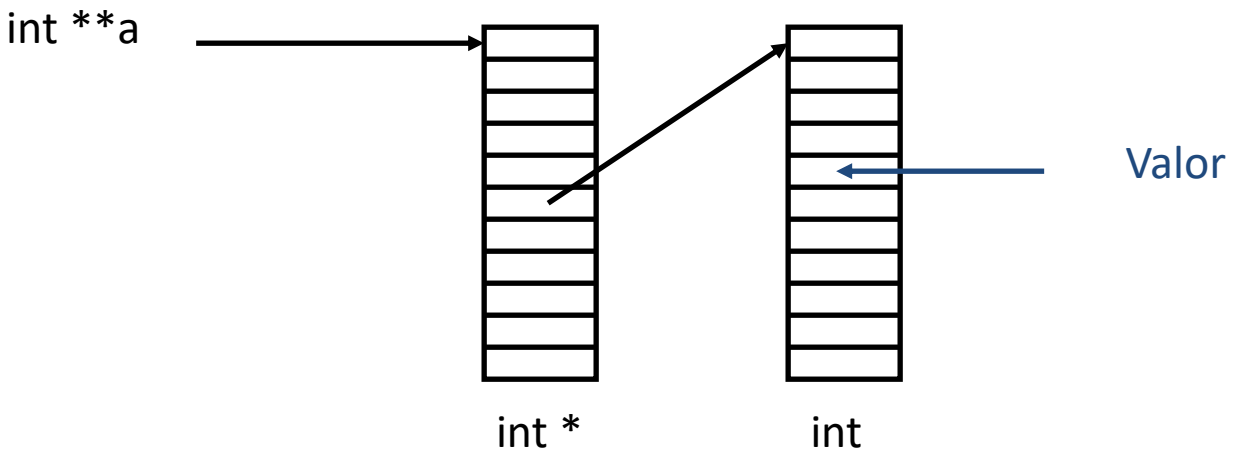
= (char*) A[2][1] + 4 * -3 = (char*) A[2][1] - 12

Arrays Multidimensionais em Java

- Arrays de ponteiros para arrays multidimensionais de tamanho variável
  - Precisa alocar espaço e inicializar o array de ponteiros

```
int **a;
```

`a[5][4]` expands to `*(*(a+5)+4)`

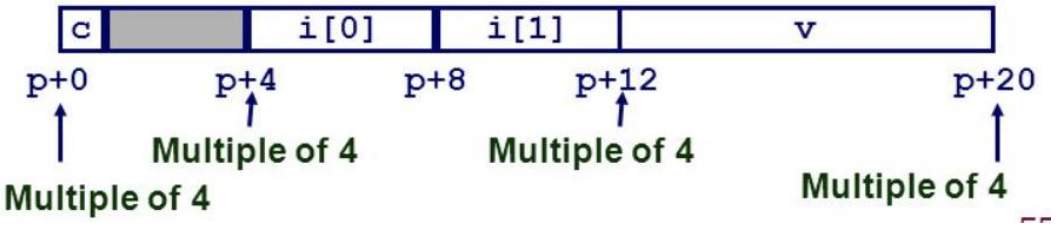


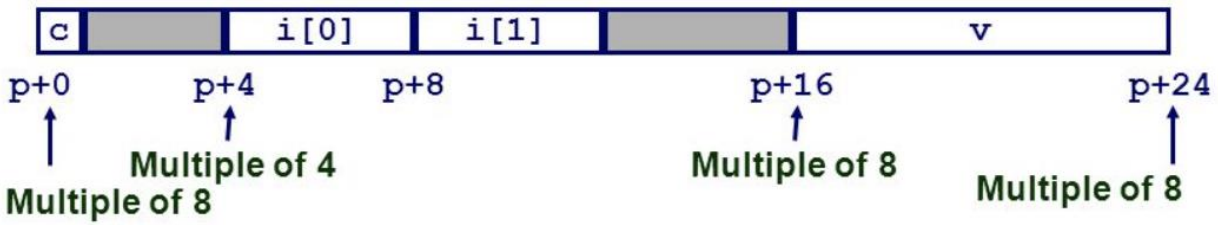
Endereços de Structures

- Semelhante a um array que “empilha” os campos da structure juntos
  - Complicado devido ao alinhamento
    - char: 1 byte, int: 4 bytes, double: 8 bytes

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

-





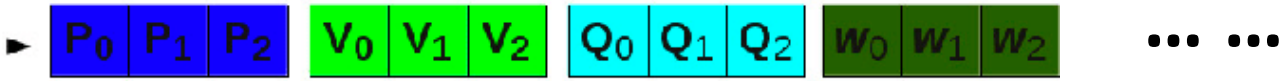
Array de Structs, e Struct de Arrays

```
1   struct astruct  {
2       int P, V, Q, W;
3   };
4   struct astruct anArrayOfStruct [100];
5
6   struct aStructOfArrayStruct {
7     int P[100];
8     int V[100];
9     int Q[100];
10    int W[100];
11  };
12  struct aStructOfArrayStruct aStructOfArray;
```

Memory Layout

Array of Structs AOS

Struct of Arrays SOA



Additional Topics for C Programming

- C Preprocessing
- Dynamic memory
- Function parameters
  - Pass by value
  - Pass a pointer

Olá Mundo!

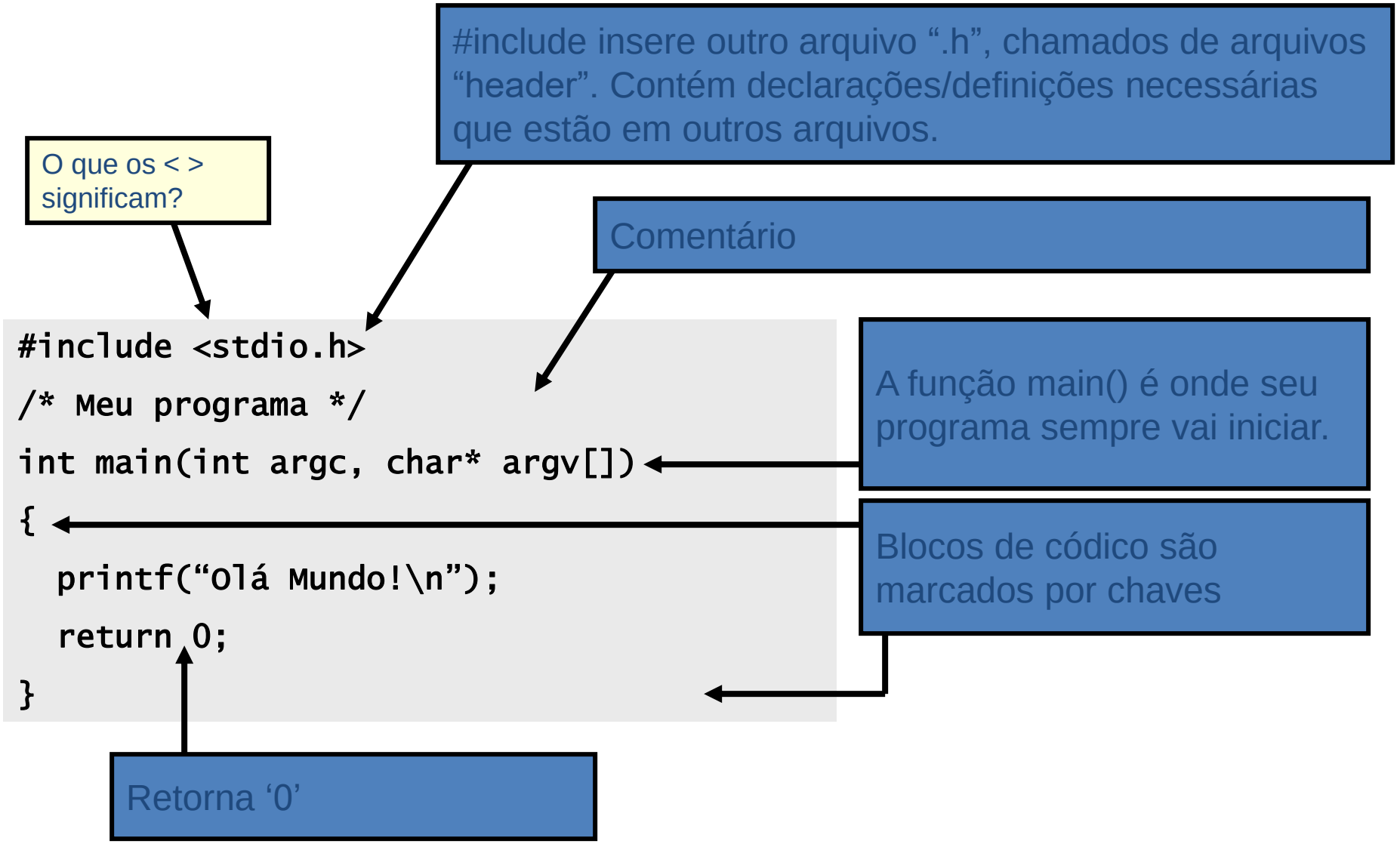
- ola.c

```
#include <stdio.h>

/* Meu programa */

int main(int argc, char* argv[]) {
  printf(“Olá Mundo!\n");
  return 0;
}
```

C Syntax and Hello World



Processo de Compilação

- gcc ola.c -o ola
  - Cria um executável
  - QUATRO estágios
  - Comando:
    gcc <opções> <código_fonte.c>

- Compiladores
  - gcc (GNU Compiler)
    - man gcc
  - icc (Intel C Compiler)
  - ...

4 Estágios do Processo de Compilação

Preprocessing
gcc -E ola.c -o ola.i
ola.c → ola.i

Compilation (após o preprocessing)
gcc -S ola.i -o ola.s

Assembling (após a compilation)
gcc -c ola.s -o ola.o

Linking de object files (após o assembling)
gcc ola.o -o ola

Saída → Executável (ola)
Rodar → ./ola (loader)

4 Estágios do Processo de Compilação

1. Preprocessing (aquilo que tem # …)
   - Expanssão dos arquivos Header (#include …)
   - Substituição de macros e funções inline (#define …)
2. Compilation
   - Gera linguagem assembly, na ISA escolhida
   - Verificação do uso de funções através dos prototypes
   - Arquivos Header: declaração dos prototypes
3. Assembling
   - Gera arquivo object
   - nm ola.o
     0000000000000000 T main
                      U puts
   - nm ou objdump para ver arquivos object

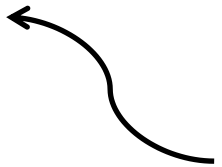
4 Estágios do Processo de Compilação

4. Linking
   - Gera o arquivo executável (nm para ver o arquivo)
   - Vincula as bibliotecas apropriadas
     - Static Linking
     - Dynamic Linking (padrão)

- Carregar e Executar
  - Avalia o tamanho do código e do segmento de dados
  - Aloca espaço de memória no modo do usuário e transfere tudo para memória
  - Carrega as bibliotecas necessárias e vincula
  - Invoca o Process Manager → Registra o Programa

Compilando um Programa

- *gcc <opções> nome.c*



4 estágios de uma vez

- Opções:

  -Wall: Mostra todos os avisos
  -o nome_do_output: O padrão é “a.out”. Especifique o nome do executável com a opção "**-o**".
  -g: Inclui informação de debuggin no binário.

- man gcc

Preprocessor

```
#define TURMAS "SI1N e EE9N\n"

int main(int argc, char* argv[])
{
  printf(TURMAS);
  return 0;
}
```

Após o preprocessor (gcc -E)

```
int main(int argc, char* argv[])
{
  printf("SI1N e EE9N\n");
  return 0;
}
```

Compilação Condicional!

```
#define CSCE212

int main(void)
{
  #ifdef CSCE212
  printf("Estou aqui\n");
  #else
  printf("Agora não estou\n");
  #endif
  return 0;
}
```

Dynamic Memory

- Java gerencia a memória para você, o C não
  - C exige que o programador explicitamente aloque e libere a memória
  - Quantidades não previamente conhecidas de memória podem ser alocadas dinamicamente durente a execução com **`malloc()`** e liberadas com **`free()`**

Diferenças com o Java

- Não usa **`new`**
- Não tem garbage collection
- Você pede por *n* bytes
  - Não é uma requisição de alto-nível, como "Eu preciso de uma instância da classe **`String`**"

malloc

- Aloca memória no heap
  - "Vive" entre invocação de funções:
    - Variáveis das funções desaparecem após o return

- Exemplo
  - Alocar um int
    - `int* iptr = (int*) malloc(sizeof(int));`
  - Alocar uma structure
    - `struct name* nameptr = (struct name*) malloc(sizeof(struct name));`

free

- Libera a memória do heap.
- Passe o ponteiro que foi retornado pelo **`malloc`**.
- Exemplo
  - ```
    int* iptr =
      (int*) malloc(sizeof(int));
    free(iptr);
    ```
- Atenção: não libere o mesmo bloco de memória duas vezes!
    ```

Parâmetros das Funções

- Os argumentos são passador por valor:
  - A função que está sendo chamada recebe uma cópia dos argumentos
  - A função chamada não consegue alterar as variáveis, só sua cópia local

Exemplo 1: swap_1

```
void swap_1(int a, int b)
{
   int temp;
   temp = a;
   a = b;
   b = temp;
}

void call_swap_1( ) {
   int x = 3;
   int y = 4;
   swap_1(x, y);
}
```

P: x=3, y=4.
Após
swap_1(x,y);
x =? y=?

A1: x=4; y=3;

A2: x=3; y=4;

Exemplo 2: swap_2

```
void swap_2(int *a, int *b)
{
   int temp;
   temp = *a;
   *a = *b;
   *b = temp;
}

void call_swap_2( ) {
   int x = 3;
   int y = 4;
   swap_1(&x, &y);
}
```

Q: x=3, y=4,
Após
swap_2(&x,&y);
x =? y=?

A1: x=3; y=4;

A2: x=4; y=3;

Exemplo 3: scanf (lê um input)

```
#include <stdio.h>

int main()
{
   int  x;
   scanf("%d\n", &x);
   printf("%d\n", x);
}
```

Q: Por que usar ponteiros no scanf?

A: Precisamos alocar o valor à x.