



UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Campus de São José do Rio Preto

Estrutura de Dados I

Aula Introdutória

Prof. Wallace Casaca

UNESP - IBILCE
Departamento de Ciências de Computação e
Estatística (DCCE)

Apresentação do professor

- **Docente:** Prof. Wallace Correa de Oliveira Casaca
- **Webpage:** www.spcovid.net.br
- **Grupo de pesquisa:** www.viser.com.br
- **E-mail:** wallace.casaca@unesp.br



Apresentação da disciplina

Curso: Estrutura de Dados I (ED I)

Carga horária semestral: 90 h

- 60 h teóricas;
- 30 h outras.

Aulas: Quarta-feira (a partir das 14:00 – Sala 2C)

Ementa

- Estruturas de Dados Não-Lineares: Árvores Binárias, Árvores de Busca e Árvores Balanceadas.
- Filas de Prioridade.
- Grafos: Conceito e Implementação.



Bibliografia

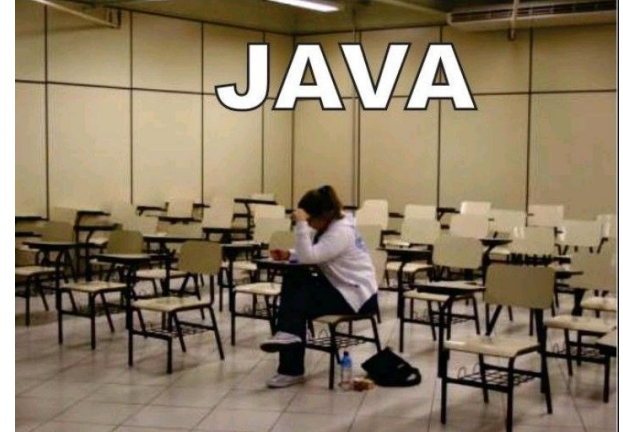
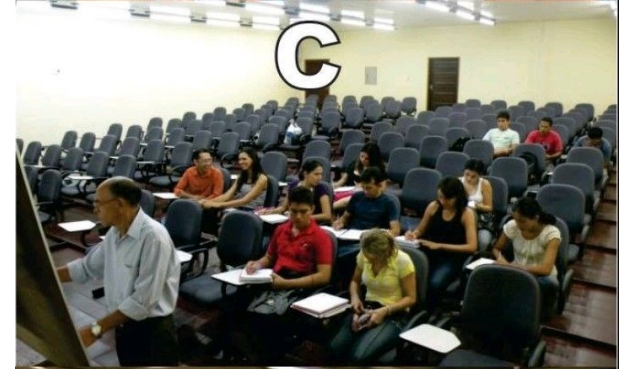
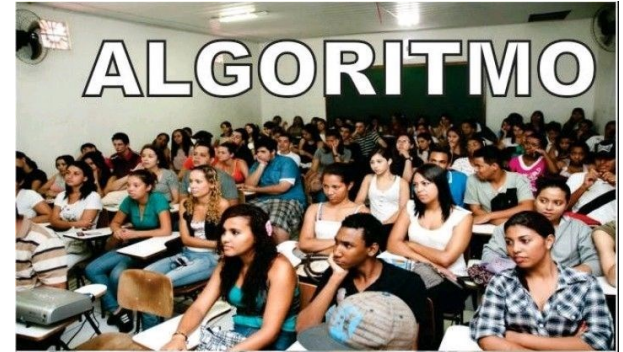
1. SCHILDT, H. **C Completo e Total**, 3.ed. São Paulo: Makron Books do Brasil, 1997.
2. CORMEN, T. H. et al. **Algoritmos: Teoria e Prática**, Editora Campus, 3ª Ed., 2012.
3. SZWARCFITER, J. L., MARKENZON, L.; **Estruturas de Dados e seus Algoritmos**, LTC, 3a Ed., 2010.
4. CELLES, W; CERQUEIRA R.; RANGEL, J.L. **Introdução à Estruturas de Dados: Com Técnicas de Programação em C**, Editora Campos, 2004.
5. FORBELLONE, A. L. V.; EBERSPACHER, H. F. **Lógica de Programação: A Construção de Algoritmos e Estruturas de Dados**, 3. ed., 2005.
6. ASCENCIO, A. F. G.; CAMPOS, E. A. V. C. **Fundamentos da Programação de Computadores: Algoritmos, Pascal, C, C++ e Java**, Pearson Prentice Hall, 2012.
7. **Apostilas e Material técnico on-line** (disponíveis em páginas web especializadas sobre o tema – ex: STL: <http://www.cplusplus.com/reference/> e <http://en.cppreference.com/w/c>).

Metodologia de ensino

- Aulas expositivas, acompanhadas de listas de exercícios e implementação de trabalhos de programação abordando os conceitos vistos em sala de aula.
- A carga horária indicada como OUTRA contabiliza trabalhos extra-classe que serão conduzidos pelos alunos.
- Apresentação de seminários.
- Aulas práticas de implementação.

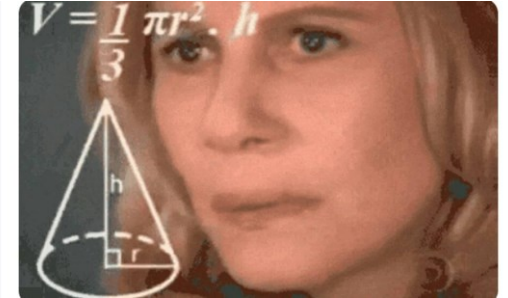
Plataformas computacionais

- Linguagem adotada: C
- Compiladores/IDEs (sugestões):
 - Plataformas on-line (sites)
 - Dev-C++
 - Code::Blocks
 - Visual C++ (Windows)
 - GCC (Linux/MAC)



Sistemática de avaliação

- Avaliações teóricas (**P1** e **P2/Projeto**).
- Trabalhos extraclasse + atividades em sala (**NC**).
- **Projeto de disciplina** (no lugar da **P2**) + seminários.



$$\text{Média Final} = 0.8 \times (\mathbf{P1} + \mathbf{P2})/2 + 0.2 \times \text{média}(\mathbf{NC})$$

Média Final \geq 5.0: **Aprovado**

Média Final $<$ 5.0: **Recuperação**

IMPORTANTE

- Evite cópias e plágios !!!
 - Nota = 0



EF – Exame final da disciplina

- **Exame final:** Avaliação versando todo o conteúdo da disciplina.
- **Quem pode fazer:** alunos que obtiveram média no semestre abaixo de 5.0 e que tenham ao menos 70% de frequência nas aulas.
- Como fica a média no final (MF):

$$\text{MF} = (\text{Média_Semestre} + \text{EF})/2$$

- Caso $\text{MF} \geq 5.0 \implies$ Aprovado ;)
- Caso $\text{MF} < 5.0 \implies$ Reprovado :(

Datas

Datas

- $P_1 = 03/05/2023$
- $P_2 = 21/06/2023$
- EXAME FINAL = 28/06/2023
- Entrega de trabalhos: **A DEFINIR**

Atendimento

- Terça-feira (manhã); ou
e-mail: wallace.casaca@unesp.br



Site virtual da disciplina de ED I

Link: <http://tiny.cc/ed2023>

Mural

Atividades

Pessoas

Notas

 Personalizar

Estrutura de Dados I



Meet



Gerar link



Escreva um aviso para sua turma

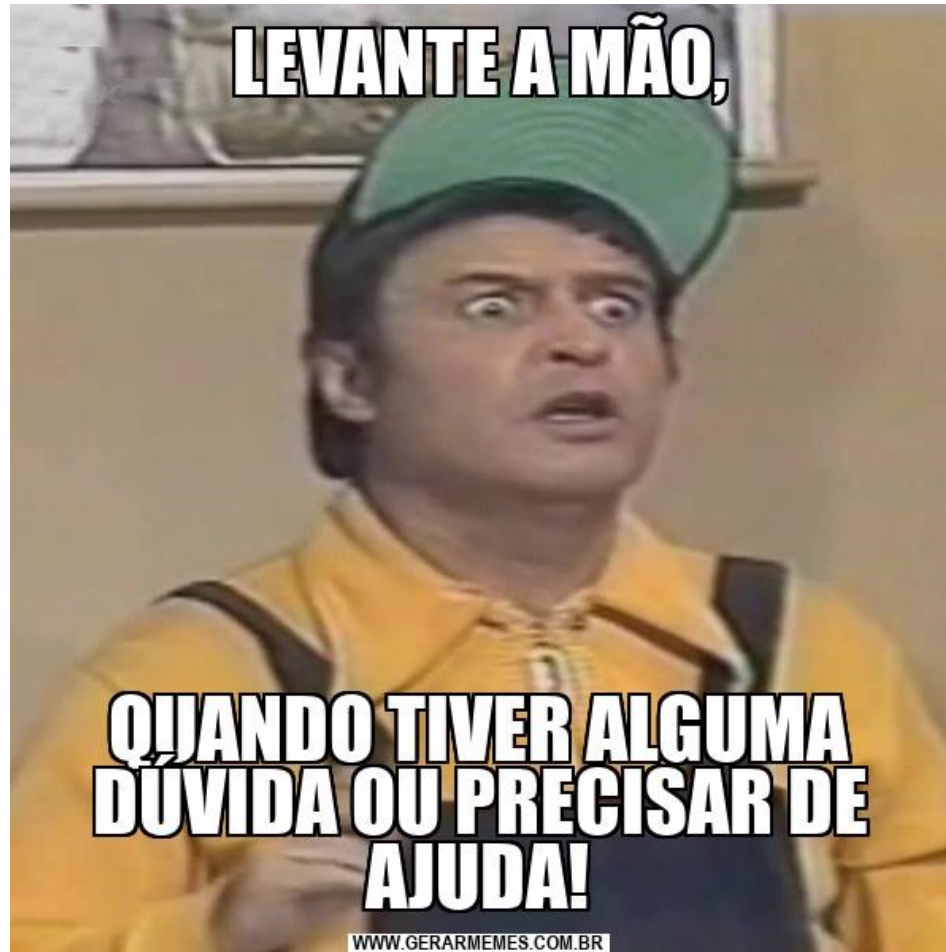


Código da turma:



gvj4zie 

Perguntas ?

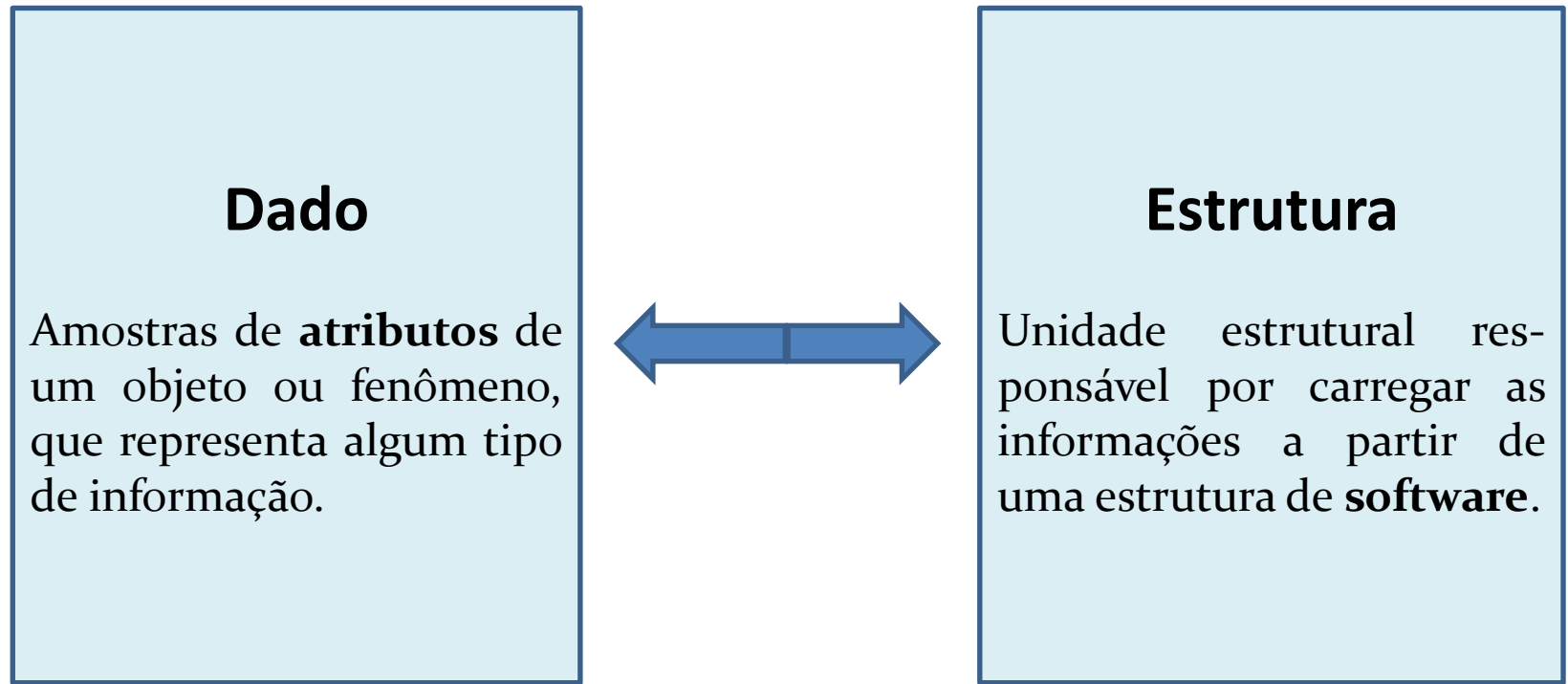


Estrutura de dados

- Pra que serve?
- Exemplos mais comuns:
 - Vetores (arrays)
 - Listas
 - Pilhas
 - Filas
 - Árvores
 - Grafos

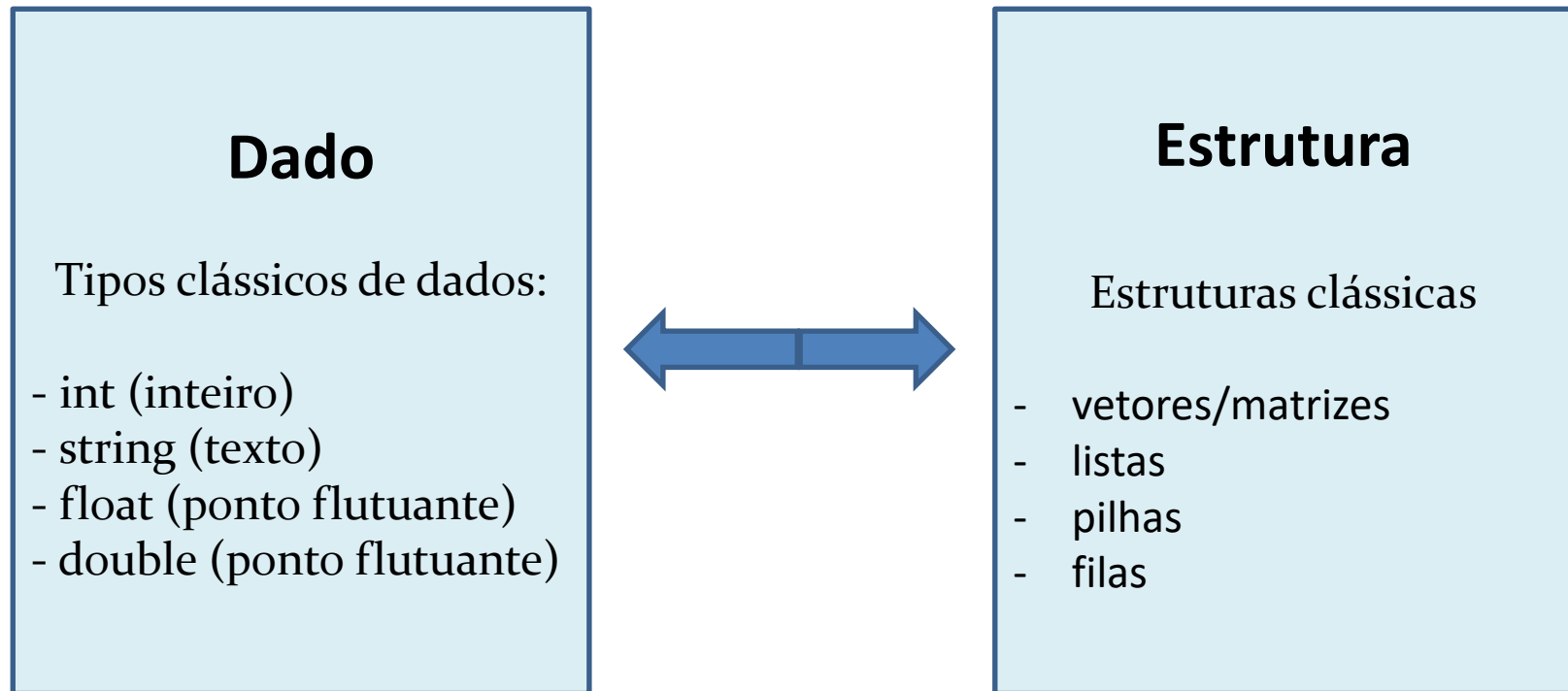
Estrutura de dados

- Uma estrutura de dados pode ser entendida por meio **duas entidades básicas**:



Estrutura de dados

- Exemplos



Estruturas de dados básicas

- **Vetores (unidimensional e bidimensional):** É uma das estruturas de dados mais simples (e também uma das mais usadas na prática).
- **Características:**
 - Acesso aos elementos por meio de índices.
 - Possuem tamanho finito de componentes.
 - Carregam dados de um mesmo tipo.
 - São ordenados por meio de indexação.
 - Caso unidimensional: único índice.
 - Caso bidimensional: índice duplo.

Exemplo - vetores

- **Vetor unidimensional**

- `int vet[8];`

10	2	5	27	34	789	33	0
0	1	2	3	4	5	6	7

- `vet[0] = 10;`

- `vet[1] = 2;`

- `vet[2] = 5;`

...

- `vet[7] = 33;`

Exemplo - vetores

- Vetor bidimensional

- `int matriz[2][2];`

	0	1
0	10	2
1	34	50

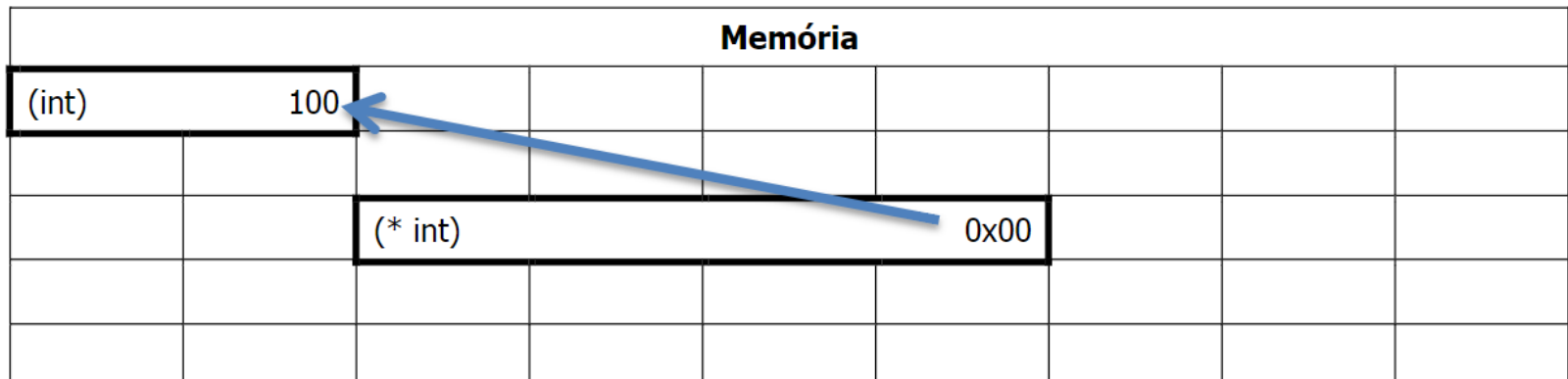
- `mat[0][0] = 10;`
 - `mat[0][1] = 2;`
 - `mat[1][0] = 34;`
 - `mat[1][1] = 50;`

Estrutura de dados e ponteiros

- Basicamente, um ponteiro é uma variável que contém **um endereço de memória**.
- Esse endereço é, em geral, **a posição de uma outra variável** na memória.
- O endereço “apontado” contém um valor específico.

Estrutura de dados e ponteiros

- **Ponteiros:** são variáveis que armazenam não um tipo de dado diretamente, mas sim o **endereço de memória** de onde um certo dado se encontra.



Por que usar ponteiros?



Por que usar ponteiros?

- Ponteiros fornecem formas adequadas para que **funções** possam modificar seus argumentos de modo mais eficiente.
- Usados para alocações dinâmicas !!!
- Usados para aumentar a performance dos códigos.
- Usados para criar estrutura de dados !!!

Ponteiros: declaração

- A declaração de um ponteiro consiste:
 - No tipo básico;
 - Uso do caractere `*` (para diferenciar entre ponteiro e uma variável convencional).
 - O nome da variável.
- **Sintaxe:** `tipo_basico *nome_variavel;`

Exemplos:

```
int *a; float* v; char *c;
```


Operadores de ponteiros

1) Operador: &

- Devolve um **endereço de memória**.
- O endereço é a posição interna da variável na memória.

Exemplo:

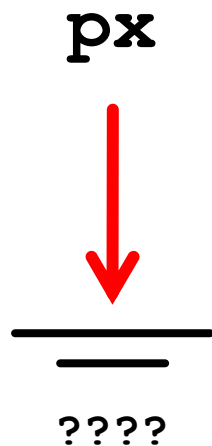
```
int x = 10, *px;
```

```
px = &x; //A variável px aponta para x.
```

Operador &

a) Declaração

```
int x, *px;
```



MEMÓRIA

Variável	Endereço
	1000
x	1002
	1004

Operador &

b) Atribuição

`px = &x;`

px



MEMÓRIA

Variável	Endereço
	1000
x	1002
	1004

O endereço gravado em **px**
é **1002**

Operadores de ponteiros

2) Operador: *

- Devolve o **valor da variável** localizada no endereço gravado pelo ponteiro.

Exemplo:

```
int x = 10, y, *px;
```

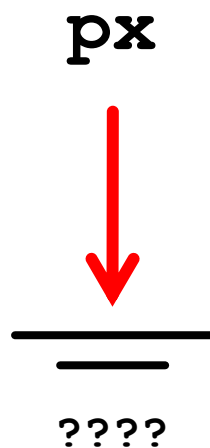
```
px = &x;    //px aponta para x
```

```
y = *px;    //y recebe o valor de x (pois  
             px aponta para x)
```

Operador *

a) Declaração

```
int x = 10, y, *px;
```



MEMÓRIA


Variável	Endereço
	1000
x = 10	1002
y	1004

Operador *

b) Atribuição

```
px = &x;  
y = *px;
```

MEMÓRIA

		Variável	Endereço
			1000
px <u> </u> <u> </u> ????		x = 10	1002
		y = 10	1004

Observações

- Ponteiros sempre devem apontar para o **tipo de dado correspondente**.
 - Exemplo: quando um ponteiro é declarado como do tipo **int**, ele assumirá que o endereço a ser armazenado será o de uma variável do tipo **int**.
- A linguagem C não informa isso ao programador !!

Atribuição de ponteiro

- Tal como acontece com variáveis convencionais, um ponteiro pode ser copiado através de um comando de atribuição.

Exemplo:

```
float a, *p1, *p2;
```

```
p1 = &a;
```

```
p2 = p1; //Passa o endereço gravado em p1 para p2
```


Inicialização de ponteiros

- **Convenção:**

- Um ponteiro que atualmente **não aponta para um local de memória válido** recebe o chamado valor nulo (NULL).
- Qualquer ponteiro que é considerado nulo, implica que ele não aponta para nada.

Ponteiros x NULL

Exemplo:

```
int *p = NULL;  
printf ("%d", *p) ;
```

- É muito comum o uso de “If’s” para verificar se um ponteiro possui o valor NULL, para só então ser manipulado.

Aritmética de ponteiros

Incremento

- Quando um ponteiro é incrementado, ele passa a apontar para a posição de memória **do próximo elemento do seu tipo base**.

Exemplo:

```
float x = 1.5, *px;  
px = &x;           //px aponta para x  
px++;              //Incrementa px
```

Aritmética dos ponteiros

a) Atribuição

`px = &x;`

px



MEMÓRIA

Variável	Endereço
x=1.5	1000
	1004
	1008

float = 4 bytes

Aritmética de ponteiros

b) Incremento

`px++;`

`px`

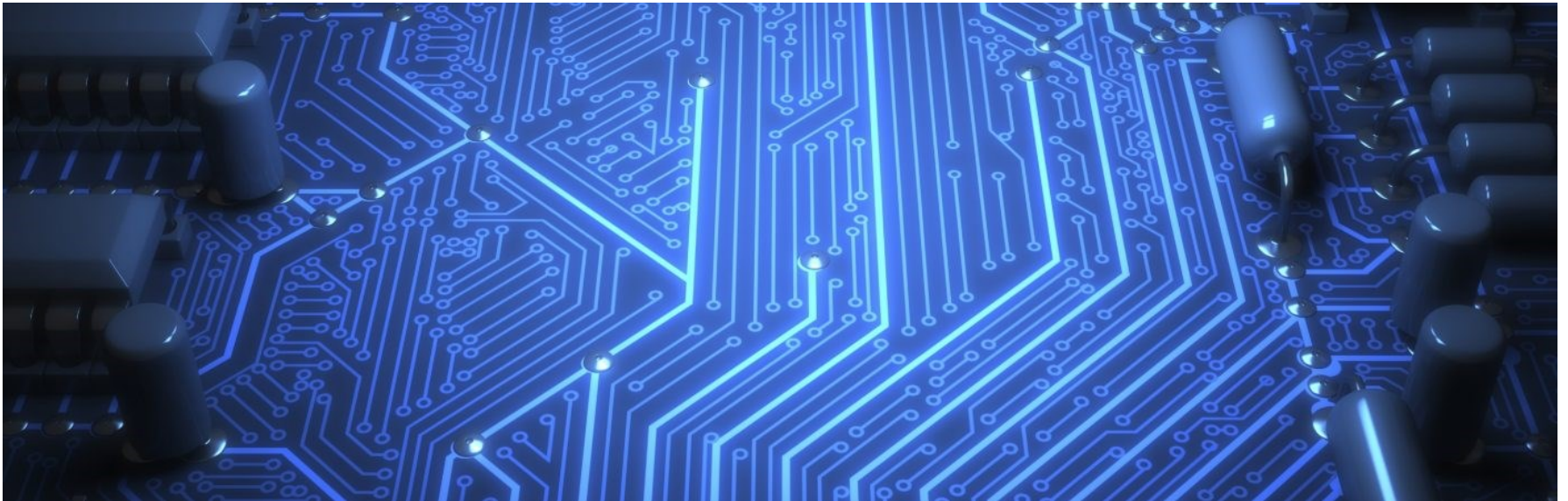
MEMÓRIA

Variável	Endereço
<code>x=1.5</code>	1000
	1008
	1016

float = 4 bytes

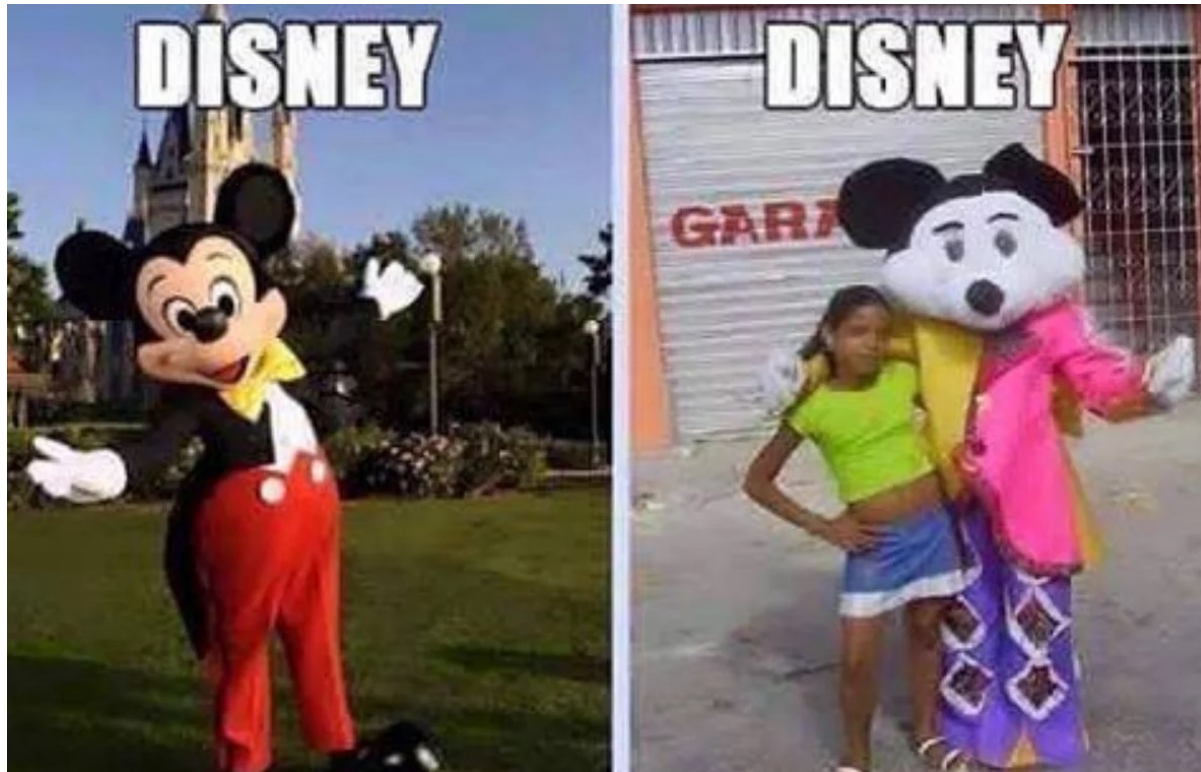
O ponteiro irá apontar para o próximo endereço de memória do tipo base no computador

TADs: Tipos Abstratos de Dados



Tipo de dado, tipo estruturado de dado e tipo abstrato de dado

- Termos similares, mas de significados diferentes !



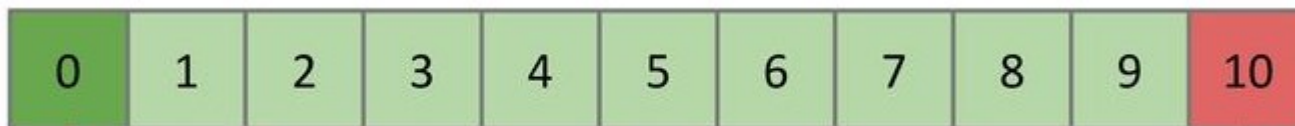
1) Tipo de dado

- Em linguagens de programação, o **tipo de dado** de uma variável é o **conjunto de valores** que ela pode assumir.
- Em geral, o tipo está associado à uma variável.
- **Exemplo:** após declarada, uma variável do tipo **inteira** poderá assumir valores inteiros negativos e positivos.

2) Tipo estruturado de dado

- É um caso de **estrutura de dados clássica**, cuja estrutura já está implementada na linguagem de programação.
- **Exemplos:**

Vetores, matrizes, *structs*.



3) Tipo abstrato de dado

- Tipos e estruturas de dados (EDs) são usados para acessar as informações via **operações apropriadas**.
- Do ponto de vista de quem implementa: é conveniente pensar nas EDs com **foco nas operações que elas suportam**, e não apenas da maneira como são implementadas.
- Uma estrutura de dados definida dessa forma é chamada de um **Tipo Abstrato de Dados (TAD)**.

Exemplo de TAD: fila

- Operações sobre **fila** (pessoas):
 - criação da estrutura;
 - inserção no fim;
 - remoção no começo.



Características de um TAD

- O usuário (“enquanto programador”) terá acesso a uma **descrição dos valores e das operações definidas no TAD**.
- De um modo geral, o programador **não precisa ter acesso à implementação interna do TAD**.
 - Idealmente: a implementação é “**invisível**” e **inacessível**.
 - **Exemplo:** pode-se criar uma lista de clientes e aplicar operações sobre ela (inserir novo cliente, consultar, ...), mas não é necessário saber como ela (a lista) foi representada/implementada internamente.

Exemplo

- O TAD estabelece o conceito da ED **desassociada** à sua implementação no computador.

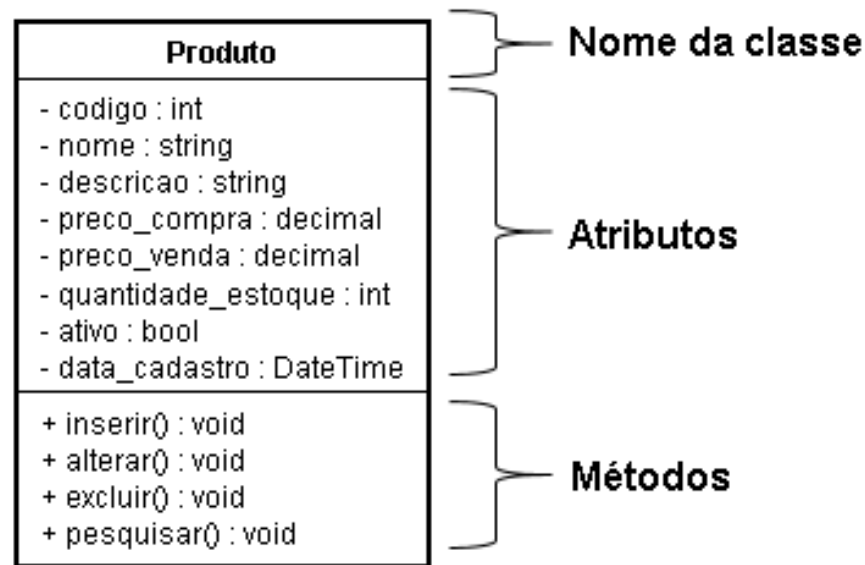
```
int main() {  
    //Exemplo de TAD: arquivos em C  
  
    //Declaração de um TAD 'arquivo'  
    FILE *file;  
  
    //Os dados da 'file' estão todos ocultos: somente  
    //podem ser acessados pelas FUNÇÕES (operações)  
    //de manipulação do tipo arquivo:  
    a) fopen ()  
    b) fclose ()  
    c) fputs ()  
    d) fgets ()  
    e) fclose ()  
    ....  
}
```

Implementação de um TAD

- Uma vez definido um TAD e suas operações, ele pode ser implementado em uma linguagem de programação.
- Isso significa que uma **Estrutura de Dados** pode ser vista como **uma implementação de um TAD**.
 - A implementação do TAD implica na **escolha de uma ED adequada para representá-lo**, que será **acessada via operações definidas no TAD**.
- **Na prática**: a ED é construída a partir dos **tipos básicos** (*integer*, *real*, *char*), ou ainda, a partir de um **tipos estruturados clássicos** (*structs*, *arrays*,...).

Analogia entre POO e implementação do TAD

- Em linguagens orientadas à objetos (C++, Java, ...): a implementação é feita por meio das **classes**.



- Em linguagens estruturadas (C, Pascal, ...): a implementação é feita pela definição de **tipos** e a implementação de **funções**.

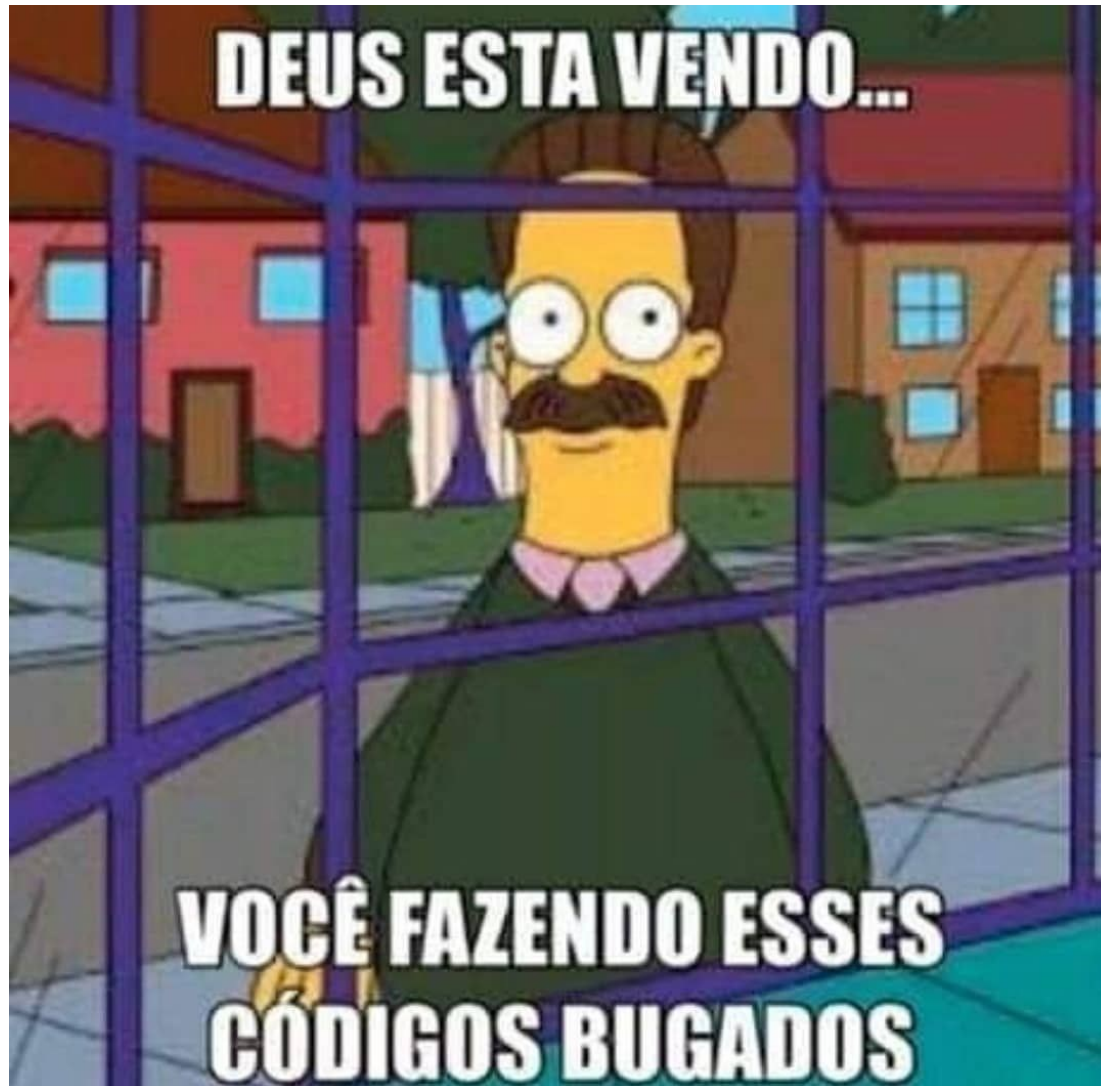
Implementação de um TAD na prática

- Para criar um TAD, empregamos a ideia de **modularização**:
 1. Um arquivo que contém a definição específica do TAD, isto é:
 - (I) **a representação das estruturas de dados.**
 - (II) **implementação das operações suportadas.**
 2. O outro arquivo deverá conter a **interface de acesso**, que apresenta as operações possíveis.
- **Vantagem:** outros programadores podem, por meio do arquivo de interface, usar o TAD sem conhecer detalhes representacionais e sem precisar acessar o módulo interno com as implementações.

Implementação de um TAD na prática

- **O que é feito na prática (convenção):** procura-se implementar TADs em arquivos separados no programa principal.
 - **nomedoTAD.h:** arquivo de cabeçalho, contendo:
 - A **declaração** das funções protótipos suportadas pelo TAD
 - **Os tipos de ponteiros** para acesso às TAD
 - **nomedoTAD.c:** arquivo contendo:
 - A **representação dos tipos de dados da TAD** (structs)
 - A **implementação das funções/operações** de acesso ao TAD.

Exemplo – TAD matriz



Exemplo (TAD em C): matriz.h

```
//TAD: matriz real m x n (m: linhas e n: colunas)
```

```
//Tipo Exportado
```

```
typedef struct matriz Matriz;
```

```
//Funções exportadas
```

```
//-----
```

```
//Função cria: aloca e retorna matriz m x n
```

```
Matriz *cria(int m, int n);
```


```
//Função libera: libera a memória de uma matriz
```

```
void libera(Matriz *mat);
```

```
//Função acessa: retorna o valor do elemento [i][j]
```

```
float acessa(Matriz *mat, int i, int j);
```

arquivo (de declaração): matriz.h



```
//Função atribui: atribui valor ao elemento [i][j]  
void atribui(Matriz* mat, int i, int j, float v);
```

```
//Função linhas: retorna nro de linhas da matriz  
int linhas(Matriz* mat);
```

```
//Função colunas: retorna nro de colunas da matriz  
int colunas(Matriz* mat);  
//-----
```

arquivo (de declaração): matriz.h

Observações

- Os tipos de dados e funções declaradas no **arquivo de cabeçalho** (**matriz.h**) são **exportados** para os módulos que incluem esse arquivo via `#include "matriz.h"`.
 - Serão visíveis para os “**clientes**” do TAD.
 - **Exemplo:** No arquivo “main.c” e arquivo com implementações das funções (“matriz.c”).
- **Em geral:** Tipos de dados e funções para fins de implementação interna do TAD não devem constar no arquivo de cabeçalho, apenas no arquivo de implementação (matriz.c).
 - **Nota:** matriz.c não é exportado ao “cliente”, isto é, são “inacessíveis” pra quem utiliza o TAD.

Exemplo (TAD em C): matriz.c

```
#include <stdlib.h> //Para usar malloc, free, exit ...  
#include <stdio.h>  //Para usar printf ,...  
#include "matriz.h" //Carrega o arquivo .h criado
```

```
//Implementação da struct matriz
```

```
struct matriz  
{  
    int lin;  
    int col;  
    float *v;  
};
```

```
//Implementação das funções do TAD matriz
```

```
//-----  
void libera(Matriz *mat)  
{  
    free(mat->v);  
    free(mat);  
}
```

Arquivo (de implementação): matriz.c

```
Matriz *cria(int m, int n)
{
    Matriz *mat = (Matriz*) malloc(sizeof(Matriz));

    if(mat == NULL)
    {
        printf("Memória insuficiente.\n");
        exit(1);
    }
    mat->lin = m;
    mat->col = n;
    mat->v = (float*) malloc(m*n*sizeof(float));
    return mat;
}
```

Arquivo (de implementação): matriz.c

```
float acessa(Matriz *mat, int i, int j)
{
    int k;
    if(i < 0 || i >= mat->lin || j < 0 || j >= mat->col)
    {
        printf("Acesso inválido!\n");
        exit(1);
    }
    k = i*mat->col+ j; //Conversao indice duplo em simples
    return mat->v[k];
}

void atribui(Matriz *mat, int i, int j, float v)
{
    int k;
    if(i < 0 || i >= mat->lin || j < 0 || j >= mat->col)
    {
        printf("Atribuição inválida!\n");
        exit(1);
    }
    k = i*mat->col + j; //Conversao (i,j) --> k (indice simples)
    mat->v[k] = v;
}
```

Arquivo (de implementação): matriz.c


```
int linhas(Matriz *mat)
{
    return mat->lin;
}
```

```
int colunas(Matriz *mat)
{
    return mat->col;
}
```

Arquivo (de implementação): matriz.c

Exemplo (TAD em C): main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "matriz.h"

int main(int argc, char *argv[])
{
    float a, b, c, d;
    Matriz *M;

    //Cria uma matriz 5x5
    M = cria(5,5);

    //Insere valores na matriz
    atribui(M,1,2,40);
    atribui(M,2,3,3);
    atribui(M,3,0,15);
    atribui(M,4,1,21);
```

Programa cliente: main.c

Exemplo (TAD em C): main.c

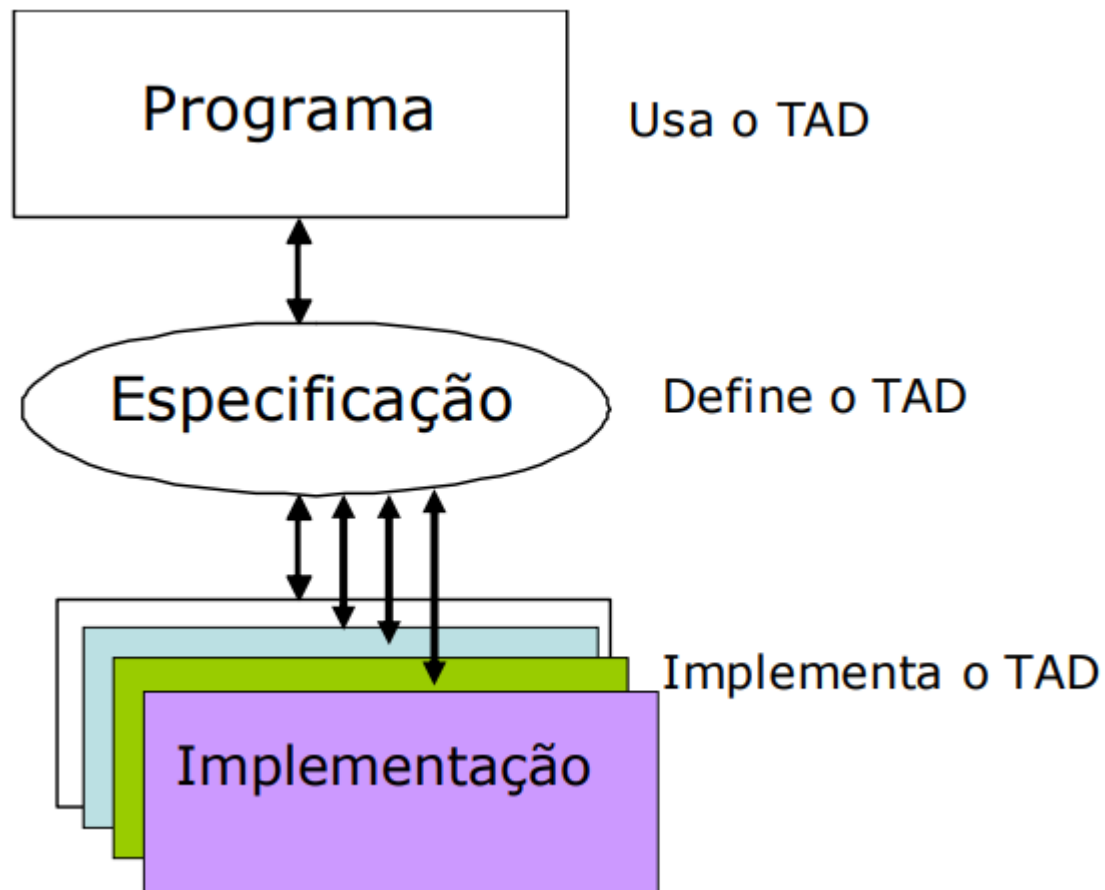
```
//Verifica se a inserção foi feita corretamente
a = acessa(M,1,2);
b = acessa(M,2,3);
c = acessa(M,3,0);
d = acessa(M,4,1);
printf("M[1][2]: %4.2f \n", a);
printf("M[2][3]: %4.2f \n", b);
printf("M[3][0]: %4.2f \n", c);
printf("M[4][1]: %4.2f \n", d);

system("PAUSE");
return 0;
}
```

Programa cliente: main.c

Em termos computacionais ...

- Quando usamos TADs, nossos sistemas ficam divididos em:



Vantagens do uso de TADs



Vantagens do uso de TADs

- **Reuso:** uma vez definido, implementado e testado, o TAD pode ser usado por diferentes programas.
- **Manutenção:** mudanças de implementação no TAD **não afetam o código fonte** das aplicações que o utilizam (decorrência do ocultamento da informação).
 - Os módulos do TAD são compilados separadamente.
 - Uma alteração força somente a recompilação do arquivo e uma nova linkedição do programa que acessa o TAD.
 - O programa em si não precisa ser recompilado.

É...essa semana foi difícil

Capitão, ainda é quarta

