Estruturas de Dados I

Capítulo 2: Tipos de Dados em C

2025/1

Prof. Abrantes Araújo Silva Filho

Introdução

- Programas = Algoritmos + Estruturas de Dados
- Estrutura dos algoritmos:
  - Operações seqüenciais
  - Operações de decisão (seleção, condição)
  - Operações de repetição
- Estrutura dos dados:
  - Tipos de dados
  - Variáveis
  - Enumerações
  - Ponteiros
  - Arrays
  - Registros

Hierarquia dos tipos de dados

- Tipos de dados: **Objeto x Função**; **Vazio x Básico x Derivado**
- Vazio: `void`
- Básicos:
  - **Inteiros (escalar):**
    - `char`
    - Signed
      - `char`
      - `[short, long, long long] int`
    - Unsigned
      - `_Bool`
      - `char`
      - `[short, long, long long] int`
    - Enumeração
      - `enum`

Hierarquia dos tipos de dados

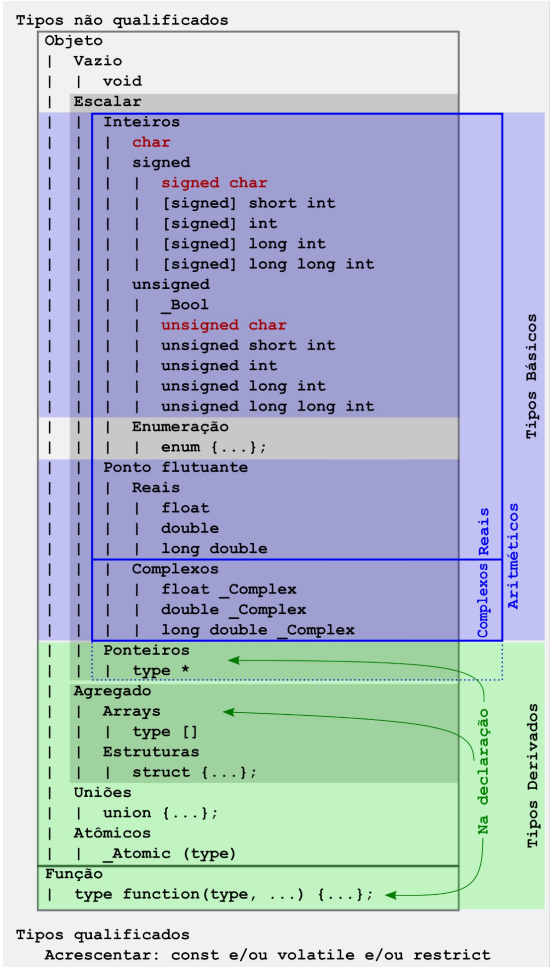
- Básicos (cont.):
  - **Ponto flutuante (escalar):**
    - Reais
      - `float`
      - `[long] double`
    - Complexos
      - `float _Complex`
      - `[long] double _Complex`
    - Imaginários
      - `float _Imaginary`
      - `[long] double _Imaginary` d

Hierarquia dos tipos de dados

- Derivados
  - **Ponteiros (escalar):**
    - `type *`
  - **Arrays (agregado):**
    - `type [ ]`
  - **Estrutura (agregado):**
    - `type struct { }`
  - **Uniões (escalar):**
    - `type union { }`
  - **Funções:**
    - `type function (type, …) { }`
- Conceito de escalar:
  - Não divisíveis
  - Compilador converte para o valor (inteiro, flutuante) correspondente

Hierarquia de Tipos de Dados em C

(ISO/IEC 9899:2018)



Abrantes Araújo Silva Filho (baseado, adaptado e atualizado de: **Krzysztof Abramowicz**, em: https://stackoverflow.com/a/20577958/6850077)

Enumerações

- Básico, inteiro, escalar
- Permitem criar novos tipos escalares, listando os elementos de seu domínio
- É um tipo cujo valor é o valor de seu tipo subjacente, que inclui o valor de constantes nomeadas explicitamente definidas (enumeration constants)
- ATENÇÃO! O tipo é:
  - `enum identificador`

```c
#include <stdio.h>

enum bussola
{
    NORTE,      // 0
    NORDESTE,   // 1
    LESTE,      // 2
    SUDESTE,    // 3
    SUL,        // 4
    SUDOESTE,   // 5
    OESTE,      // 6
    NOROESTE    // 7
};

int main (void)
{
    enum bussola direcao = LESTE;
    direcao = NORTE;

    printf("%u\n", direcao);
    return 0;
}
```

```c
#include <stdio.h>

int main (void)
{
    // tipo: "enum cor" com as enumeration constants:
    //       VERMELHO = 0; VERDE = 1; AZUL = 2
    enum cor { VERMELHO, VERDE, AZUL };

    // variável c do tipo: "enum cor", e valor 1
    enum cor c = VERDE;

    // objeto *pc do tipo: "ponteiro para enum cor"
    enum cor *pc = &c;

    printf("%u\n", *pc);

    return 0;
}
```

Enumerações

```
enum centavos_americanos
{
    Penny = 1,
    Nickel = 5,
    Dime = 10,
    Quarter = 25,
    Half = 50
};
```

```
enum teste
{
    A, B, C = 10, D, E = 1, F, G = F + C
};
// A=0; B=1; C=10; D=11; E=1; F=2; G=12
```

```
enum mes
{
    Janeiro = 1, Fevereiro, Marco, Abril, Maio, Junho,
    Julho, Agosto, Setembro, Outubro, Novembro, Dezenbro
};
```

Enumerações

- Como são inteiros escalares, podemos usar em qualquer situação onde um inteiro poderia ser utilizado

```c
#include <stdio.h>

enum bussola
{
    NORTE, NORDESTE, LESTE, SUDESTE,
    SUL, SUDOESTE, OESTE, NOROESTE
};

int main (void)
{
    enum bussola b = LESTE;

    switch (b)
    {
    case NORTE:
        puts("Norte");
        break;
    case LESTE:
        puts("Leste");
        break;
    case SUL:
        puts("Sul");
        break;
    default:
        puts("Outra direção");
        break;
    }
}
```

Enumerações

- Se for uma enumeração cíclica, podemos facilmente usar o operador de resto da divisão para obter o próximo valor

```
enum bussola
{
    NORTE, NORDESTE, LESTE, SUDESTE,
    SUL, SUDOESTE, OESTE, NOROESTE
};

int proxima_direcao(enum bussola b)
{
    return ((b + 1) % 8);
}
```

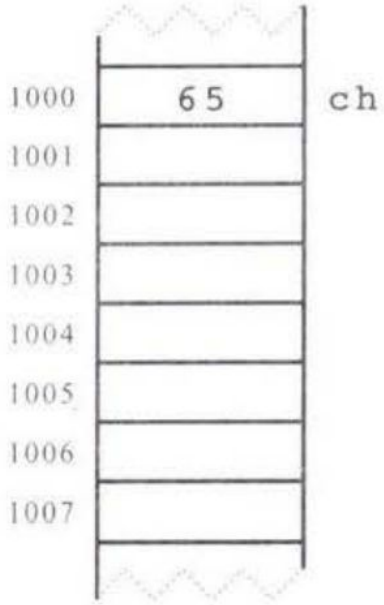
Enumerações

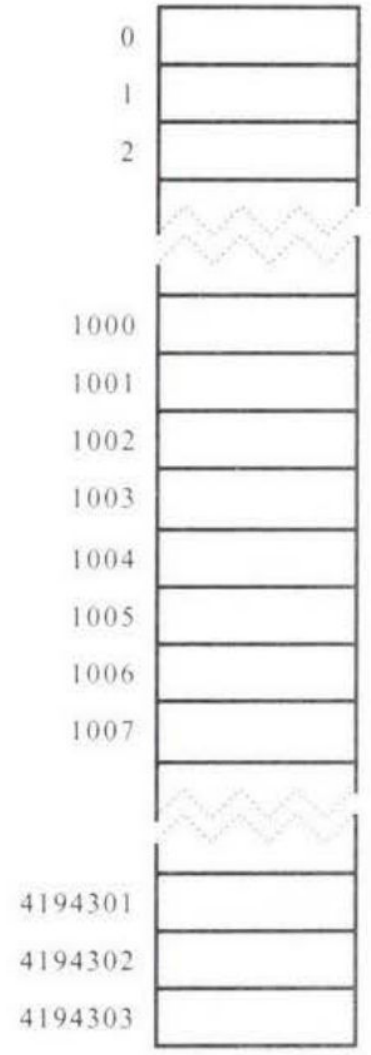
- Cuidado com o nome do tipo! Se achar melhor, defina um nome usando typedef

```c
#include <stdio.h>

int main (void)
{
    enum cor { VERMELHO, VERDE, AZUL };

    enum cor c1 = VERDE;    // CERTO

    // cor c2 = AZUL;       // ERRADO!!!!

    // Uso de typedef para criar um novo NOME DE TIPO:
    typedef enum cor cor_t;

    cor_t c2 = AZUL;        // CERTO

}
```

Dados e memória

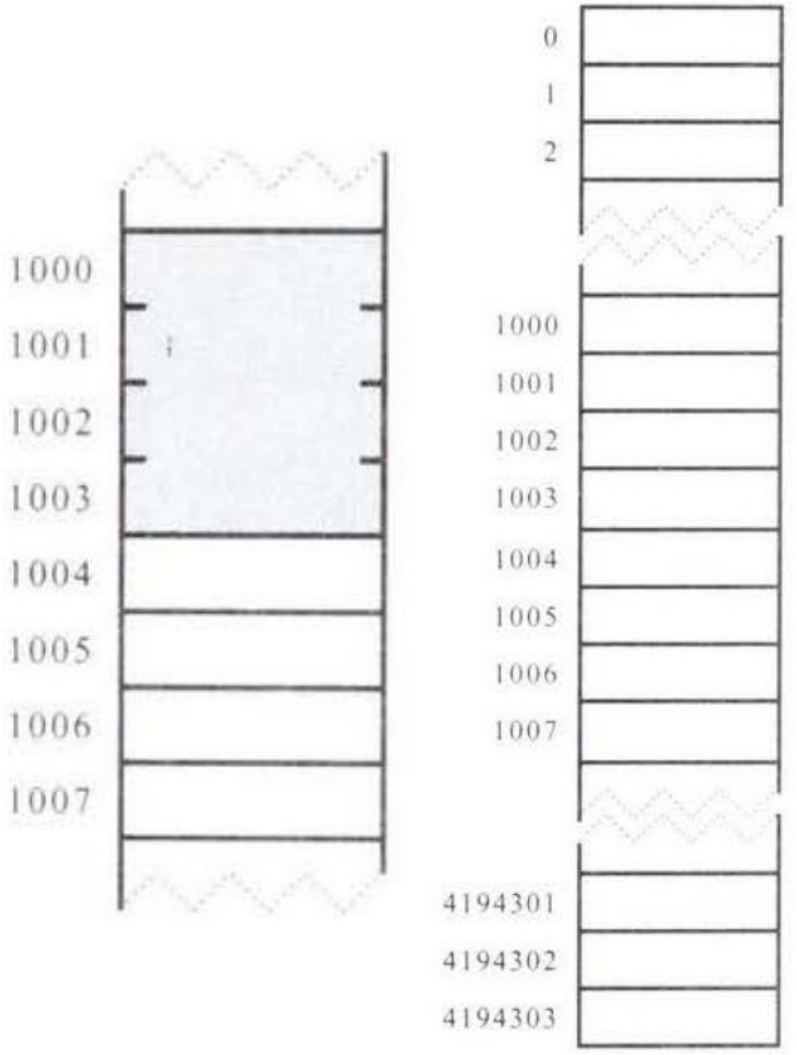
- Lembre-se de coisas básicas:
  - bit, byte, word
  - Endereço de memória da célula
  - Célula = 1 byte
  - Cada byte tem o tamanho suficiente para armazenar 1 char, geralmente 8 bits
  - Não é possível predizer o endereço que uma variável será armazenada





Dados e memória

- Lembre-se de coisas básicas:
  - Valores que são maiores do que 1 byte são armazenados em células consecutivas de memória, ex.:
    - int = 32 bits = 4 bytes
  - Valores que ocupam mais de 1 byte são identificados pelo endereço do primeiro byte
  - O operador **sizeof** retorna quantos bytes serão necessários para um objeto particular. O operando de sizeof é um tipo de dado ou uma expressão.



Ponteiros

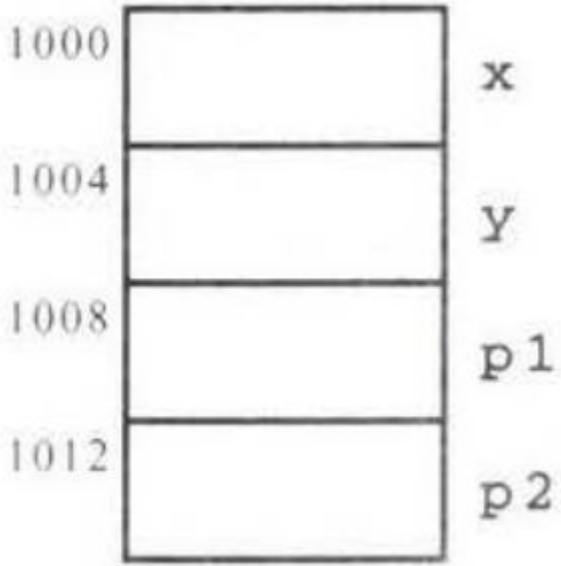
- Tipo de dado que armazena um endereço de memória
- Servem para muitas coisas:
  - Referenciar uma grande estrutura de dados de modo compacto
  - Facilitam a troca e o compartilhamento de dados entre diferentes partes de um programa
  - Permitem alocar memória “nova” durante a execução do programa
  - Permitem estabelecer relações entre itens de dados, modelar conexões entre os dados
- Lembrete:
  - **lvalue** (**lhs**): uma expressão que se refere a uma localização da memória capaz de armazenar um dado (estão à esquerda de uma sentença de atribuição).
    - Todo lvalue é armazenado em algum lugar na memória e tem um endereço
    - Uma vez declarado, o endereço do lvalue nunca muda (o conteúdo pode mudar)
    - Dependendo do tipo, o lvalue pode necessitar de quantidades diferentes de memória
    - O endereço do lvalue é o valor de um ponteiro, que pode ser armazenado na memória e manipulado como dado

Ponteiros: declaração

```c
#include <stdio.h>

int main (void)
{
    int *p;         //      tipo: ponteiro para int
                    // tipo base: int

    char *pc;       //      tipo: ponteiro para char
                    // tipo base: char


    // CUIDADO! O asterisco pertence sintaticamente
    // ao nome da variável, não ao tipo base:

    int *p1, *p2;     // duas variáveis do tipo ponteiro para int
    int *p1, p2;      // um ponteiro para int (p1) e um inteiro (p2)
}
```

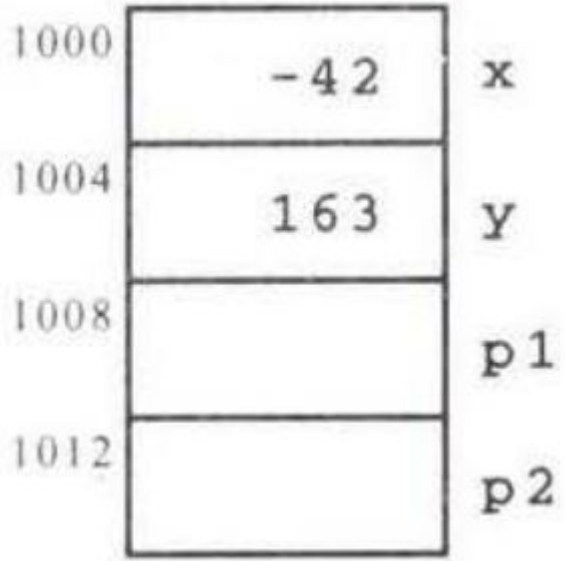
Ponteiros: operações fundamentais

```c
#include <stdio.h>

int main (void)
{
    int x, y;
    int *p1, *p2;

    x = -42, y = 163;

    p1 = &x;   // O operador & retorna o endereço
    p2 = &y;   // onde um lvalue está armazenado.

    *p1 = 17;  // O operador * retorna o lvalue que
               // é apontado pelo ponteiro. Isso se
               // chama DESREFERENCIAR o ponteiro.
               // Note que o * produz um lvalue e,
               // por isso, podemos atribuir valor.

    p1 = p2;   // ALOCAÇÃO DE PONTEIRO:
               // Atribui um novo valor ao ponteiro.
               // Idêntico à copia de um inteiro.
               // Cria um "aliase"

    *p1 = *p2; // ALOCAÇÃO DE VALOR:
               // Afeta os lvalues apontados.
}
```



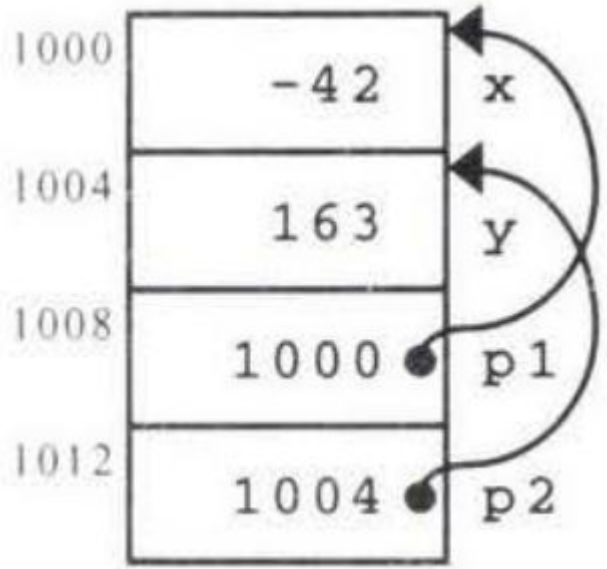
Ponteiros: operações fundamentais

```c
#include <stdio.h>

int main (void)
{
    int x, y;
    int *p1, *p2;

    x = -42, y = 163;

    p1 = &x;   // O operador & retorna o endereço
    p2 = &y;   // onde um lvalue está armazenado.

    *p1 = 17;  // O operador * retorna o lvalue que
               // é apontado pelo ponteiro. Isso se
               // chama DESREFERENCIAR o ponteiro.
               // Note que o * produz um lvalue e,
               // por isso, podemos atribuir valor.

    p1 = p2;   // ALOCAÇÃO DE PONTEIRO:
               // Atribui um novo valor ao ponteiro.
               // Idêntico à copia de um inteiro.
               // Cria um "aliase"

    *p1 = *p2; // ALOCAÇÃO DE VALOR:
               // Afeta os lvalues apontados.
}
```



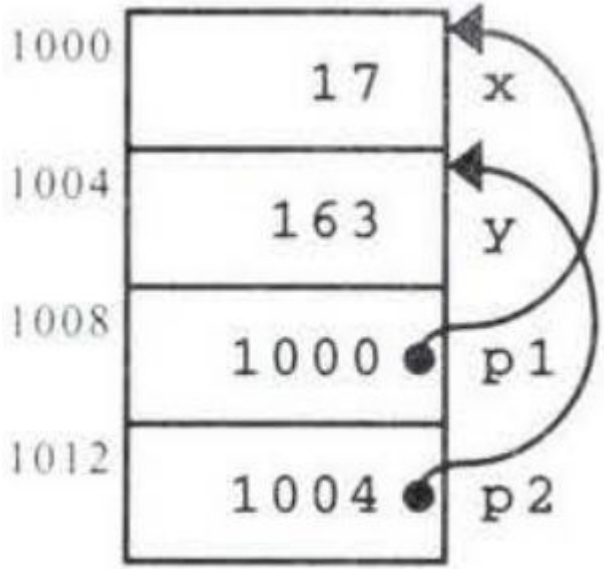
Ponteiros: operações fundamentais

```c
#include <stdio.h>

int main (void)
{
    int x, y;
    int *p1, *p2;

    x = -42, y = 163;

    p1 = &x;   // O operador & retorna o endereço
    p2 = &y;   // onde um lvalue está armazenado.

    *p1 = 17;  // O operador * retorna o lvalue que
               // é apontado pelo ponteiro. Isso se
               // chama DESREFERENCIAR o ponteiro.
               // Note que o * produz um lvalue e,
               // por isso, podemos atribuir valor.

    p1 = p2;   // ALOCAÇÃO DE PONTEIRO:
               // Atribui um novo valor ao ponteiro.
               // Idêntico à copia de um inteiro.
               // Cria um "aliase"

    *p1 = *p2; // ALOCAÇÃO DE VALOR:
               // Afeta os lvalues apontados.
}
```



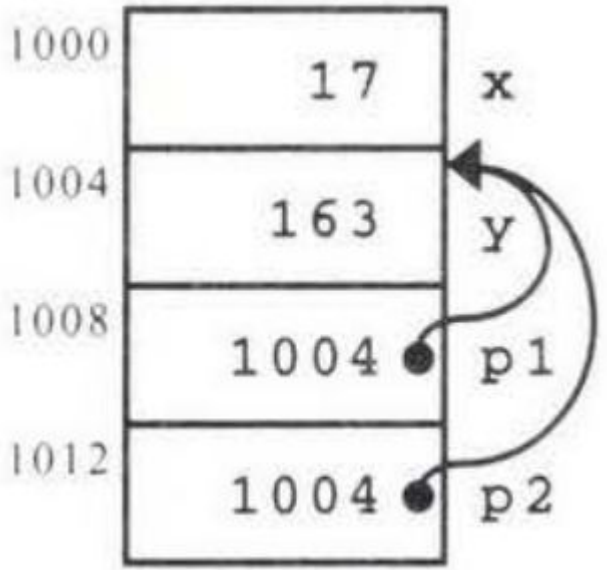
Ponteiros: operações fundamentais

```c
#include <stdio.h>

int main (void)
{
    int x, y;
    int *p1, *p2;

    x = -42, y = 163;

    p1 = &x;   // O operador & retorna o endereço
    p2 = &y;   // onde um lvalue está armazenado.

    *p1 = 17;  // O operador * retorna o lvalue que
               // é apontado pelo ponteiro. Isso se
               // chama DESREFERENCIAR o ponteiro.
               // Note que o * produz um lvalue e,
               // por isso, podemos atribuir valor.

    p1 = p2;   // ALOCAÇÃO DE PONTEIRO:
               // Atribui um novo valor ao ponteiro.
               // Idêntico à copia de um inteiro.
               // Cria um "aliase"

    *p1 = *p2; // ALOCAÇÃO DE VALOR:
               // Afeta os lvalues apontados.
}
```



Ponteiros: operações fundamentais

```c
#include <stdio.h>

int main (void)
{
    int x, y;
    int *p1, *p2;

    x = -42, y = 163;

    p1 = &x;   // O operador & retorna o endereço
    p2 = &y;   // onde um lvalue está armazenado.

    *p1 = 17;  // O operador * retorna o lvalue que
               // é apontado pelo ponteiro. Isso se
               // chama DESREFERENCIAR o ponteiro.
               // Note que o * produz um lvalue e,
               // por isso, podemos atribuir valor.

    p1 = p2;   // ALOCAÇÃO DE PONTEIRO:
               // Atribui um novo valor ao ponteiro.
               // Idêntico à copia de um inteiro.
               // Cria um "aliase"

    *p1 = *p2; // ALOCAÇÃO DE VALOR:
               // Afeta os lvalues apontados.
}
```



Ponteiros: NULL

- NULL é uma constante especial (definida em stdlib.h) que serve para **indicar que o ponteiro não aponta para nenhum dado válido**. Internamente NULL é representado pelo valor 0.
- NUNCA DESREFERENCIE UM PONTEIRO NULL!!! NULL indica que o ponteiro não contém dados válidos, então desreferenciar não faz sentido (e, de quebra, vai quebrar seu programa).

Ponteiros: NULL

```c
#include <stdio.h>

int main (void)
{
    int x = 50;
    int *px = &x;

    px = NULL;
    printf("x = %d\n", *px);  // ERRO!!!!
}
```

```
[abrantesasf@ideapad ~]$ make ponteiro3
cc     ponteiro3.c  -std=c17 -Wall -Wpedantic -lcs50 -lm -lcrypt -ledit -o ponteiro3

[abrantesasf@ideapad ~]$ ./ponteiro3
Segmentation fault (core dumped)
```

Ponteiros e passagem de argumentos

- Na linguagem C **não existe passagem por referência**, todos os argumentos passados aos parâmetros são **passados por VALOR**, ou seja, os parâmetros recebem uma cópia dos valores dos argumentos.

- Podemos **usar ponteiros para simular passagem por referência**! Continua sendo passagem por valor (o parâmetro recebe uma cópia do ponteiro), mas podemos manipular os dados originais dos argumentos.

Ponteiros e passagem de argumentos

```c
#include <stdio.h>

void trocar(int x, int y)
{
    int temp = x;
    x = y;
    y = temp;
}

int main (void)
{
    int x = 10, y = 20;

    printf(" Antes: x = %d e y = %d\n", x, y);

    trocar(x, y);

    printf("Depois: x = %d e y = %d\n", x, y);
}
```

Ponteiros e passagem de argumentos

```c
#include <stdio.h>

void trocar(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

int main (void)
{
    int x = 10, y = 20;

    printf(" Antes: x = %d e y = %d\n", x, y);

    trocar(&x, &y);

    printf("Depois: x = %d e y = %d\n", x, y);
}
```

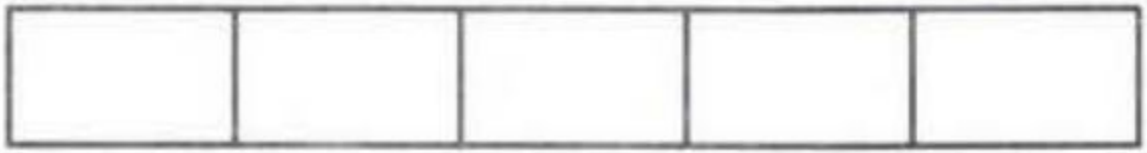
Ponteiros: tamanho

- Um ponteiro precisa ser grande o suficiente para armazenar o maior endereço de memória da arquitetura, portanto:
  - 8 bits 256 endereços
  - 16 bits 65536 endereços
  - 32 bits 4 bilhões de endereços 4 GB
  - 64 bits 18 quintilhões de endereços 16 EB (em teoria)

- Em arquiteturas 64 bits, os ponteiros têm 64 bits (8 bytes)!
  - Nas figuras anteriores sobre ponteiros, estávamos usando um arquitetura de quantos bits?
  - Obs.: na verdade os ponteiros têm 64 bits atualmente, mas apenas 48 são efetivamente utilizados pela CPU para endereçamento virtual

Arrays

- São **coleções ordenadas de dados homogêneos** (dados do mesmo tipo). Cada item do array é um **elemento**.



- Propriedade fundamentais:
  - **Tipo do elemento**
  - **Tamanho da array**

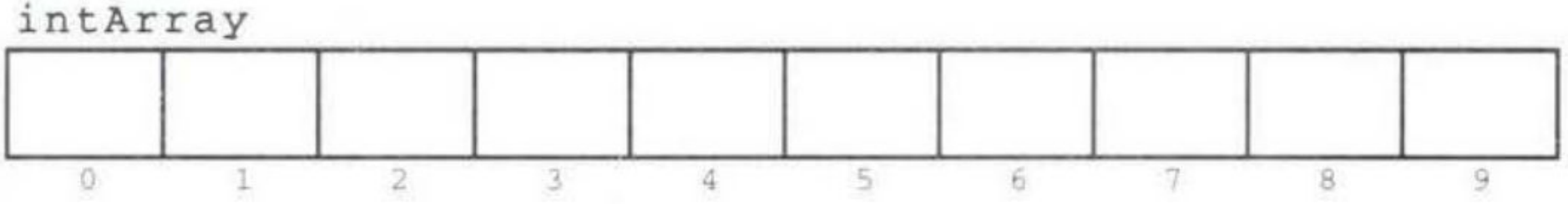
- Declaração:

```
tipo nome[tamanho];
```

Arrays

- Use **constantes simbólicas** para tornar seu programa mais gerenciável
- **Índice** começa em 0 e representa o **deslocamento, em bytes, a partir do 1º elemento**. O compilador calcula corretamente pelo tamanho do tipo.

```c
#include <stdio.h>

#define TAMANHO 10

int main (void)
{
    int intArray[TAMANHO];
}
```



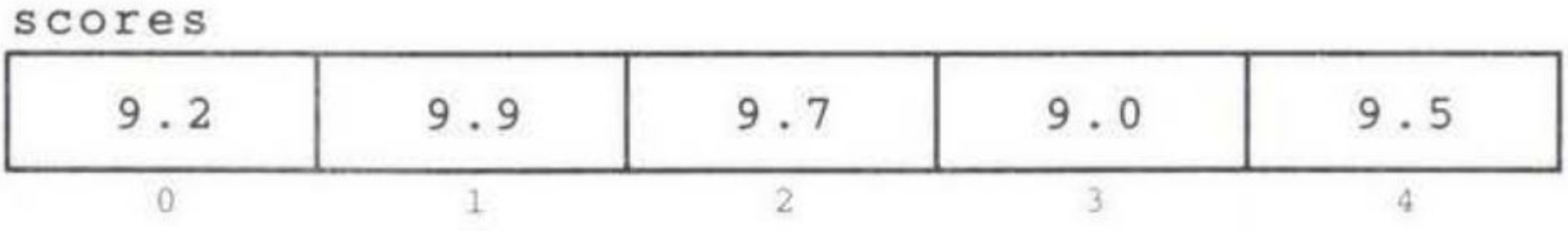
Arrays: seleção de elementos

- Para acessar os elementos individuais é necessário utilizar uma expressão de seleção no formato:

  `array[índice]`

- O índice não precisa ser constante, pode ser qualquer expressão que avalia para um inteiro.

```c
#include <stdio.h>

#define TAMANHO 5

int main (void)
{
    double scores[TAMANHO];

    scores[0] = 9.2;
    scores[1] = 9.9;
    scores[2] = 9.7;
    scores[3] = 9.0;
    scores[4] = 9.5;
}
```



Arrays: tamanho alocado e tamanho efetivo

- Em algumas situações não sabemos previamente quantos elementos o array terá. Nesses casos costuma-se declarar um array com o maior número de elementos que será útil na situação em questão mesmo que, depois, nem todos esses elementos sejam utilizados.
  - **Tamanho alocado**: a quantidade de elementos especificada na declaração
  - **Tamanho efetivo**: a quantidade de elementos ativamente em uso

- Um array **NUNCA ALTERA SEU TAMANHO** em tempo de execução!

- Como alternativa podemos utilizar **variable-length** arrays, que são arrays cujo tamanho é dado por uma variável.
  - ATENÇÃO: o tamanho do array continua fixo!

```c
#include <cs50.h>
#include <stdio.h>

#define LIMITE 200

int main (void)
{
    int n;

    do
    {
        n = get_int("Informe o número de elementos do array: ");
    }
    while (n < 1 || n > LIMITE);

    // "Variable-length array": um array cujo tamanho é
    // dado por uma variável. Atenção: isso não significa
    // que o tamanho do array varia!
    double dados[n];
}
```

Arrays: inicialização

- É possível inicializar o array, ou seja, atribuir valores durante a declaração.
- Existem diferentes maneiras de inicializar arrays
- Se necessário é possível fazer com que TODOS os elementos sejam inicializadas com o valor 0 (zero), mas não é possível que todos os elementos sejam inicializados com outros valores (é preciso usar um loop para isso).

```c
/* Inicialização tradicional */
int numeros[5] = {4, 8, 13, 22, 45};  // inicialização básica

int numeros[] = {4, 8, 13, 22, 45};  // inicialização sem o tamanho

int numeros[5] = {4, 8, 13, 22, 45, 46}; // erro!

/* Inicializaão parcial */
int numeros[5] = {4, 8};  // os dois primeiros serão 4 e 8, e o resto 0

int numeros[5] = {0};  // todos os elementos serão 0

int numeros[5] = {1};  // o primeiro elemento será 1, o resto será 0

/* Inicialização designada */
int numeros[5] = {[2] = 13, [1] = 8, [0] = 4, [4] = 45, [3] = 22};

int numeros[5] = {[4] = 45};  // o último elemento será 45, o resto será 0

int numeros[12] = {31, 28, [4] = 31, 30, 31, [1] = 29}; // 31 29  0  0 31 30
                                                        // 31  0  0  0  0  0
int foo[] = {1, [6] = 23}; // 1, 0, 0, 0, 0, 0, 23

int bar[] = {1, [6] = 4, 9, 10};  // 1, 0, 0, 0, 0, 0, 4, 9, 10
```

```c
#include <cs50.h>
#include <stdio.h>

#define LIMITE 200

int main (void)
{
    int n;

    do
    {
        n = get_int("Informe o número de elementos do array: ");
    }
    while (n < 1 || n > LIMITE);

    double dados[n];

    // Para saber o tamanho de um array, existe um idioma
    // específico: sizeof(array) / sizeof(tipo)
    // Obs.: sizeof retorna um unsigned log int.
    printf("O usuário criou um array com %lu elementos.\n",
           sizeof(dados) / sizeof(double));
}
```

Arrays: passagem de argumentos

- Subprogramas podem receber arrays como argumentos.

- Não especificamos o número de elementos dos parâmetros para poder receber arrays com qualquer número de elementos.

- Ao passar arrays como argumentos, o que é passado é uma **CÓPIA do ENDEREÇO do 1º elemento do array**! Isso simula a passagem por referência. Costumamos dizer que o array foi “rebaixado para um ponteiro”.

```c
#include <cs50.h>
#include <stdio.h>

int main (void)
{
    int digitos[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};

    printf("Endereço do array: %p\n", (void *) &digitos);
    printf("Conteúdo do array: %p\n", (void *) digitos);
    printf("Tamanho do array : %lu\n", sizeof(digitos)/sizeof(int));
    printf("\n");

    for (int i = 0; i < 10; i++)
    {
        printf("Endereço do array[%d]: %p\tConteúdo: %d\n",
               i, (void *) &digitos[i], digitos[i]);
    }

}
```

```
[abrantesasf@ideapad ~]$ ./array3
Endereço do array: 0x7ffdf1e24c80
Conteúdo do array: 0x7ffdf1e24c80
Tamanho do array : 10

Endereço do array[0]: 0x7ffdf1e24c80    Conteúdo: 1
Endereço do array[1]: 0x7ffdf1e24c84    Conteúdo: 2
Endereço do array[2]: 0x7ffdf1e24c88    Conteúdo: 3
Endereço do array[3]: 0x7ffdf1e24c8c    Conteúdo: 4
Endereço do array[4]: 0x7ffdf1e24c90    Conteúdo: 5
Endereço do array[5]: 0x7ffdf1e24c94    Conteúdo: 6
Endereço do array[6]: 0x7ffdf1e24c98    Conteúdo: 7
Endereço do array[7]: 0x7ffdf1e24c9c    Conteúdo: 8
Endereço do array[8]: 0x7ffdf1e24ca0    Conteúdo: 9
Endereço do array[9]: 0x7ffdf1e24ca4    Conteúdo: 0
```

```c
#include <cs50.h>
#include <stdio.h>

int somar(int a[])
{
    int tamanho = (int) (sizeof(a) / sizeof(int));

    int soma = 0;

    for (int i = 0; i < tamanho; i++)
    {
        soma += a[i];
    }

    return soma;
}

int main (void)
{
    int digitos[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
    printf("A soma dos dígitos é: %d\n", somar(digitos));
}
```

```
[abrantesasf@ideapad ~]$ ./array4
A soma dos dígitos é: 3
```

```c
#include <stdio.h>

int somar(int a[])
{
    int tamanho = (int) (sizeof(a) / sizeof(int));

    int soma = 0;

    for (int i = 0; i < tamanho; i++)
    {
        soma += a[i];
    }

    printf("Endereço do \"array\" a: %p\n", (void *) &a);
    printf("Conteúdo do \"array\" a: %p\n", (void *) a);
    return soma;
}

int main (void)
{
    int digitos[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
    printf("Endereço do array digitos: %p\n", (void *) &digitos);
    somar(digitos);
}
```

```
[abrantesasf@ideapad ~]$ ./array5
Endereço do array digitos: 0x7ffd8f7ac420
Endereço do "array" a: 0x7ffd8f7ac3f8
Conteúdo do "array" a: 0x7ffd8f7ac420
```

```c
#include <stdio.h>

int somar(int a[], int tamanho)
{
    int soma = 0;

    for (int i = 0; i < tamanho; i++)
    {
        soma += a[i];
    }

    return soma;
}

int main (void)
{
    int digitos[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
    printf("A soma dos dígitos é de: %i\n",
           somar(digitos, 10));
}
```

Em C **é impossível determinar o tamanho de um array passado como argumento**, pois o que é passado é um PONTEIRO para o array! Sempre que passarmos um "array", **também temos que passar o tamanho**!

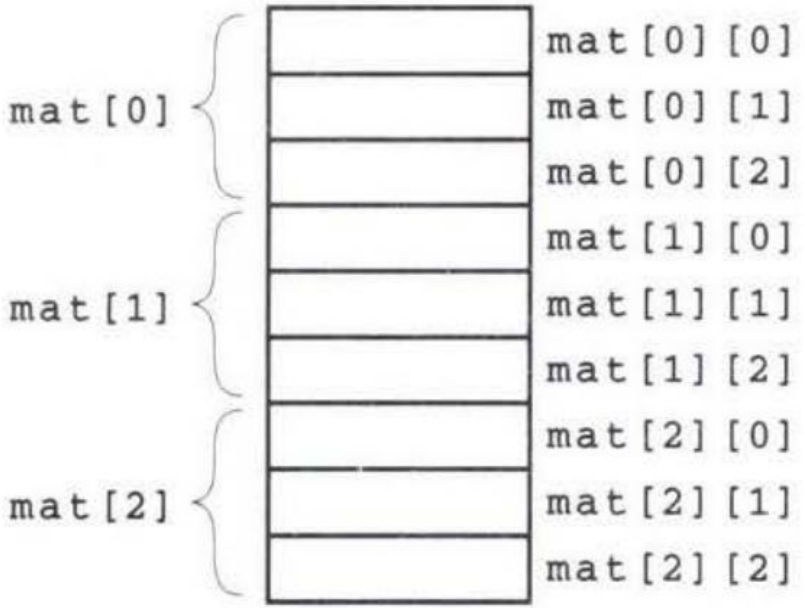
Arrays multidimensionais

- Os elementos de um array podem ser de qualquer tipo, incluindo outros arrays. Arrays de arrays são chamados de arrays multidimensionais.
- O mais comum é o array 2D, chamado de matriz.
- Os arrays podem ter dimensões:
  - completamente definidas
  - sem definição total das dimensões (**só a PRIMEIRA pode ficar em aberto**)

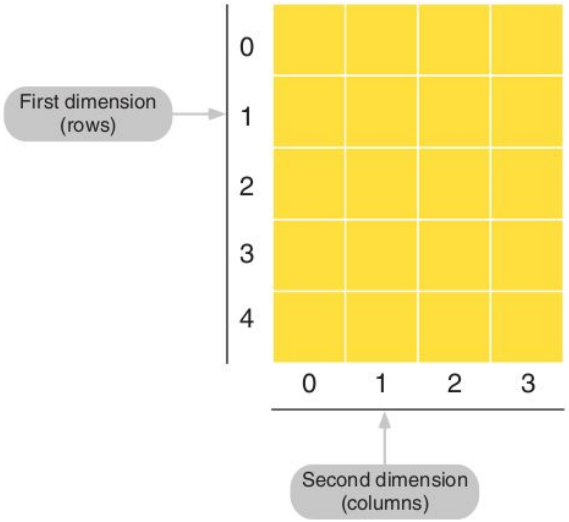
```c
#include <stdio.h>

int main (void)
{
    int array1d_1[10];              // 10 elementos
    // int array1d_2[];             // ERRO!!!
    int array1d_2[] = {1, 2, 3};    // 3 elementos

    int array2d_1[5][5];            // 25 elementos
    //int array2d_2[][5];           // Erro!!!
    int array2d_2[][3] = {{1, 2, 3}, {4, 5, 6}};  // 6 elementos
    int array2d_3[][3] = {{1, 2, 3},
                          {4, 5, 6}};             // 6 elementos
    //int array2d_4[][] = {{1, 2}, {3, 4}};       // ERRO!!!!

    int array3d_1[3][3][3];         // 27 elementos
    //int array3d_2[][2][2];        // ERRO!!!!
    int array3d_2[][2][2] = {{{1, 2}, {3, 4}},
                             {{5, 6}, {7, 8}}};   // 8 elementos
    //int array3d_3[][][3];                       // ERRO!!!!
    //int array3d_4[][][];                        // ERRO!!!!

}
```

Arrays multidimensionais: conceito x memória

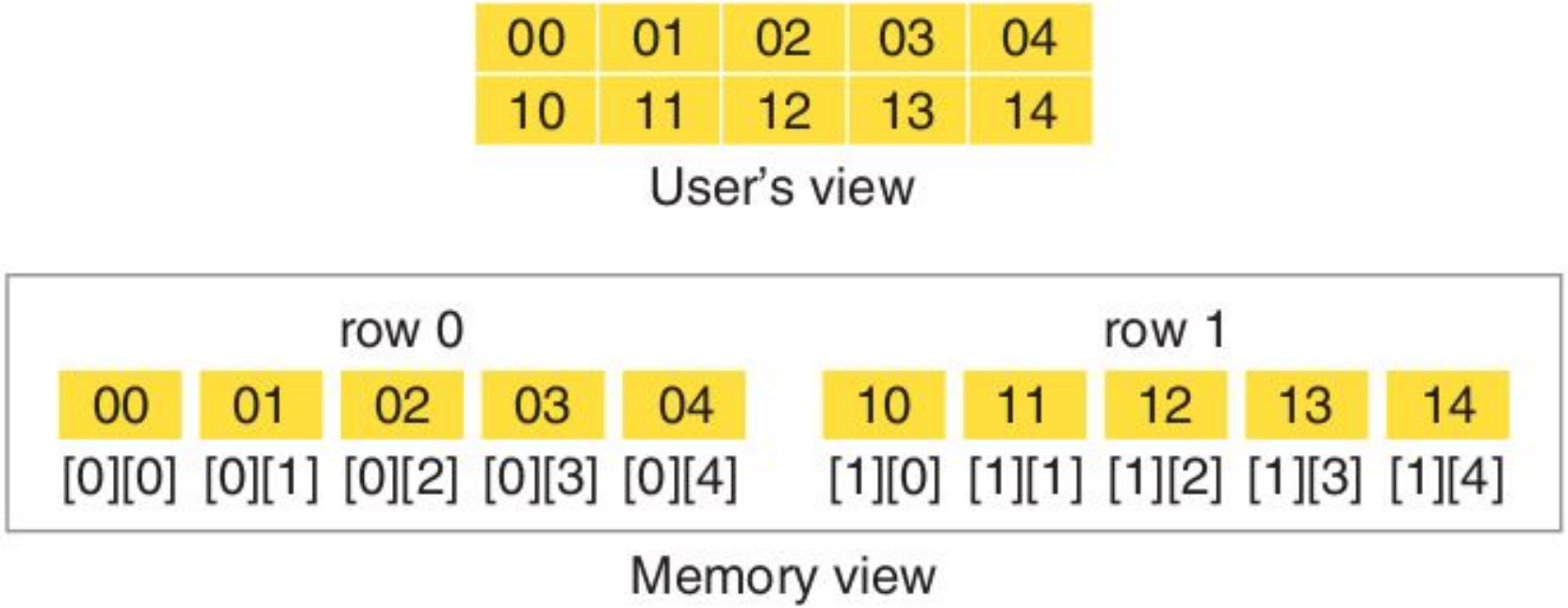
mat[0][0]	mat[0][1]	mat[0][2]
mat[1][0]	mat[1][1]	mat[1][2]
mat[2][0]	mat[2][1]	mat[2][2]



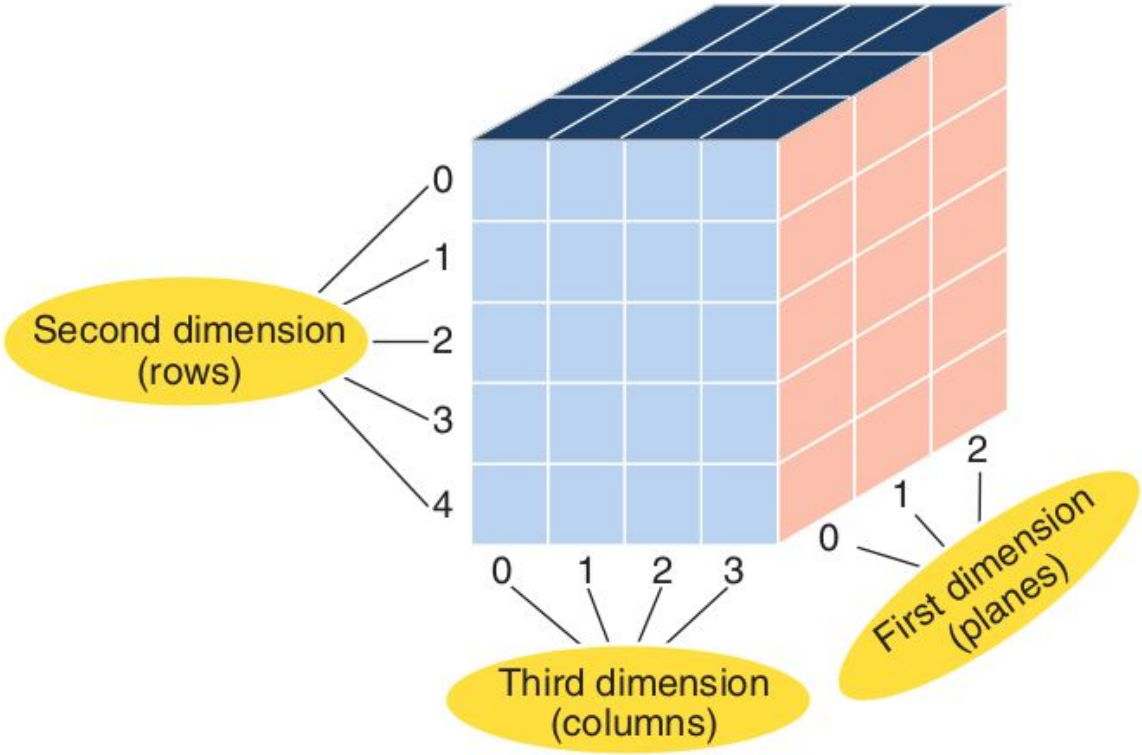
Arrays multidimensionais: conceito x memória



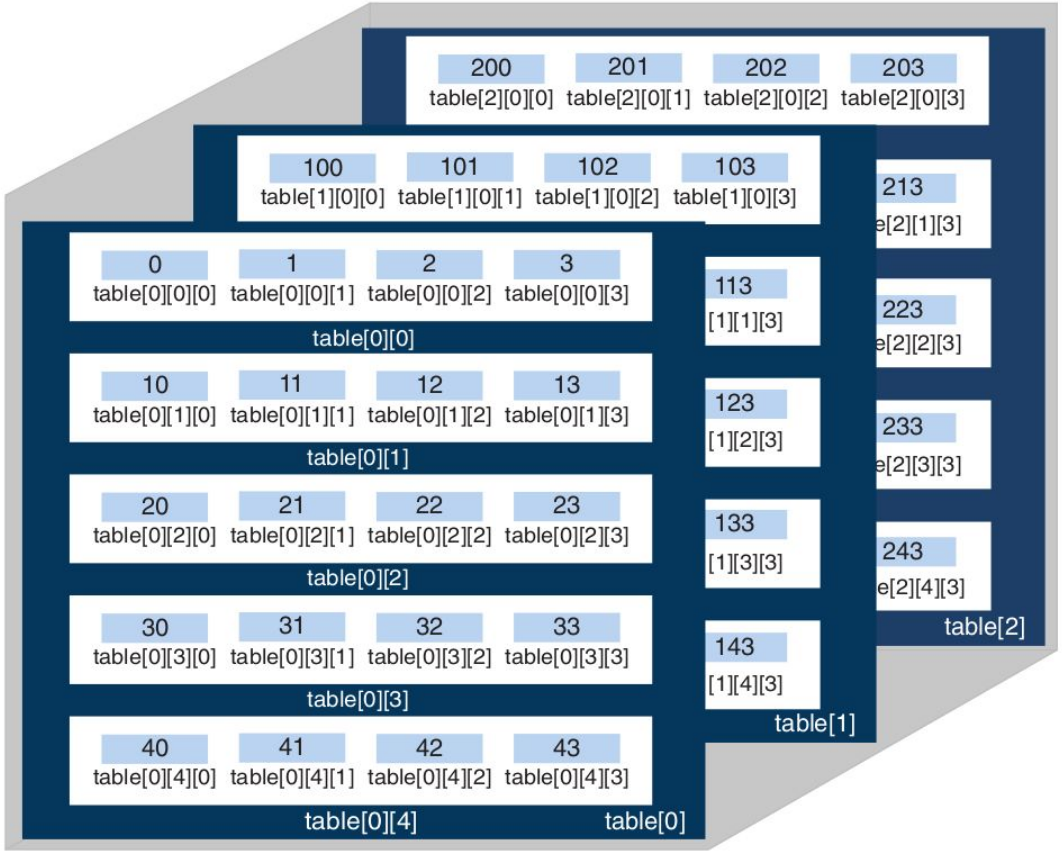
Arrays multidimensionais: conceito x memória



Arrays multidimensionais: conceito x memória



Arrays multidimensionais: conceito x memória



Arrays multidimensionais: conceito x memória

```c
#include <stdio.h>

int main (void)
{
    // Conceitualmente, um array de 2D é
    // entendido como uma matriz:
    int array2d[4][5] =
        {
            { 1,  2,  3,  4,  5},
            { 6,  7,  8,  9, 10},
            {11, 12, 13, 14, 15},
            {16, 17, 18, 19, 20}
        };

    // Internamente, C entende como um
    // array de 4 elementos, cada elemento
    // sendo um array de 5 elementos. E o
    // armazenamento é LINEAR na memória:
    for (int l = 0; l < 4; l++)
    {
        for (int c = 0; c < 5; c++)
        {
            printf("elem[%d][%d]\t%p\t%d\n",
                   l, c, (void *) &array2d[l][c], array2d[l][c]);
        }
    }
}
```

```
[abrantesasf@ideapad ~]$ ./array8
elem[0][0]      0x7fff295a88a0  1
elem[0][1]      0x7fff295a88a4  2
elem[0][2]      0x7fff295a88a8  3
elem[0][3]      0x7fff295a88ac  4
elem[0][4]      0x7fff295a88b0  5
elem[1][0]      0x7fff295a88b4  6
elem[1][1]      0x7fff295a88b8  7
elem[1][2]      0x7fff295a88bc  8
elem[1][3]      0x7fff295a88c0  9
elem[1][4]      0x7fff295a88c4  10
elem[2][0]      0x7fff295a88c8  11
elem[2][1]      0x7fff295a88cc  12
elem[2][2]      0x7fff295a88d0  13
elem[2][3]      0x7fff295a88d4  14
elem[2][4]      0x7fff295a88d8  15
elem[3][0]      0x7fff295a88dc  16
elem[3][1]      0x7fff295a88e0  17
elem[3][2]      0x7fff295a88e4  18
elem[3][3]      0x7fff295a88e8  19
elem[3][4]      0x7fff295a88ec  20
```

```c
#include <stdio.h>

int main (void)
{
    // Conceitualmente, um array de 3D é
    // entendido como um cubo, um array onde
    // cada elemento é um array de 2D:
    int array2d[2][3][4] =
        { // array 3D:
            { // 1º array 2D:
                {1,  2,  3,  4}, // 1ª linha e 4 colunas
                {5,  6 , 7,  8}, // 2ª linha e 4 colunas
                {9, 10, 11, 12}  // 3º linha e 4 colunas
            },
            { // 2º array 2D:
                {13, 14, 15, 16}, // 1ª linha e 4 colunas
                {17, 18, 19, 20}, // 2ª linha e 4 colunas
                {21, 22, 23, 24}  // 3ª linha e 4 colunas
            }
        };

    // Internamente, C entende como um
    // array de 2 elementos, cada elemento
    // sendo um array 2D de 3x4 elementos.
    // O armazenamento é LINEAR na memória:
    for (int a = 0; a < 2; a++)
    {
        for (int l = 0; l < 3; l++)
        {
            for (int c = 0; c < 4; c++)
            {
                printf("elem[%d][%d][%d]\t%p\t%d\n",
                       a, l, c, (void *) &array2d[a][l][c], array2d[a][l][c]);
            }
        }
    }
}
```

```
[abrantesasf@ideapad ~]$ ./array9
elem[0][0][0]   0x7ffe5bf77480  1
elem[0][0][1]   0x7ffe5bf77484  2
elem[0][0][2]   0x7ffe5bf77488  3
elem[0][0][3]   0x7ffe5bf7748c  4
elem[0][1][0]   0x7ffe5bf77490  5
elem[0][1][1]   0x7ffe5bf77494  6
elem[0][1][2]   0x7ffe5bf77498  7
elem[0][1][3]   0x7ffe5bf7749c  8
elem[0][2][0]   0x7ffe5bf774a0  9
elem[0][2][1]   0x7ffe5bf774a4  10
elem[0][2][2]   0x7ffe5bf774a8  11
elem[0][2][3]   0x7ffe5bf774ac  12
elem[1][0][0]   0x7ffe5bf774b0  13
elem[1][0][1]   0x7ffe5bf774b4  14
elem[1][0][2]   0x7ffe5bf774b8  15
elem[1][0][3]   0x7ffe5bf774bc  16
elem[1][1][0]   0x7ffe5bf774c0  17
elem[1][1][1]   0x7ffe5bf774c4  18
elem[1][1][2]   0x7ffe5bf774c8  19
elem[1][1][3]   0x7ffe5bf774cc  20
elem[1][2][0]   0x7ffe5bf774d0  21
elem[1][2][1]   0x7ffe5bf774d4  22
elem[1][2][2]   0x7ffe5bf774d8  23
elem[1][2][3]   0x7ffe5bf774dc  24
```

```c
#include <cs50.h>
#include <stdio.h>

double media(int a[], int tam)
{
    double soma = 0.0;
    for (int t = 0; t < tam; t++)
    {
        soma += (double) a[t];
    }
    return soma / (double) tam;
}

int main (void)
{
    int n = get_int("Informe o tamanho do array: ");
    int numeros[n];

    for (int i = 0; i < n; i++)
    {
        numeros[i] = get_int("Informe o %iº elemento: ", i + 1);
    }

    printf("A média dos números é: %f\n", media(numeros, n));
}
```

Já vimos que em C é impossível determinar o tamanho de um array passado como argumento, pois o que é passado é um PONTEIRO para o array! Sempre que passarmos um "array", também temos que passar o tamanho!

```c
#include <cs50.h>
#include <stdio.h>

double media(...)
{
    ...
}

int main (void)
{
    int notas[2][3] =
        {
            {7, 8, 5},
            {3, 9, 8}
        };

    printf("A média das notas é: %f\n", media(...));
}
```

E se o array tiver mais de 1 dimensão??? Lembre-se que temos que especificar todas exceto, talvez, a primeira. O que fazer????

```c
#include <cs50.h>
#include <stdio.h>

double media(int a[2][3], int l, int c)
{
    double soma = 0.0;
    for (int i = 0; i < l; i++)
    {
        for (int j = 0; j < c; j++)
            soma += (double) a[i][j];
    }
    return soma / (double) (l * c);
}

int main (void)
{
    int notas[2][3] =
        {
            {7, 8, 5},
            {3, 9, 8}
        };

    printf("A média das notas é: %f\n", media(notas, 2, 3));
}
```

Se soubermos as dimensões, podemos informar as dimensões do array no parâmetro, e depois passar a quantidade de linhas e colunas.

```c
#include <cs50.h>
#include <stdio.h>

double media(int a[][3], int l, int c)
{
    double soma = 0.0;
    for (int i = 0; i < l; i++)
    {
        for (int j = 0; j < c; j++)
            soma += (double) a[i][j];
    }
    return soma / (double) (l * c);
}

int main (void)
{
    int notas[2][3] =
        {
            {7, 8, 5},
            {3, 9, 8}
        };

    printf("A média das notas é: %f\n", media(notas, 2, 3));
}
```

Pode-se omitir a 1ª dimensão (todas as outras devem ser informadas), mas isso não é uma boa prática. Como norma de boa prática, acostume-se a sempre informar as dimensões dos arrays nos parâmetros!

```c
#include <cs50.h>
#include <stdio.h>

double media(...)
{
    ...
}

int main (void)
{
    int linhas = get_int("Informe o número de linhas: ");
    int colunas = get_int("Informe o número de colunas: ");
    int notas[linhas][colunas];

    for (int i = 0; i < linhas; i++)
    {
        for (int j = 0; j < colunas; j++)
            notas[i][j] = get_int("Informe a nota[%d][%d]: ", i, j);
    }

    printf("A média das notas é: %f\n", media(...));
}
```

E se for um variable-length array, cujo tamanho só é conhecido em runtime???

Como escrever os parâmetros????

```c
double media(int a[l][c], int l, int c)
{
    ...
}
```

ERRADO!

```c
double media(int l, int c, int a[l][c])
{
    ...
}
```

CERTO!

E se for um variable-length array, cujo tamanho só é conhecido em runtime???

Devemos utilizar variáveis nas dimensões, e informar essas variáveis ANTES do array! O motivo é que nesse caso o compilador processa os parâmetros da esquerda para a direita.

```c
#include <cs50.h>
#include <stdio.h>

double media(int l, int c, int a[l][c])
{
    double soma = 0.0;
    for (int i = 0; i < l; i++)
        for (int j = 0; j < c; j++)
            soma += (double) a[i][j];
    return soma / (double) (l * c);
}


int main (void)
{
    int linhas = get_int("Informe o número de linhas: ");
    int colunas = get_int("Informe o número de colunas: ");
    int notas[linhas][colunas];

    for (int i = 0; i < linhas; i++)
    {
        for (int j = 0; j < colunas; j++)
            notas[i][j] = get_int("Informe a nota[%d][%d]: ", i, j);
    }

    printf("A média das notas é: %f\n", media(linhas, colunas, notas));
}
```

Informe as dimensões em parâmetros localizados ANTES das dimensões dos arrays, nos casos de variable-length arrays.

Arrays e Ponteiros

- Em C, para você entender completamente os arrays, você precisa entender os ponteiros; e para você entender completamente os ponteiros, você precisa entender os arrays. Não é possível aprender um e ignorar o outro.
  - Arrays são implementados como ponteiros (lembra de alguns slides atrás?)
  - Operações em ponteiros só fazem sentido se considerarmos sua relação com um array
  - São CONCEITOS DIFERENTES, mas relacionados!
    - Não caia no erro de dizer que ponteiros e arrays são a mesma coisa!

Arrays e Ponteiros

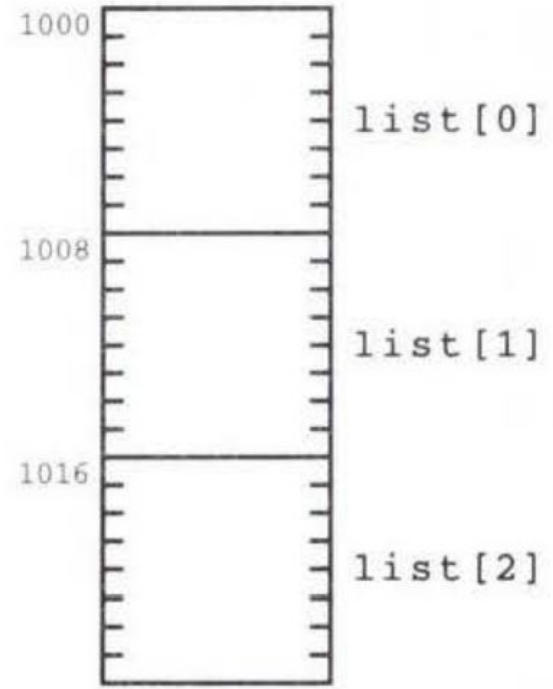
- Todo elemento é um lvalue e tem um endereço que pode ser obtido com:
  - `&list[1]` = endereço do elemento 1
  - `&list[i]` = endereço do i-ésimo elemento

- Para calcular o endereço correto de um elemento no array, o compilador pega o **endereço base** do array e adiciona o valor de `i` multiplicado pelo **tamanho do elemento** no array (cálculo do deslocamento):

  (end. base) + (i x tamanho)

  1000 + 2 x 8 = 1016

```c
#include <stdio.h>

int main (void)
{
    double list[3];
}
```



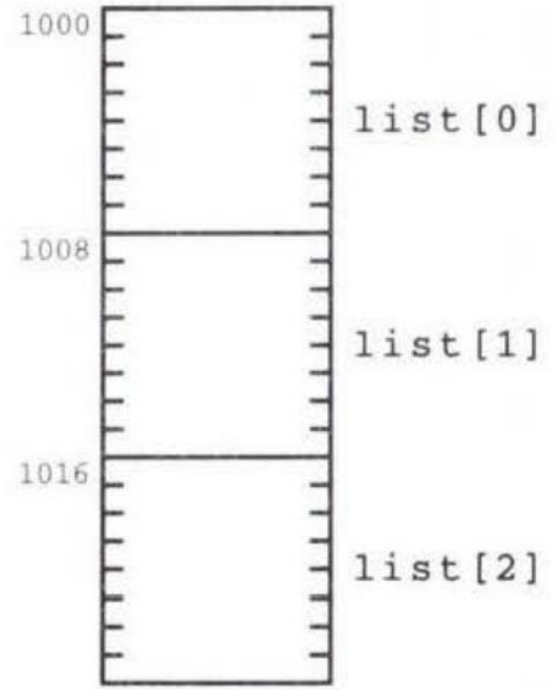
Aritmética de Ponteiros

- Realizar adição ou subtração com ponteiros é dita aritmética de ponteiros. A regra é simples:
  - **Se `p` é um ponteiro para o elemento inicial de um array `arr`, e `k` é um inteiro, a seguinte identidade é válida:**

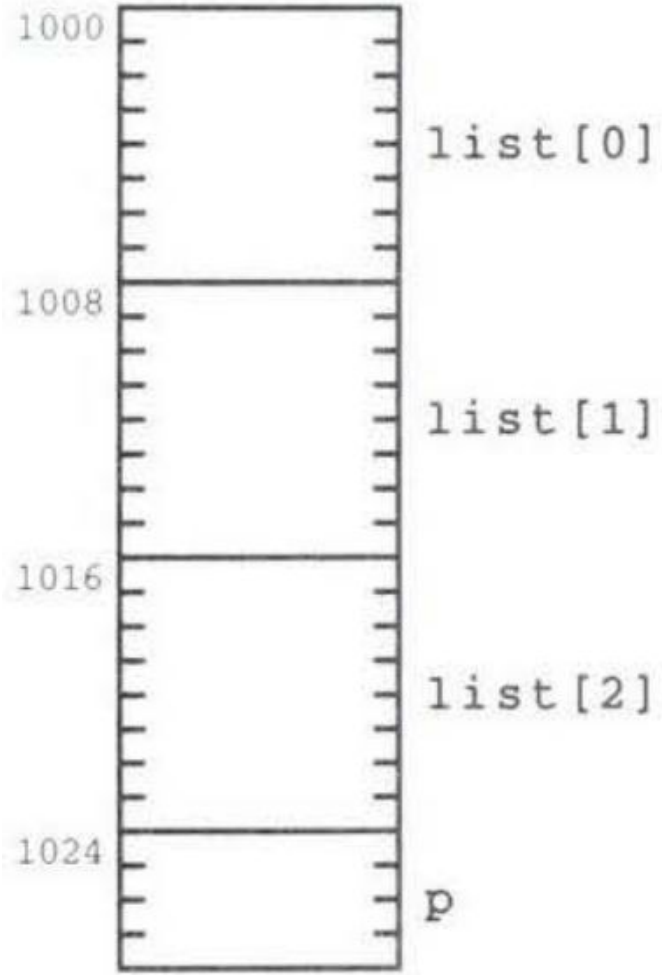
    `p + k` é definido como `&arr[k]`

  - Ou seja: se você adiciona um inteiro `k` ao valor de um ponteiro, o resultado é o **ENDEREÇO** do elemento do array no índice `k`, para um **array que se inicia no endereço original do ponteiro**.

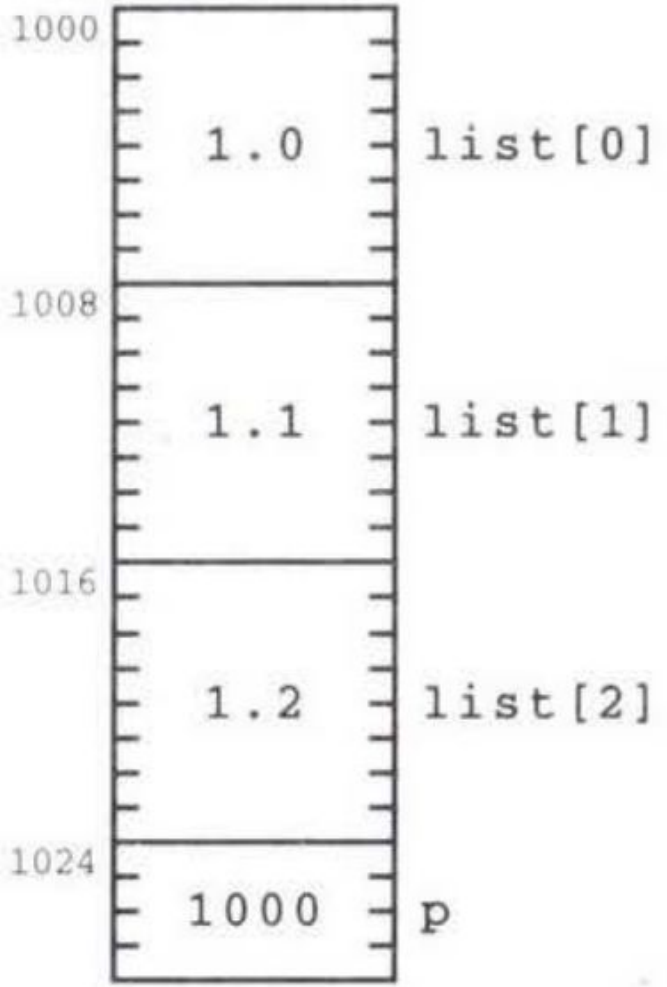
```c
#include <stdio.h>

int main (void)
{
    double list[3];
}
```



```c
#include <stdio.h>

int main (void)
{
    double list[3];
    double *p;
}
```



```c
#include <stdio.h>

int main (void)
{
    double list[3];
    double *p;

    list[0] = 1.0;
    list[1] = 1.1;
    list[2] = 1.2;

    p = &list[0];
}
```



Aritmética de Ponteiros

- As únicas operações de ponteiros com números inteiros são definidas como a adição e a subtração.
  - Operações de multiplicação, divisão ou módulo de inteiros com ponteiros não são definidas!

- Não é possível somar dois ponteiros:

  p1 + p2 = ERRO!!

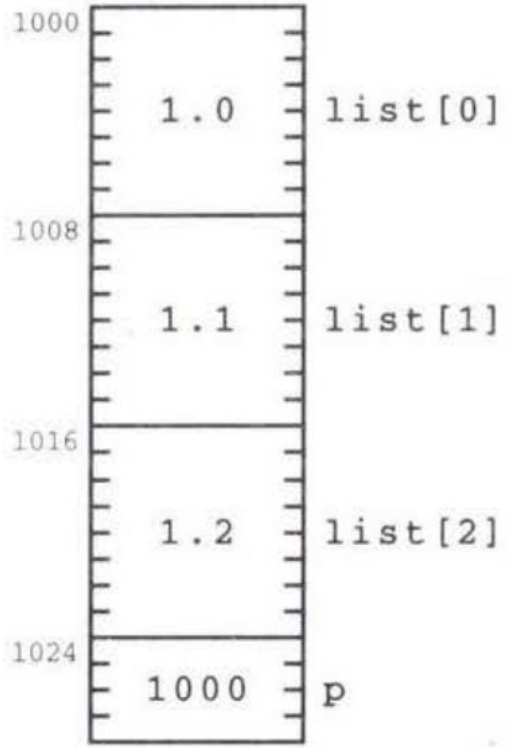
- É possível subtrair 2 ponteiros:

  p2 - p1 = n.º de elementos ENTRE esses ponteiros

  (sem contar o elemento apontado por p2)

```c
#include <stdio.h>

int main (void)
{
    double list[3];
    double *p;

    list[0] = 1.0;
    list[1] = 1.1;
    list[2] = 1.2;

    p = &list[0];   // p aponta para list[0]

    printf("Valor em list[1]: %f\n", *(p + 1));

    p = p + 1;      // p aponta para list[1]

    printf("Valor em list[0]: %f\n", *(p - 1));
    printf("Valor em list[2]: %f\n", *(p + 1));

    //                                                  list[2]   list[0]
    printf("Qtd. de elementos entre p2 e p1: %ld\n", (p + 1) - (p - 1));
}
```

```
[abrantesasf@ideapad ~]$ ./array15
Valor em list[1]: 1.100000
Valor em list[0]: 1.000000
Valor em list[2]: 1.200000
Qtd. de elementos entre p2 e p1: 2
```



Incremento e decremento de ponteiros: `*p++`

- O que a expressão `*p++` está fazendo?
  - Desreferenciando o ponteiro p e incrementando o valor que estava apontado por p?
  - Desreferenciando o ponteiro p e incrementando o valor do ponteiro p?
  - Incrementando o valor do ponteiro p e desreferenciando esse novo endereço?
  - Alguma outra coisa?

- Muito difícil de saber à primeira vista.

- Só a tabela de precedência e associatividade nos salva!

- **Tenha MUITO CUIDADO ao usar incremento e decremento de ponteiros**! Só use se souber o que está fazendo e se realmente for necessário!

Incremento e decremento de ponteiros: *p++

[ ]	Array Index	ary[i]	N	Left-Right	16
f(...)	Function Call	doIt(x, y)	Y		
.	Direct Member Selection	str.mem	N		
->	Indirect Member Selection	ptr->mem	N		
++ --	Postfix Increment • Decrement	a++	Y		
++ --	Prefix Increment • Decrement	++a	Y	Right-Left	15
sizeof	Size in Bytes	sizeof(int)	N		
~	Ones Complement	-a	N		
!	Not	!a	N		
+ -	Plus • Minus	+a	N		
&	Address	&a	N		
*	Dereference/Indirection	*ptr	N		

Incremento e decremento de ponteiros: *p++

```c
#include <stdio.h>

int main (void)
{
    int lista[3] = {10, 20, 30};
    int *pl;

    // Por que a expressão abaixo funciona
    // mesmo sem usar o operador &?
    pl = lista;

    // Localizações:
    for (int i = 0; i < 3; i++)
        printf("lista[%d]: %d em %p\n", i, lista[i], (void *) &lista[i]);
    printf("\n");

    // Ponteiro pl:
    printf("pl em: %p\n", (void *) pl);

    // Operação misteriosa:
    printf("Retorno de *pl++: %d\n", *pl++);

    // E o ponteiro pl agora?
    printf("pl em: %p\n", (void *) pl);
}
```

*pl++ está retornando o endereço atual de pl, que é então desreferenciado. Depois o próprio ponteiro pl é incrementado!

*pl++ = *pl
pl++

```
[abrantesasf@ideapad ~]$ ./array16
lista[0]: 10 em 0x7fff2da840cc
lista[1]: 20 em 0x7fff2da840d0
lista[2]: 30 em 0x7fff2da840d4

pl em: 0x7fff2da840cc
Retorno de *pl++: 10
pl em: 0x7fff2da840d0
```

Incremento e decremento de ponteiros: *p++ = *(p++)

```c
#include <stdio.h>

int main (void)
{
    int lista[3] = {10, 20, 30};
    int *pl;

    // Por que a expressão abaixo funciona
    // mesmo sem usar o operador &?
    pl = lista;

    // Localizações:
    for (int i = 0; i < 3; i++)
        printf("lista[%d]: %d em %p\n", i, lista[i], (void *) &lista[i]);
    printf("\n");

    // Ponteiro pl:
    printf("pl em: %p\n", (void *) pl);

    // Operação misteriosa:
    printf("Retorno de *(pl++): %d\n", *(pl++));

    // E o ponteiro pl agora?
    printf("pl em: %p\n", (void *) pl);
}
```

CUIDADO!!! *(pl++) continua retornando o endereço atual de pl, que é então desreferenciado. Depois o próprio ponteiro pl é incrementado!

*(pl++) = *pl
pl++

```
[abrantesasf@ideapad ~]$ ./array17
lista[0]: 10 em 0x7fff4e193b8c
lista[1]: 20 em 0x7fff4e193b90
lista[2]: 30 em 0x7fff4e193b94

pl em: 0x7fff4e193b8c
Retorno de *(pl++): 10
pl em: 0x7fff4e193b90
```

Incremento e decremento de ponteiros: *++p

```c
#include <stdio.h>

int main (void)
{
    int lista[3] = {10, 20, 30};
    int *pl;

    // Por que a expressão abaixo funciona
    // mesmo sem usar o operador &?
    pl = lista;

    // Localizações:
    for (int i = 0; i < 3; i++)
        printf("lista[%d]: %d em %p\n", i, lista[i], (void *) &lista[i]);
    printf("\n");

    // Ponteiro pl:
    printf("pl em: %p\n", (void *) pl);

    // Operação misteriosa:
    printf("Retorno de *++pl: %d\n", *++pl);

    // E o ponteiro pl agora?
    printf("pl em: %p\n", (void *) pl);
}
```

*++pl está incrementando o endereço atual de pl e, depois do incremento, o ponteiro é desreferenciado!

*++pl = ++pl
*pl

```
[abrantesasf@ideapad ~]$ ./array18
lista[0]: 10 em 0x7ffc13fea79c
lista[1]: 20 em 0x7ffc13fea7a0
lista[2]: 30 em 0x7ffc13fea7a4

pl em: 0x7ffc13fea79c
Retorno de *++pl: 20
pl em: 0x7ffc13fea7a0
```

Incremento e decremento de ponteiros: `*(p + 1)`

```c
#include <stdio.h>

int main (void)
{
    int lista[3] = {10, 20, 30};
    int *pl;

    // Por que a expressão abaixo funciona
    // mesmo sem usar o operador &?
    pl = lista;

    // Localizações:
    for (int i = 0; i < 3; i++)
        printf("lista[%d]: %d em %p\n", i, lista[i], (void *) &lista[i]);
    printf("\n");

    // Ponteiro pl:
    printf("pl em: %p\n", (void *) pl);

    // Operação misteriosa:
    printf("Retorno de *(pl + 1): %d\n", *(pl + 1));

    // E o ponteiro pl agora?
    printf("pl em: %p\n", (void *) pl);
}
```

*(pl + 1) está retornando o próximo lvalue do array, SEM ALTERAR O PONTEIRO, e é esse lvalue que está sendo desreferenciado. Aqui **não há perigo de modificar o ponteiro acidentalmente**!

*(pl + 1) = &lista[1]

```
[abrantesasf@ideapad ~]$ ./array19
lista[0]: 10 em 0x7ffc301c9afc
lista[1]: 20 em 0x7ffc301c9b00
lista[2]: 30 em 0x7ffc301c9b04

pl em: 0x7ffc301c9afc
Retorno de *(pl + 1): 20
pl em: 0x7ffc301c9afc
```

Relação entre arrays e ponteiros

- **O nome de um array é tratado como sinônimo de um ponteiro para o elemento inicial do array**! A regra aqui é a seguinte:
  - **Para qualquer array `arr`, a seguinte igualdade sempre é válida:**

`arr` **é definido como** **`&arr[0]`**

- Por isso essas declarações são idênticas internamente:

```
double somar (double a[], int tam) {...};

double somar (double *a,  int tam) {...};
```

- Como regra, utilize a declaração que melhor reflete o uso pretendido.

Relação entre arrays e ponteiros

- Uma diferença CRUCIAL entre arrays e ponteiros ocorre quando as variáveis são declaradas (não quando são passadas como argumentos): **alocação de memória**!

```
int lista[5];
int *p;
```

reserva 20 bytes consecutivos na memória

reserva 8 bytes consecutivos na memória

- Ao declarar um array temos todo o espaço de armazenamento reservado para os números; ao declarar um ponteiro temos apenas o espaço de armazenamento para 1 único endereço de memória.

Relação entre arrays e ponteiros

- É vantajoso usar ponteiros para arrays? Depende… Isso aqui não tem benefícios práticos:

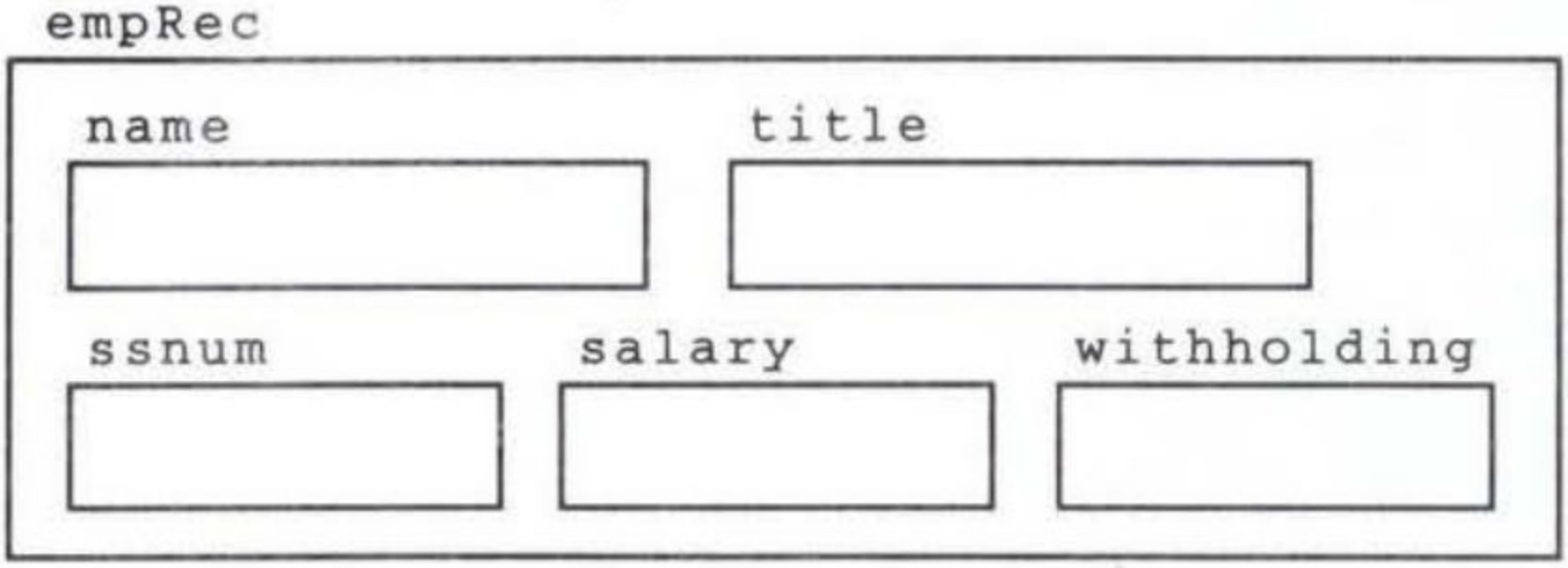
```
int lista[5] = {1, 2, 3, 4, 5};
int *pl = lista;
```

(você pode usar o próprio nome “lista” como um ponteiro)

- A vantagem real de usar um ponteiro como array vem do fato de que você pode inicializar um ponteiro para uma nova área de memória que não tinha sido alocada previamente, o que permite que você crie novos arrays enquanto seu programa está em execução. Isso é a chamada “**Alocação Dinâmica**” (em breve).

Registros

- São tipos de dados que armazenam vários campos de dados que formam um todo coerente, um “objeto” coerente. Por exemplo: você precisa manter dados de funcionários:



Registros

- Em C, o principal tipo para criar registros é o **`struct`**:

```c
#include <cs50.h>
#include <stdio.h>

struct st_funcionario
{
    string nome;
    string titulo;
    string cpf;
    double salario;
    int    idade;
};

typedef struct st_funcionario funcionario_t;

int main (void)
{
    struct st_funcionario abrantes;

    funcionario_t flavia;
}
```

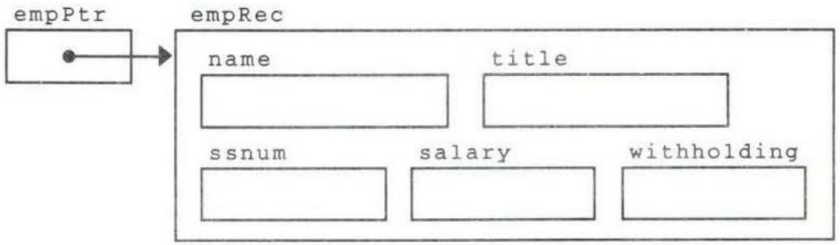
Seleção e inicialização de registros

- Para fazer a seleção de um registro em particular (um struct), basta usar seu nome.
- Para fazer a seleção de um campo em um registro em particular, temos de usar o operador “.” (ponto):
  - **`struct.campo`**
  - O operador ponto retorna um lvalue.
- Podemos iniciar um struct no momento de sua declaração.

```
struct st_funcionario abrantes;
abrantes.nome = "Abrantes Araújo Silva Filho";
abrantes.titulo = "Professor";
abrantes.cpf = "321654987-00";
abrantes.salario = 1000.0;
abrantes.idade = 50;

funcionario_t flavia;
flavia.nome = "Flávia Cardoso";
flavia.titulo = "Dra.";
flavia.cpf = "123.456.789-00";
flavia.salario = 1000.0;
flavia.idade = 50;
```

Ponteiros para registros

- São muito usados pois:
  - structs são passados por valor
  - permitem manipular grandes quantidades de registros




```c
#include <cs50.h>
#include <stdio.h>

struct st_funcionario
{
    string nome;
    string titulo;
    string cpf;
    double salario;
    int    idade;
};

typedef struct st_funcionario funcionario_t;

int main (void)
{
    funcionario_t abrantes;
    abrantes.nome = "Abrantes Araújo Silva Filho";
    abrantes.titulo = "Professor";
    abrantes.cpf = "321654987-00";
    abrantes.salario = 1000.0;
    abrantes.idade = 50;

    funcionario_t *pabrantes;
    pabrantes = &abrantes;
}
```

Ponteiros para registros: CUIDADO com a seleção!

- Apesar de parecer certo, isso aqui é **ERRADO**:

```
*pabrantes.salario
```

O operador ponto “.” tem maior precedência do que o operador * e, portanto essa expressão é interpretada como a coisa sem sentido abaixo:

```
*(pabrantes.salario)
```

Isso **é errado** pois você está tentando pegar o campo salário de um ponteiro e tentando desreferenciar essa coisa sem sentido!

Ponteiros para registros: CUIDADO com a seleção!

- O jeito **CERTO** de usar ponteiro para acessar uma struct é:

```
(*pabrantes).salario
```

  Primeiro desreferenciamos o ponteiro, retornando a struct. Depois acessamos o campo salário nessa struct.

- Existe um operador para facilitar o uso de ponteiros para structs, que faz o desreferenciamento e a seleção automaticamente: **->**

```
pabrantes->salario
```

  Atenção: o operador seta só pode ser usado para ponteiros para structs! Se usar uma struct diretamente, o operador ponto deve ser utilizado.

Alocação dinâmica

- Variáveis globais são criadas e vinculadas em tempo de compilação, nos segmentos DATA ou BSS da memória. Esse tipo de alocação é dita:
  - **Alocação Estática**

- Variáveis locais são criadas e vinculadas no momento da chamada da função e ficam nos STACK FRAMES. Esse tipo de alocação é dita:
  - **Alocação Automática**

- Um terceiro tipo de alocação permite que novos blocos de memória possam ser utilizados e depois liberados quando precisarmos. Essas variáveis ficam no segmento HEAP e a alocação é dita:
  - **Alocação dinâmica**

Alocação dinâmica

- Existem diversas funções (em stdlib.h) para alocação dinâmica. Por enquanto vamos ficar com a mais básica: **`malloc`**

- A função malloc aloca um bloco de memória (na heap) com um determinado tamanho em bytes, retornando um ponteiro para void (void *).
  - isso é necessário pois malloc não sabe o tipo de dado que será apontado pelo ponteiro, então ela retorna um ponteiro “genérico”
  - lembre-se de que os ponteiros têm tipos específicos em C

- Entendendo o uso de malloc: alocar espaço para 20 milhões de doubles:

```
double *pd = (double *) malloc(sizeof(double) * 20000000);
```

Alocação dinâmica

- É **OBRIGATÓRIO verificar se malloc conseguiu alocar o bloco de memória, ANTES de tentar fazer qualquer coisa com esse bloco**. Se malloc não conseguiu alocar a memória, ela retorna um ponteiro para NULL:

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

int main (void)
{
    double *pd = (double *) malloc(sizeof(double) * 20000000);
    if (pd == NULL)
    {
        printf("ERRO na alocação de memória!\n");
        return 1;
    }

    free(pd);
}
```

Alocação dinâmica

- É **OBRIGATÓRIO liberar a memória após o uso, ANTES do programa terminar**. Se você não liberar a memória, causará **memory leak**:

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

int main (void)
{
    double *pd = (double *) malloc(sizeof(double) * 20000000);
    if (pd == NULL)
    {
        printf("ERRO na alocação de memória!\n");
        return 1;
    }

    free(pd);
}
```

Alocação dinâmica

- Se você não precisa mais de um bloco de memória e liberou esse bloco, mas o programa continuará, É **IMPORTANTE ATRIBUIR NULL AO PONTEIRO**, para evitar **dangling pointer**:

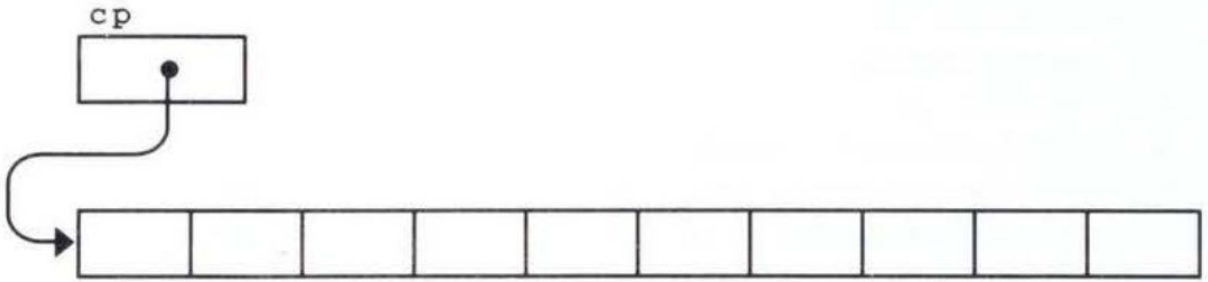
```c
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
    double *pd = (double *) malloc(sizeof(double) * 20000000);
    if (pd == NULL)
    {
        printf("ERRO na alocação de memória!\n");
        return 1;
    }

    free(pd);

    pd = NULL;

    // O programa continua aqui:
    {...}
}
```

Arrays Dinâmicos

- Entenda que:

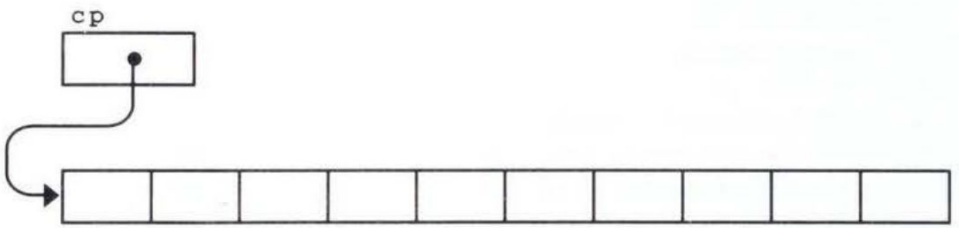
```
char *cp = (char *) malloc(sizeof(char) * 10);
if (cp == NULL)
{
    printf("ERRO na alocação de memoria!\n");
    return 1;
}
```

Cria a seguinte situação na memória: (cp no stack; bloco anônimo na heap):



Arrays Dinâmicos

- Por causa da relação entre ponteiros e arrays em C, esse bloco de memória apontado por cp funciona EXATAMENTE como se fosse um array de 10 elementos (10 caracteres).
- Arrays alocados na HEAP e referenciados por um ponteiro são chamados arrays dinâmicos. São muito importantes:
  - Permite criar novos arrays em tempo de execução
  - Economia de espaço: a memória não é alocada até que você chama malloc
  - Você pode ajustar o tamanho de um array dinâmico de acordo com a quantidade de dados, no momento da criação do array dinâmico ou depois (novo array e cópia)
- Você usa os arrays dinâmicos exatamente da mesma forma como os arrays normais: com aritmética de ponteiro ou expressão de seleção.



```c
#include <cs50.h>
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
    int n = get_int("Quantos números? ");

    int *pnotas = (int *) malloc(sizeof(int) * n);
    if (pnotas == NULL)
    {
        printf("ERRO na alocação de memoria!\n");
        return 1;
    }

    // Usando seleção com []
    for (int i = 0; i < n; i++)
        pnotas[i] = get_int("Informe o %dº inteiro: ", i + 1);

    // Usando aritmética de ponteiro
    for (int i = 0; i < n; i++)
        printf("%dº inteiro: %d\n", i + 1, *(pnotas + i));

    free(pnotas);
    pnotas = NULL;
}
```

```
[abrantesasf@ideapad ~]$ ./malloc2
Quantos números? 3
Informe o 1º inteiro: 10
Informe o 2º inteiro: 20
Informe o 3º inteiro: 30
1º inteiro: 10
2º inteiro: 20
3º inteiro: 30
```

Registros Dinâmicos

- Da mesma maneira que os arrays, também é possível termos registros dinâmicos, ou seja, registros alocados na HEAP.

```c
#include <cs50.h>
#include <stdio.h>
#include <stdlib.h>

struct st_pessoa
{
    string nome;
    int    idade;
    char   sexo;
};

typedef struct st_pessoa pessoa_t;

int main (void)
{
    pessoa_t *pabrantes = (pessoa_t *) malloc(sizeof(pessoa_t));
    if (pabrantes == NULL)
    {
        printf("ERRO na alocação de memória!\n");
        return 1;
    }

    pabrantes->nome = "Abrantes Araújo Silva Filho";
    pabrantes->idade = 50;
    (*pabrantes).sexo = 'M';

    free(pabrantes);
    pabrantes = NULL;
}
```

Registros Dinâmicos

- Em algumas situações é importante definir tipos que são ponteiros para registros de forma direta. Isso será visto posteriormente, quando aprendermos sobre interfaces e tipos abstratos de dados.

```
struct st_pessoa
{
    string nome;
    int    idade;
    char   sexo;
};

typedef struct st_pessoa *pessoa_t;
```