

Estrutura de Dados I

Capítulo 8: Tipos Abstratos de Dados (TADs)

Introdução às Estruturas de Dados

- Estruturas de dados **armazenam os valores (dados) que estão sendo manipulados** por um algoritmo, de forma **organizada** e **otimizada**.
- O mesmo dado pode ser **armazenado por diferentes estruturas de dados**
- A **escolha de qual estrutura de dados utilizar é fundamental** pois:
 - Para os mesmos dados algumas estruturas podem ocupar mais ou menos espaço
 - Para os mesmos dados algumas estruturas podem permitir algoritmos mais ou menos eficientes em termos de complexidade de tempo e espaço
 - Se a estrutura correta foi escolhida, os algoritmos tendem a ser simples. **É mais fácil mudar a estrutura de dados do que criar um algoritmo mais complexo!**
 - É mais fácil projetar o programa ao redor de estruturas de dados corretas.
- Problema do estudante:
 - **Confusão de nomenclatura**, discordância entre autores, sopa de letrinhas, etc...

Exemplos de dificuldades iniciais do estudante

- Uma **variável** é uma estrutura de dados?
- Um **tipo de dado** é uma estrutura de dados?
- Um **array** é uma estrutura de dados?
- Um **tipo abstrato de dado** é uma estrutura de dados?
- Qual a diferença entre um **tipo de dado** e um **tipo abstrato de dado**?
- Uma **pilha** é uma estrutura de dados?
- Um **dicionário** é uma estrutura de dados? Ou um tipo abstrato de dado?
- **Heap** é uma estrutura de dados? Ou tipo de dado? Ou tipo abstrato de dado?
- Uma estrutura de dados tem **interface**?
- Uma **árvore** é um tipo de dado, um tipo abstrato de dado ou uma estrutura?
- O que é uma **estrutura de dados composta ou híbrida**?
- ...

Tipo de Dado

01111111	01000101	01001100	01000110	00000010
00000000	00000000	00000000	00000000	00000000
00000010	00000000	00111110	00000000	00000001
10110000	00000101	01000000	00000000	00000000
01000000	00000000	00000000	00000000	00000000
11010000	00010011	00000000	00000000	00000000
00000000	00000000	00000000	00000000	01000000
00001001	00000000	01000000	00000000	00100100

- Ao final de tudo, os dados que utilizamos em nossos programas são apenas bits 0 e 1.

Se nossos programas tivessem que processar direta e exclusivamente bits, o processo seria insustentável. É necessário uma abstração dos bits.

- O primeiro conceito importante que surge é o de **tipo de dado**, que nos permite **especificar como entenderemos e trabalharemos com conjuntos e padrões particulares de bits**.
- Os dados derivam de descrições **matemáticas** ou em **linguagem natural** do mundo e, assim, os blocos mais básicos de dados são:
 - **Números** (naturais, inteiros, reais...)
 - **Caracteres**

Tipo de Dado

01111111	01000101	01001100	01000110	00000010
00000000	00000000	00000000	00000000	00000000
00000010	00000000	00111110	00000000	00000001
10110000	00000101	01000000	00000000	00000000
01000000	00000000	00000000	00000000	00000000
11010000	00010011	00000000	00000000	00000000
00000000	00000000	00000000	00000000	01000000
00001001	00000000	01000000	00000000	00100100

- O **tipo de dado** não é, então, uma estrutura de dados. É uma **maneira de especificar como determinados padrões particulares de bits 0s e 1s serão entendidos e manipulados para representar números e caracteres.**
- Define-se então um **tipo de dado** como um **conjunto de valores** e um **conjunto de operações** sobre esses valores.

Tipo de dado = Conjunto de Valores + Conjunto de Operações

Tipo de Dado

01111111	01000101	01001100	01000110	00000010
00000000	00000000	00000000	00000000	00000000
00000010	00000000	00111110	00000000	00000001
10110000	00000101	01000000	00000000	00000000
01000000	00000000	00000000	00000000	00000000
11010000	00010011	00000000	00000000	00000000
00000000	00000000	00000000	00000000	01000000
00001001	00000000	01000000	00000000	00100100

- Idealmente só precisaríamos de apenas 2 tipos de dados:
 - Número
 - Caractere
- Na prática, como a memória é limitada, **trocamos espaço por exatidão** (faixa de valores no caso dos inteiros; precisão no caso dos decimais) e **criamos mais tipos de dados do que os idealmente necessários**:
 - short int, int, long int, long long int
 - float, double, long double
- O número de bits utilizado nessa troca é “machine-dependent”, mesmo que o programador pense mais em termos de necessidades numéricas ao invés de capacidades do computador (fique atento às variações máquina/compilador).
 - -32.768 até 32.767, ao invés de 16 bits

Estruturas de Dados

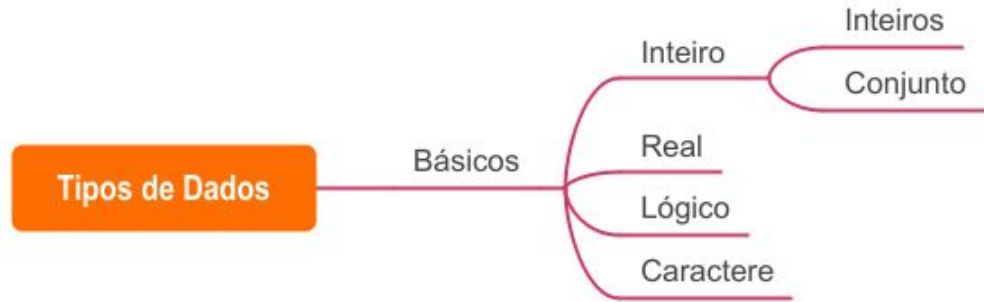
- São as **estruturas de memória que armazenam valores** (dados) que estão sendo manipulados pelo algoritmo, de forma **organizada** e **otimizada** para o processamento que está sendo realizado.
- Podem ser extremamente simples ou muito complexas.
- Podemos dividir as estruturas de dados em duas grandes “categorias”:
 - Estruturas para armazenar **dados que se comportam como uma unidade**
 - Estruturas para armazenar **diversos dados ao mesmo tempo**

Estruturas de Dados

Tipos de Dados

Estruturas de Dados

Estruturas de Dados

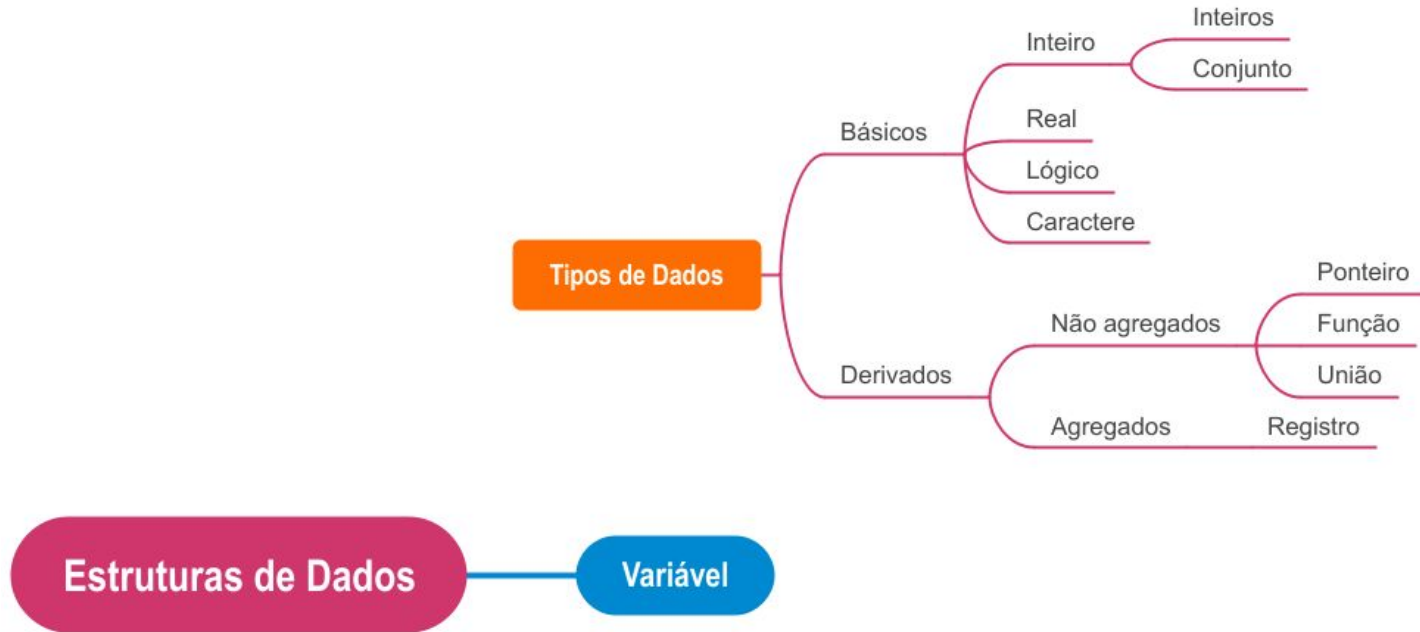


Estruturas de Dados

Variável

Tipos de dados básicos (primitivos): não podem ser decompostos em partes menores, representam o entendimento direto dos padrões de bits como dados.

Estruturas de Dados



Tipos de dados derivados: são construídos a partir dos tipos de dados básicos, e ainda são considerados como unidade.

Estruturas de Dados



Estruturas de Dados

Variável

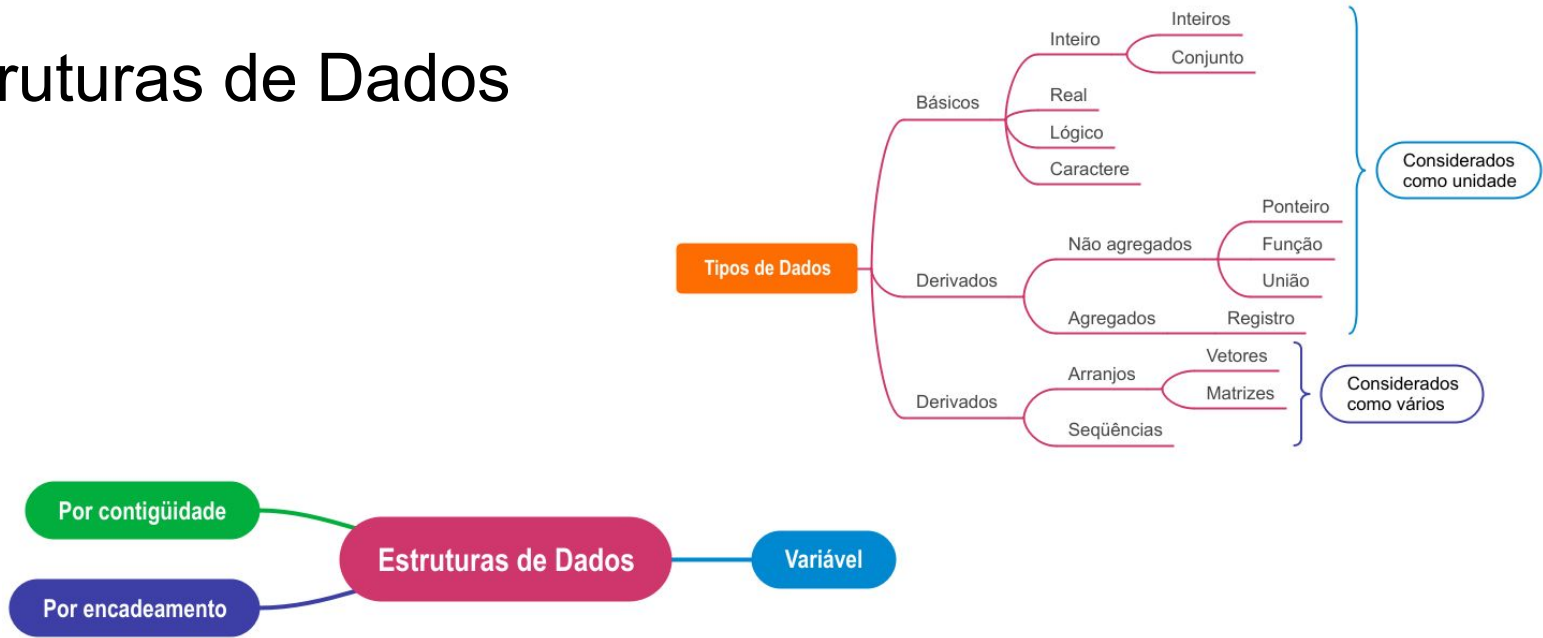
Como são **considerados como unidade**, basta uma variável (mesmo que, fisicamente, o armazenamento seja complexo). Obs.: structs podem ser complexos.

Estruturas de Dados



Tipos de dados derivados (considerados como vários): como armazenar diversos dados (ordenados ou não)?

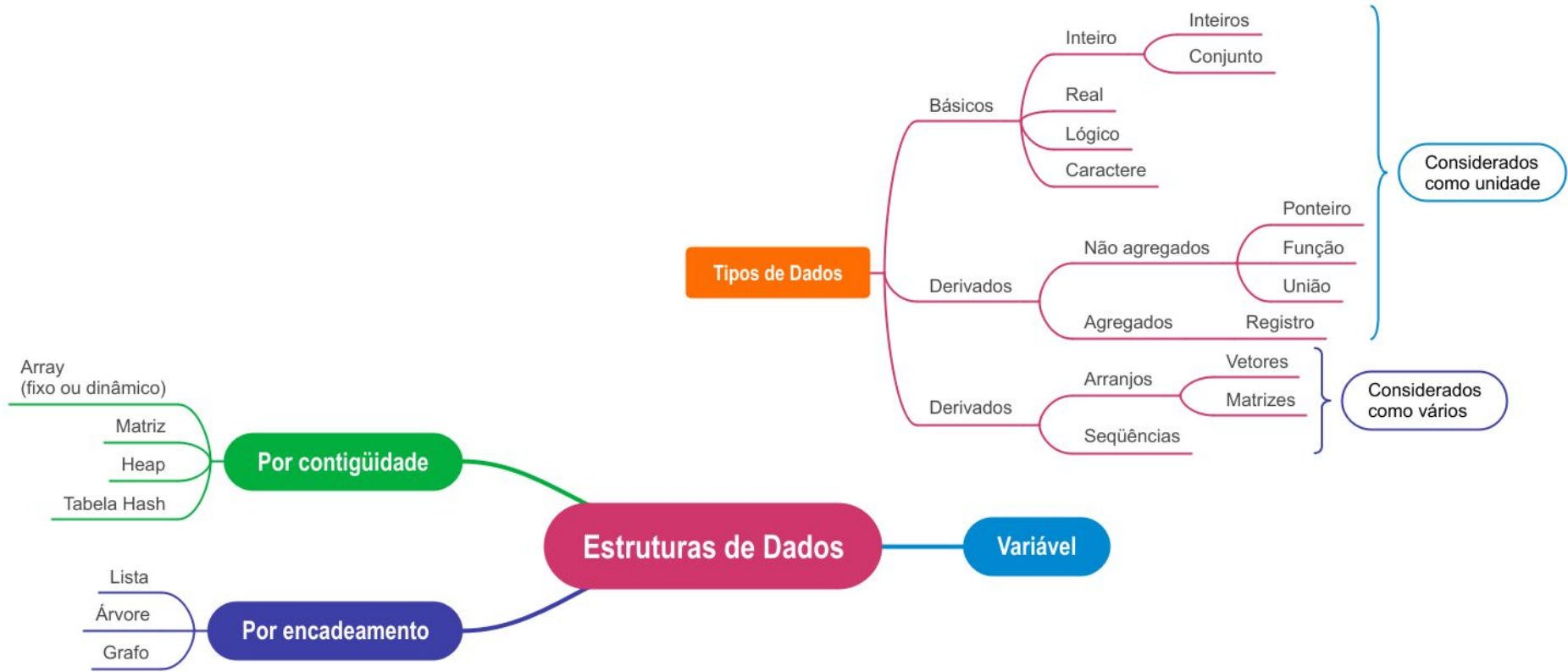
Estruturas de Dados



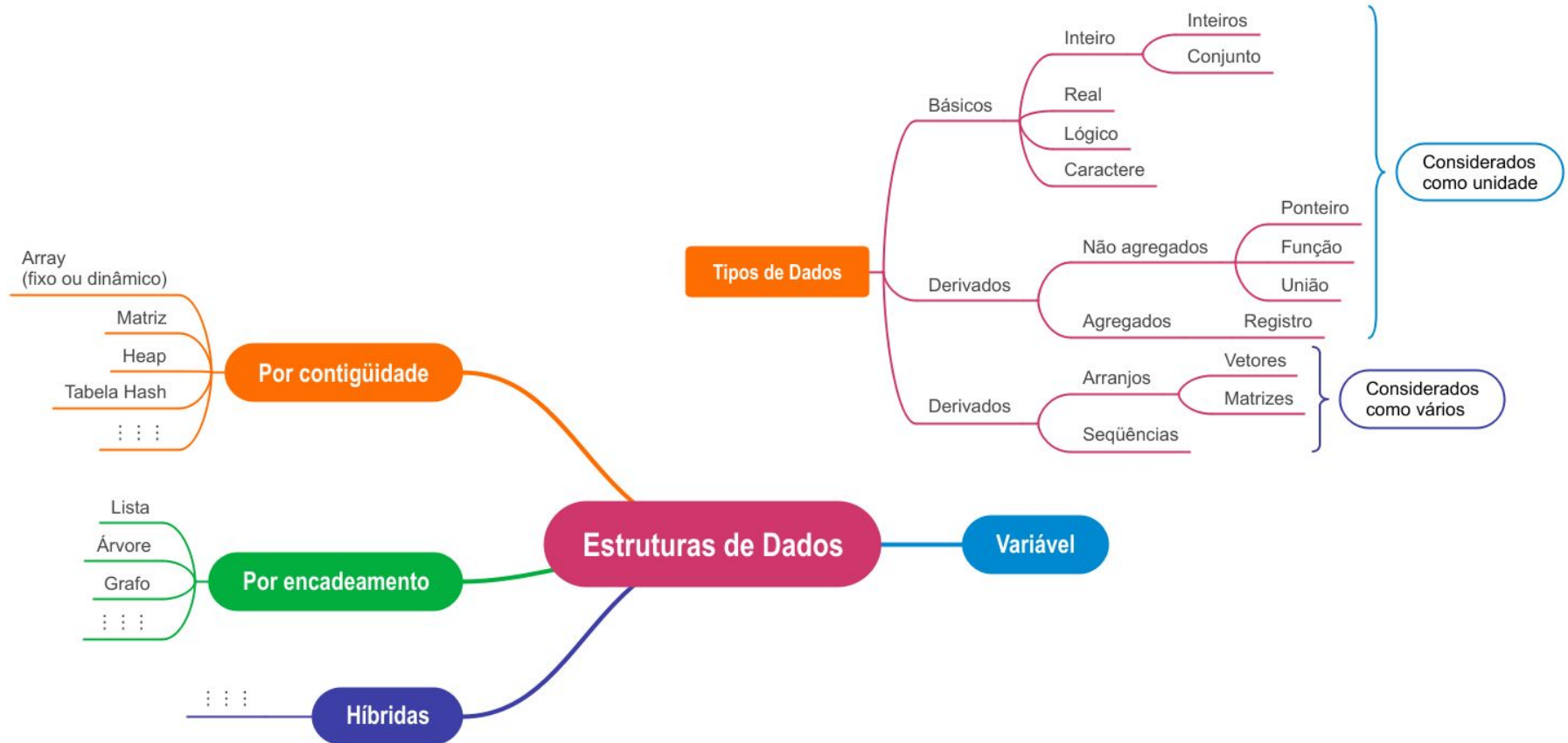
Por contigüidade: os dados são armazenados em posições contíguas na memória. A ordem é definida implicitamente pela posição, geralmente com tamanho máximo definido previamente.

Por encadeamento: os dados são armazenados em posições aleatórias na memória alocadas dinamicamente, com ponteiros para indicar o endereço do próximo dado.

Estruturas de Dados



Estruturas de Dados



Contigüidade x Encadeamento

- **Estrutura por Contigüidade:**

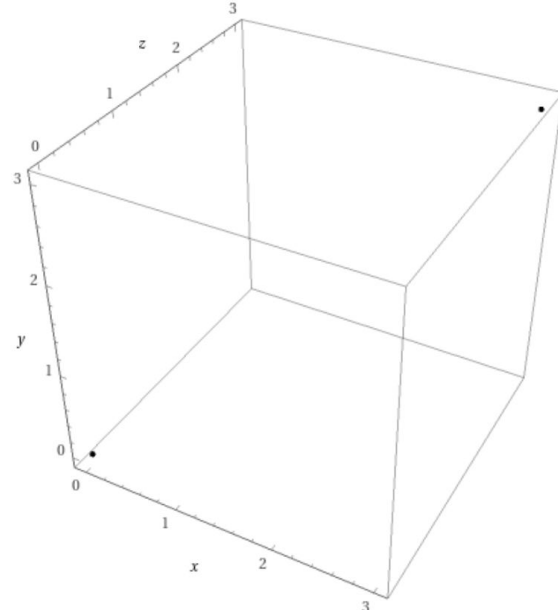
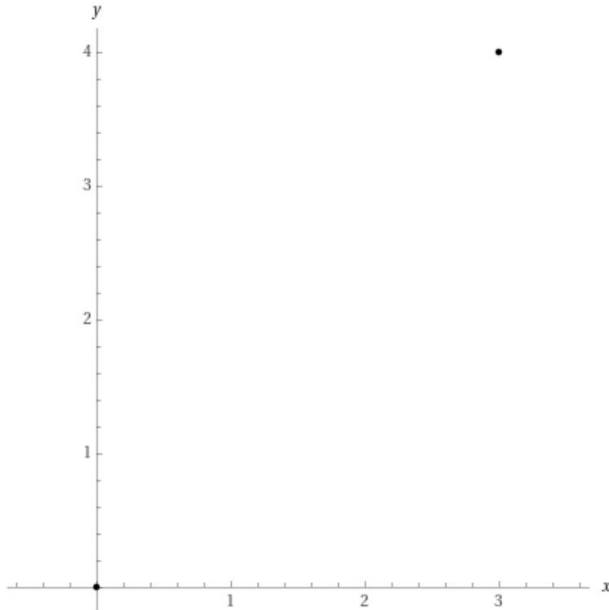
- Vantagens:
 - Tempo de acesso constante dado o índice
 - Eficiência de espaço (só tem os dados, não tem ponteiros)
 - Localidade de memória (transferência, iteração e cache)
- Desvantagens:
 - Não ajusta tamanho em execução (array dinâmico ajusta)
 - Não permite compartilhamento de memória
 - Inserção e exclusão complexa e custosa pelo deslocamento

- **Estrutura por Encadeamento:**

- Vantagens:
 - Compartilhamento de memória
 - Flexibilidade
 - Inserção e exclusão facilitada
- Desvantagens:
 - Transferência
 - Espaço
 - Acesso
 - Busca

Tipos de Dados criados pelo programador

- Podemos criar nossos próprios tipos de dados, **combinando as estruturas de dados** e **especificando os comportamentos**. Vamos criar tipos para pontos no plano e no espaço cartesiano, para calcular as distâncias:



Interface: ponto.h

```
1  /**
2   * Arquivo: ponto.h
3   * Versão : 1.0
4   * Data   : 2024-10-14 16:43
5   * -----
6   * Esta interface cria dois Tipos de Dados, Ponto2D e Ponto3D, para armazenar
7   * coordenadas de pontos no plano e no espaço. Também fornece funções para
8   * calcular a distância entre 2 pontos.
9   *
10  * Prof.: Abrantes Araújo Silva Filho (Computação Raiz)
11  *      www.computacaoraiz.com.br
12  *      www.youtube.com.br/computacaoraiz
13  *      github.com/computacaoraiz
14  *      twitter.com/ComputacaoRaiz
15  *      www.linkedin.com/company/computacaoraiz
16  *      www.abrantes.pro.br
17  *      github.com/abrantesasf
18  */
19
20 /* Inicia boilerplate da interface: */
21
22 #ifndef _PONTOS_H
23 #define _PONTOS_H
24
25 /* Includes: */
26
27 #include "genlib.h"
28 #include <math.h>
29 #include "simpio.h"
30
```

Interface: ponto.h

```
31 /**
32  * TIPO DE DADO: Ponto2D
33  * -----
34  * Este tipo de dado armazena as coordenadas X e Y de um ponto no plano,
35  * considerando um sistema de coordenadas cartesiano.
36  */
37
38 struct st_Ponto2D
39 {
40     double x;    // abscissa
41     double y;    // ordenada
42 };
43
44 typedef struct st_Ponto2D Ponto2D;
45
46 /**
47  * TIPO DE DADO: Ponto3D
48  * -----
49  * Este tipo de dado armazena as coordenadas X, Y e Z de um ponto no espaço,
50  * considerando um sistema de coordenadas cartesiano.
51  */
52
53 struct st_Ponto3D
54 {
55     double x;    // abscissa
56     double y;    // ordenada
57     double z;    // profundidade
58 };
59
60 typedef struct st_Ponto3D Ponto3D;
61
```

Interface: ponto.h

```
62 /**
63  * FUNÇÃO: euclidiana_2d
64  * Uso: euclidiana_2d(Ponto2D, Ponto2D);
65  * -----
66  * A função recebe como argumentos dois Ponto2D e retorna a distância
67  * Euclidiana entre esses dois pontos, no plano.
68  */
69
70 double euclidiana_2d (Ponto2D P, Ponto2D Q);
71
72 /**
73  * FUNÇÃO: euclidiana_2d
74  * Uso: euclidiana_2d(Ponto2D, Ponto2D);
75  * -----
76  * A função recebe como argumentos dois Ponto2D e retorna a distância
77  * Euclidiana entre esses dois pontos, no plano.
78  */
79
80 double euclidiana_3d (Ponto3D P, Ponto3D Q);
81
82 /* Finaliza boilerplate: */
83
84 #endif
```

Implementação: ponto.c

```
1  /**
2   * Arquivo: ponto.c
3   * Versão : 1.0
4   * Data   : 2024-10-14 16:43
5   * -----
6   * Este arquivo implementa a interface ponto.h.
7   *
8   * Prof.: Abrantes Araújo Silva Filho (Computação Raiz)
9   *          www.computacaoraiz.com.br
10  *          www.youtube.com.br/computacaoraiz
11  *          github.com/computacaoraiz
12  *          twitter.com/ComputacaoRaiz
13  *          www.linkedin.com/company/computacaoraiz
14  *          www.abrantes.pro.br
15  *          github.com/abrantesasf
16  */
17
18  /* Includes: */
19
20  #include "ponto.h"
21
```

Implementação: ponto.c

```
22 /**
23  * FUNÇÃO: euclidiana_2d
24  * Uso: euclidiana_2d(Ponto2D, Ponto2D);
25  * -----
26  * A função recebe como argumentos dois Ponto2D e retorna a distância
27  * Euclidiana entre esses dois pontos, no plano.
28  */
29
30 double euclidiana_2d (Ponto2D P, Ponto2D Q)
31 {
32     return sqrt(pow((P.x - Q.x), 2.0) +
33                 pow((P.y - Q.y), 2.0));
34 }
35
36 /**
37  * FUNÇÃO: euclidiana_2d
38  * Uso: euclidiana_2d(Ponto2D, Ponto2D);
39  * -----
40  * A função recebe como argumentos dois Ponto2D e retorna a distância
41  * Euclidiana entre esses dois pontos, no plano.
42  */
43
44 double euclidiana_3d (Ponto3D P, Ponto3D Q)
45 {
46     return sqrt(pow((P.x - Q.x), 2.0) +
47                 pow((P.y - Q.y), 2.0) +
48                 pow((P.z - Q.z), 2.0));
49 }
50
```

Cliente: geometria.c

```
#include "genlib.h"
#include "ponto.h"
#include "simpio.h"
```

```
/* Função Main: */
```

```
int main (void)
{
```

```
    Ponto2D A;
    A.x = 0.0, A.y = 0.0;
```

```
    Ponto2D B;
    B.x = 3.0, B.y = 4.0;
```

```
    printf("A distância entre (0.1f, 0.1f) e (0.1f, 0.1f) é de 0.5f.\n",
           A.x, A.y, B.x, B.y, euclidiana_2d(A, B));
```

```
    Ponto3D X;
    X.x = 0.0, X.y = 0.0, X.z = 0.0;
```

```
    Ponto3D Y;
    Y.x = 3.0, Y.y = 3.0, Y.z = 3.0;
```

```
    printf("A distância entre (0.1f, 0.1f, 0.1f) e (0.1f, 0.1f, 0.1f) é de 0.5f.\n",
           X.x, X.y, X.z, Y.x, Y.y, Y.z, euclidiana_3d(X, Y));
```

```
}
```

```
[abrantestasf@ideapad ~/ed1/cap08]$ ./geometria
```

```
A distância entre (0.0, 0.0) e (3.0, 4.0) é de 5.00000.
```

```
A distância entre (0.0, 0.0, 0.0) e (3.0, 3.0, 3.0) é de 5.19615.
```

Ponto2D e Ponto3D: algumas questões

- Eles são realmente tipos de dados?
- A interface está bem projetada?
- O comportamento dos tipos está bem especificado?
- Há algum problema com esse tipo de dado?

Ponto2D e Ponto3D: algumas questões

- Eles são realmente tipos de dados?
 - Sim, são tipos de dados (TD)
- A interface está bem projetada?
 - Sim (apenas se considerarmos que queremos um **tipo concreto de dado**, não abstrato)
- O comportamento dos tipos está bem especificado?
 - Sim, as funções estão lá
- Há algum problema com esse tipo de dado?
 - **Sim, um problema GRAVE! O tipo de dado não é abstrato!!**

Tipos Abstratos de Dados (TADs)

- Um TAD é um tipo de dado que é **definido em termos de seu comportamento ao invés de sua representação interna**.
- Em um TAD a **representação/implementação interna não é importante e deve ficar escondida, opaca**, o que importa é o comportamento.
- Em um TAD a **interface esconde completamente a representação interna da implementação**. O usuário não consegue, de jeito nenhum, acessar detalhes internos da implementação e, portanto, ele é **obrigado a utilizar a interface para tudo**.
- Um tipo de dado que não esconde completamente sua representação interna é chamado de **Tipo Concreto de Dado (TCD)**.

Problemas dos tipos Ponto2D e Ponto3D

```
struct st_Ponto2D
{
    double x;    // abscissa
    double y;    // ordenada
};

typedef struct st_Ponto2D Ponto2D;

struct st_Ponto3D
{
    double x;    // abscissa
    double y;    // ordenada
    double z;    // profundidade
};

typedef struct st_Ponto3D Ponto3D;
```

A representação interna do tipo de dado está visível na interface! Isso NÃO É um tipo abstrato de dado!

Problemas dos tipos Ponto2D e Ponto3D

```
Ponto2D A;
```

```
A.x = 0.0, A.y = 0.0;
```

```
Ponto2D B;
```

```
B.x = 3.0, B.y = 4.0;
```

```
Ponto3D X;
```

```
X.x = 0.0, X.y = 0.0, X.z = 0.0;
```

```
Ponto3D Y;
```

```
Y.x = 3.0, Y.y = 3.0, Y.z = 3.0;
```

**Isso faz com que o cliente
utilize diretamente a
REPRESENTAÇÃO
INTERNA do tipo de dado!
É um erro! Por quê?**

Problemas dos tipos Ponto2D e Ponto3D

```
double meu_euclides (double x1, double x2, double y1, double y2)
{
    return sqrt(pow((x1 - x2), 2.0) + pow((y1 - y2), 2.0));
}
```

```
[abrantesasf@ideapad ~/ed1/cap08]$ ./geometria
A distância entre (0.0, 0.0) e (3.0, 4.0) é de 5.00000.
A distância entre (0.0, 0.0) e (3.0, 4.0) é de 4.24264.
```

Como o usuário sabe a representação interna, ele pode criar outras funções que substituem o comportamento esperado de seu tipo de dado, e isso pode terminar em um erro!

```
printf("A distância entre (%.1f, %.1f) e (%.1f, %.1f) é de %.5f.\n",
      A.x, A.y, B.x, B.y, meu_euclides(A.x, B.x, A.y, Y.y));
```

Problemas dos tipos Ponto2D e Ponto3D

```
struct st_Ponto2D
{
    double X;    // abscissa
    double Y;    // ordenada
};

typedef struct st_Ponto2D Ponto2D;
```

```
struct st_Ponto3D
{
    double X;    // abscissa
    double Y;    // ordenada
    double Z;    // profundidade
};

typedef struct st_Ponto3D Ponto3D;
```

Se você alterar a representação interna do tipo de dado, TODOS os programas clientes que utilizavam a estrutura antiga irão parar de funcionar!

Solução: TADs Ponto2D e Ponto3D - pontoTAD.h

```
1  /**
2   * Arquivo: pontoTAD.h
3   * Versão : 1.0
4   * Data   : 2024-10-14 19:30
5   * -----
6   * Esta interface cria dois Tipos Abstratos de Dados, Ponto2D e Ponto3D, para
7   * armazenar coordenadas de pontos no plano e no espaço. Também fornece funções
8   * para criar e remover esses pontos, e calcular a distância entre 2 pontos.
9   *
10  * Prof.: Abrantes Araújo Silva Filho (Computação Raiz)
11  *      www.computacaoraiz.com.br
12  *      www.youtube.com.br/computacaoraiz
13  *      github.com/computacaoraiz
14  *      twitter.com/ComputacaoRaiz
15  *      www.linkedin.com/company/computacaoraiz
16  *      www.abrantes.pro.br
17  *      github.com/abrantesasf
18  */
19
20 /* Inicia boilerplate da interface: */
21
22 #ifndef _PONTOSTAD_H
23 #define _PONTOSTAD_H
24
25 /*** Includes: ***/
26
```

Solução: TADs Ponto2D e Ponto3D - pontoTAD.h

```
27  /** Tipos Abstratos de Dados: */
28
29  /**
30   * TIPO DE DADO: Ponto2D
31   * -----
32   * Este tipo de dado armazena as coordenadas X e Y de um ponto no plano,
33   * considerando um sistema de coordenadas cartesiano. Para a criação de um
34   * Ponto2D, ver a documentação da função "criar_Ponto2D" neste documento.
35   */
36
37  typedef struct st_Ponto2D *Ponto2D;
38
39  /**
40   * TIPO DE DADO: Ponto3D
41   * -----
42   * Este tipo de dado armazena as coordenadas X, Y e Z de um ponto no espaço,
43   * considerando um sistema de coordenadas cartesiano. Para a criação de um
44   * Ponto3D, ver a documentação da função "criar_Ponto3D" neste documento.
45   */
46
47  typedef struct st_Ponto3D *Ponto3D;
48
```

Solução: TADs Ponto2D e Ponto3D - pontoTAD.h

```
49 /** Declarações de Subprogramas: ***/
50
51 /**
52  * FUNÇÃO: criar_Ponto2D
53  * Uso: criar_Ponto2D(x, y);
54  * -----
55  * Esta função recebe dois valores double, x e y, que correspondem às
56  * coordenadas x (abscissa) e y (ordenada) de um ponto no plano cartesiano,
57  * e retorna um ponteiro para um Ponto2D.
58  *
59  * Depois de criado esse ponto precisa ser apagado com o procedimento
60  * "apagar_Ponto2D", antes da finalização do programa.
61  */
62
63 Ponto2D criar_Ponto2D (double x, double y);
64
65 /**
66  * FUNÇÃO: criar_Ponto3D
67  * Uso: criar_Ponto3D(x, y, z);
68  * -----
69  * Esta função recebe três valores double, x, y e z, que correspondem às
70  * coordenadas x (abscissa), y (ordenada) e z (profundidade) de um ponto no
71  * espaço cartesiano, e retorna um ponteiro para um Ponto3D.
72  *
73  * Depois de criado esse ponto precisa ser apagado com o procedimento
74  * "apagar_Ponto3D", antes da finalização do programa.
75  */
76
77 Ponto3D criar_Ponto3D (double x, double y, double z);
78
```

Solução: TADs Ponto2D e Ponto3D - pontoTAD.h

```
79 /**
80  * PROCEDIMENTO: apagar_Ponto2D
81  * Uso: apagar_Ponto2D(&P);
82  * -----
83  * Este procedimento recebe um PONTEIRO para um Ponto2D como argumento e libera
84  * a memória alocada para esse Ponto2D, fazendo com que o Ponto2D aponte para
85  * NULL para evitar dangling pointer.
86  */
87
88 void apagar_Ponto2D (Ponto2D *P);
89
90 /**
91  * PROCEDIMENTO: apagar_Ponto3D
92  * Uso: apagar_Ponto3D(&P);
93  * -----
94  * Este procedimento recebe um PONTEIRO para um Ponto3D como argumento e libera
95  * a memória alocada para esse Ponto3D, fazendo com que o Ponto3D aponte para
96  * NULL para evitar dangling pointer.
97  */
98
99 void apagar_Ponto3D (Ponto3D *P);
100
```

Solução: TADs Ponto2D e Ponto3D - pontoTAD.h

```
101 /**
102  * FUNÇÃO: Ponto2D_getX
103  * Uso: Ponto2D_getX(P);
104  * -----
105  * Esta função recebe como argumento um Ponto2D e retorna a coordenada X
106  * (abscissa) deste ponto, como um valor double.
107  */
108
109 double Ponto2D_getX (Ponto2D P);
110
111 /**
112  * FUNÇÃO: Ponto2D_getY
113  * Uso: Ponto2D_getY(P);
114  * -----
115  * Esta função recebe como argumento um Ponto2D e retorna a coordenada Y
116  * (ordenada) deste ponto, como um valor double.
117  */
118
119 double Ponto2D_getY (Ponto2D P);
120
121 /**
122  * FUNÇÃO: Ponto3D_getX
123  * Uso: Ponto3D_getX(P);
124  * -----
125  * Esta função recebe como argumento um Ponto3D e retorna a coordenada X
126  * (abscissa) deste ponto, como um valor double.
127  */
128
129 double Ponto3D_getX (Ponto3D P);
130
```

Solução: TADs Ponto2D e Ponto3D - pontoTAD.h

```
131 /**
132  * FUNÇÃO: Ponto3d_getY
133  * Uso: Ponto3D_getY(P);
134  * -----
135  * Esta função recebe como argumento um Ponto3D e retorna a coordenada Y
136  * (ordenada) deste ponto, como um valor double.
137  */
138
139 double Ponto3D_getY (Ponto3D P);
140
141 /**
142  * FUNÇÃO: Ponto2D_getZ
143  * Uso: Ponto2D_getZ(P);
144  * -----
145  * Esta função recebe como argumento um Ponto2D e retorna a coordenada Z
146  * (profundidade) deste ponto, como um valor double.
147  */
148
149 double Ponto3D_getZ (Ponto3D P);
150
151 /**
152  * PROCEDIMENTO: Ponto2D_setX
153  * Uso: Ponto2D_setX(Ponto2D P, double x);
154  * -----
155  * Esta função recebe como argumentos um Ponto2D e um valor double
156  * correspondendo a uma coordenada X (abscissa), e atribui esse valor como a
157  * nova abscissa do Ponto2D.
158  */
159
160 void Ponto2D_setX (Ponto2D P, double x);
161
```

Solução: TADs Ponto2D e Ponto3D - pontoTAD.h

```
162 /**
163  * PROCEDIMENTO: Ponto2D_setY
164  * Uso: Ponto2D_setY(Ponto2D P, double y);
165  * -----
166  * Esta função recebe como argumentos um Ponto2D e um valor double
167  * correspondendo a uma coordenada Y (ordenada), e atribui esse valor como a
168  * nova ordenada do Ponto2D.
169  */
170
171 void Ponto2D_setY (Ponto2D P, double y);
172
173 /**
174  * PROCEDIMENTO: Ponto3D_setX
175  * Uso: Ponto3D_setX(Ponto3D P, double x);
176  * -----
177  * Esta função recebe como argumentos um Ponto3D e um valor double
178  * correspondendo a uma coordenada X (abscissa), e atribui esse valor como a
179  * nova abscissa do Ponto3D.
180  */
181
182 void Ponto3D_setX (Ponto3D P, double x);
183
184 /**
185  * PROCEDIMENTO: Ponto3D_setY
186  * Uso: Ponto3D_setY(Ponto3D P, double y);
187  * -----
188  * Esta função recebe como argumentos um Ponto3D e um valor double
189  * correspondendo a uma coordenada Y (ordenada), e atribui esse valor como a
190  * nova ordenada do Ponto3D.
191  */
192
193 void Ponto3D_setY (Ponto3D P, double y);
194
```

Solução: TADs Ponto2D e Ponto3D - pontoTAD.h

```
195 /**
196  * PROCEDIMENTO: Ponto3D_setZ
197  * Uso: Ponto3D_setZ(Ponto3D P, double z);
198  * -----
199  * Esta função recebe como argumentos um Ponto3D e um valor double
200  * correspondendo a uma coordenada Z (profundidade), e atribui esse valor como a
201  * nova profundidade do Ponto3D.
202  */
203
204 void Ponto3D_setZ (Ponto3D P, double z);
205
206 /**
207  * FUNÇÃO: euclidiana_2d
208  * Uso: euclidiana_2d(Ponto2D, Ponto2D);
209  * -----
210  * A função recebe como argumentos dois Ponto2D e retorna a distância
211  * Euclidiana entre esses dois pontos, no plano.
212  */
213
214 double euclidiana_2d (Ponto2D P, Ponto2D Q);
215
216 /**
217  * FUNÇÃO: euclidiana_3d
218  * Uso: euclidiana_3d(Ponto3D, Ponto3D);
219  * -----
220  * A função recebe como argumentos dois Ponto3D e retorna a distância
221  * Euclidiana entre esses dois pontos, no espaço.
222  */
223
224 double euclidiana_3d (Ponto3D P, Ponto3D Q);
225
226 /* Finaliza boilerplate: */
227
228 #endif
```

Solução: TADs Ponto2D e Ponto3D - pontoTAD.c

```
1  /**
2   * Arquivo: pontoTAD.c
3   * Versão : 1.0
4   * Data   : 2024-10-14 19:44
5   * -----
6   * Este arquivo implementa a interface pontoTAD.h.
7   *
8   * Prof.: Abrantes Araújo Silva Filho (Computação Raiz)
9   *      www.computacaoraiz.com.br
10  *      www.youtube.com.br/computacaoraiz
11  *      github.com/computacaoraiz
12  *      twitter.com/ComputacaoRaiz
13  *      www.linkedin.com/company/computacaoraiz
14  *      www.abrantes.pro.br
15  *      github.com/abrantesasf
16  */
17
18  /** Includes: */
19
20  #include <math.h>
21  #include "pontoTAD.h"
22  #include <stdio.h>
23  #include <stdlib.h>
24
```

Solução: TADs Ponto2D e Ponto3D - pontoTAD.c

```
25  /** Tipos Abstratos de Dados: */
26
27  /**
28   * TIPO DE DADO: Ponto2D
29   * -----
30   * Implementação do tipo de dado Ponto2D, que armazena as coordenadas de um
31   * ponto no plano, considerando um sistema de coordenadas cartesiano.
32   */
33
34  struct st_Ponto2D
35  {
36      double coordX;
37      double coordY;
38  };
39
40  /**
41   * TIPO DE DADO: Ponto3D
42   * -----
43   * Implementação do tipo de dado Ponto3D, que armazena as coordenadas de um
44   * ponto no espaço, considerando um sistema de coordenadas cartesiano.
45   */
46
47  struct st_Ponto3D
48  {
49      double coordX;
50      double coordY;
51      double coordZ;
52  };
53
```

Solução: TADs Ponto2D e Ponto3D - pontoTAD.c

```
54 /** Definições de Subprogramas: */
55
56 /**
57  * FUNÇÃO: criar_Ponto2D
58  * Uso: criar_Ponto2D(x, y);
59  * -----
60  * Um Ponto2D é um ponteiro para uma struct st_Ponto2D e, portanto, essa função
61  * retorna um ponteiro para essa struct (indiretamente, através do retorno de um
62  * Ponto2D). As coordenadas são mantidas em variáveis double independentes.
63  */
64
65 Ponto2D criar_Ponto2D (double x, double y)
66 {
67     Ponto2D T = malloc(sizeof(struct st_Ponto2D));
68     if (T == NULL)
69     {
70         printf("Não foi possível criar o Ponto2D!\n");
71         exit(1);
72     }
73
74     T->coordX = x;
75     T->coordY = y;
76
77     return T;
78 }
79
```

Solução: TADs Ponto2D e Ponto3D - pontoTAD.c

```
80  /**
81   * FUNÇÃO: criar_Ponto3D
82   * Uso: criar_Ponto3D(x, y, z);
83   * -----
84   * Um Ponto3D é um ponteiro para uma struct st_Ponto3D e, portanto, essa função
85   * retorna um ponteiro para essa struct (indiretamente, através do retorno de um
86   * Ponto3D). As coordenadas são mantidas em variáveis double independentes.
87   */
88
89  Ponto3D criar_Ponto3D (double x, double y, double z)
90  {
91      Ponto3D T = malloc(sizeof(struct st_Ponto3D));
92      if (T == NULL)
93      {
94          printf("Não foi possível criar o Ponto3D!\n");
95          exit(1);
96      }
97
98      T->coordX = x;
99      T->coordY = y;
100     T->coordZ = z;
101
102     return T;
103 }
104
```

Solução: TADs Ponto2D e Ponto3D - pontoTAD.c

```
105  /**
106   * PROCEDIMENTO: apagar_Ponto2D
107   * Uso: apagar_Ponto2D(&P);
108   * -----
109   * Este procedimento recebe um PONTEIRO para um Ponto2D como argumento e libera
110   * a memória alocada para esse Ponto2D, fazendo com que o Ponto2D aponte para
111   * NULL para evitar dangling pointer. É necessário utilizar um PONTEIRO para um
112   * Ponto2D pois, caso contrário, não seria possível atribuir o valor NULL ao
113   * Ponto2D que está sendo desalocado (ponteiros são passados por valor).
114   */
115
116 void apagar_Ponto2D (Ponto2D *P)
117 {
118     if (*P != NULL)
119     {
120         free(*P);
121         *P = NULL;
122     }
123 }
124
```

Solução: TADs Ponto2D e Ponto3D - pontoTAD.c

```
125  /**
126   * PROCEDIMENTO: apagar_Ponto3D
127   * Uso: apagar_Ponto3D(&P);
128   * -----
129   * Este procedimento recebe um PONTEIRO para um Ponto3D como argumento e libera
130   * a memória alocada para esse Ponto3D, fazendo com que o Ponto3D aponte para
131   * NULL para evitar dangling pointer. É necessário utilizar um PONTEIRO para um
132   * Ponto3D pois, caso contrário, não seria possível atribuir o valor NULL ao
133   * Ponto3D que está sendo desalocado (ponteiros são passados por valor).
134   */
135
136 void apagar_Ponto3D (Ponto3D *P)
137 {
138     if (*P != NULL)
139     {
140         free(*P);
141         *P = NULL;
142     }
143 }
144
```

Solução: TADs Ponto2D e Ponto3D - pontoTAD.c

```
145  /**
146   * FUNÇÃO: Ponto2D_getX
147   * Uso: Ponto2D_getX(P);
148   * -----
149   * Verifica se o Ponto2D não é nulo e retorna a coordenada X.
150   */
151
152  double Ponto2D_getX (Ponto2D P)
153  {
154      if (P == NULL)
155      {
156          printf("Erro ao obter a coordenada X do ponto especificado.\n");
157          exit(1);
158      }
159      return P->coordX;
160  }
161
```

Solução: TADs Ponto2D e Ponto3D - pontoTAD.c

```
230 /**
231  * PROCEDIMENTO: Ponto2D_setX
232  * Uso: Ponto2D_setX(Ponto2D P, double x);
233  * -----
234  * Verifica se o Ponto2D não é nulo e atribui coordenada X.
235  */
236
237 void Ponto2D_setX (Ponto2D P, double x)
238 {
239     if (P == NULL)
240     {
241         printf("Erro ao atribuir a coordenada X ao ponto especificado.\n");
242         exit(1);
243     }
244     P->coordX = x;
245 }
246
```

Solução: TADs Ponto2D e Ponto3D - pontoTAD.c

```
315  /**
316   * FUNÇÃO: euclidiana_2d
317   * Uso: euclidiana_2d(Ponto2D, Ponto2D);
318   * -----
319   * A função recebe como argumentos dois Ponto2D e retorna a distância
320   * Euclidiana entre esses dois pontos, no plano.
321   */
322
323  double euclidiana_2d (Ponto2D P, Ponto2D Q)
324  {
325      if (P == NULL || Q == NULL)
326      {
327          printf("Erro ao calcular distância Euclidiana no plano.\n");
328          exit(1);
329      }
330      return sqrt(pow((P->coordX - Q->coordX), 2.0) +
331                  pow((P->coordY - Q->coordY), 2.0));
332  }
333
```

Solução: TADs Ponto2D e Ponto3D - pontoTAD.c

```
334 /**
335  * FUNÇÃO: euclidiana_3d
336  * Uso: euclidiana_3d(Ponto3D, Ponto3D);
337  * -----
338  * A função recebe como argumentos dois Ponto3D e retorna a distância
339  * Euclidiana entre esses dois pontos, no espaço.
340  */
341
342 double euclidiana_3d (Ponto3D P, Ponto3D Q)
343 {
344     if (P == NULL || Q == NULL)
345     {
346         printf("Erro ao calcular a distancia Euclidiana no espaço.\n");
347     }
348     return sqrt(pow((P->coordX - Q->coordX), 2.0) +
349                 pow((P->coordY - Q->coordY), 2.0) +
350                 pow((P->coordZ - Q->coordZ), 2.0));
351 }
```

Solução: cliente - geometriaTAD.c

```
1  /**
2   * Arquivo: geometriaTAD.c
3   * Versão : 1.0
4   * Data   : 2024-10-14 21:43
5   * -----
6   * Este programa implementa um cliente simples para demonstrar o uso de tipos
7   * abstratos de dados criados pelo usuário, o Ponto2D e o Ponto3D.
8   *
9   * Prof.: Abrantes Araújo Silva Filho (Computação Raiz)
10  *      www.computacaoraiz.com.br
11  *      www.youtube.com.br/computacaoraiz
12  *      github.com/computacaoraiz
13  *      twitter.com/ComputacaoRaiz
14  *      www.linkedin.com/company/computacaoraiz
15  *      www.abrantes.pro.br
16  *      github.com/abrantesasf
17  */
18
19 #include "genlib.h"
20 #include "pontoTAD.h"
21 #include "simpio.h"
22
```

Solução: cliente - geometriaTAD.c

```
23  /* Função Main: */
24
25  int main (void)
26  {
27      // Cria pontos:
28      Ponto2D P = criar_Ponto2D(0.0, 0.0);
29      Ponto2D Q = criar_Ponto2D(3.0, 4.0);
30
31      Ponto3D W = criar_Ponto3D(0.0, 0.0, 0.0);
32      Ponto3D T = criar_Ponto3D(3.0, 3.0, 3.0);
33
34      // Testa gets e cálculo das distâncias:
35      printf("Distância entre (%.1f, %.1f) e (%.1f, %.1f): %.5f\n",
36            Ponto2D_getX(P), Ponto2D_getY(P), Ponto2D_getX(Q), Ponto2D_getY(Q),
37            euclidiana_2d(P, Q));
38
39      printf("Distância entre (%.1f, %.1f, %.1f) e (%.1f, %.1f, %.1f): %.5f\n",
40            Ponto3D_getX(W), Ponto3D_getY(W), Ponto3D_getZ(W),
41            Ponto3D_getX(T), Ponto3D_getY(T), Ponto3D_getZ(T),
42            euclidiana_3d(W, T));
43  }
```

Solução: cliente - geometriaTAD.c

```
44     // Testa sets:
45     Ponto2D_setX(P, 3.0);
46     Ponto2D_setY(P, 4.0);
47
48     Ponto3D_setX(W, 3.0);
49     Ponto3D_setY(W, 3.0);
50     Ponto3D_setZ(W, 3.0);
51
52     // Verifica sets:
53     printf("Distância entre (%.1f, %.1f) e (%.1f, %.1f): %.5f\n",
54           Ponto2D_getX(P), Ponto2D_getY(P), Ponto2D_getX(Q), Ponto2D_getY(Q),
55           euclidiana_2d(P, Q));
56
57     printf("Distância entre (%.1f, %.1f, %.1f) e (%.1f, %.1f, %.1f): %.5f\n",
58           Ponto3D_getX(W), Ponto3D_getY(W), Ponto3D_getZ(W),
59           Ponto3D_getX(T), Ponto3D_getY(T), Ponto3D_getZ(T),
60           euclidiana_3d(W, T));
61
62     // Testa desalocação de pontos:
63     apagar_Ponto2D(&P);
64     apagar_Ponto2D(&Q);
65     apagar_Ponto3D(&W);
66     apagar_Ponto3D(&T);
67
68     // Tem que dar erro:
69     //Ponto2D_setX(P, 3.0);
70
71     return 0;
72 }
```

Por que pontoTAD.h é melhor do que ponto.h?

- Implementa **tipos abstratos definidos pelo seu comportamento** (criar ponto, apagar ponto, set de coordenadas, get de coordenadas, cálculo da distância) **e não pela sua estrutura interna**
- **Estrutura interna está oculta**, então o cliente não consegue acessar diretamente. **Tudo tem de ser feito pela interface.**
- Como a representação interna está oculta, **o programador pode alterar completamente a implementação sem alterar os clientes** (desde que ele não altere a interface).
- **Segurança**: a barreira da abstração impede que um interfira no outro.

NÃO CONFUNDA TAD COM ESTRUTURA DE DADOS

- Uma dificuldade comum dos estudantes é confundir os conceitos de TAD e de Estruturas de Dados (pois realmente são muito relacionados). Mas atenção: **NÃO CONFUNDA TAD COM ESTRUTURA DE DADOS!**
 - Um **TAD é um tipo abstrato de dado** criado pelo programador cuja principal característica é ser **definido por seu comportamento**, não por sua estrutura interna.
 - Uma **Estrutura de Dados** é uma **estrutura de memória** capaz de **armazenar de forma organizada e otimizada**, os dados que estão em processamento.
 - **Podemos implementar um determinado TAD com diversas Estruturas de Dados**, por exemplo: uma pilha (um TAD importante na computação) pode ser implementado com as seguintes estruturas de dados:
 - Arrays
 - Listas encadeadas

NÃO CONFUNDA TAD COM ESTRUTURA DE DADOS

- Podemos criar TADs apenas para implementar uma determinada estrutura de dados, por exemplo: uma árvore binária de busca pode ser um TAD implementado sobre a estrutura de dados árvore binária.
 - Sim, isso é confuso e muitos autores não entram em consenso.
- Se estiver em dúvida, lembre-se do seguinte: o objeto em questão é definido pelo comportamento ao invés da estrutura interna?

Principais TADs na computação

- **Containers**: são TADs que permitem o armazenamento e a busca de dados **de forma independente do conteúdo**. Os dois principais containers são:
 - **Stack** (pilha): LIFO
 - `push(item, stack)`
 - `pop(stack)`
 - **Queue** (fila): FIFO
 - `enqueue(item, queue)`
 - `dequeue(queue)`
- Podem ser implementados usando as seguintes estruturas de dados:
 - arrays
 - lists

Principais TADs na computação

- **Dicionários**: são TADs que permitem o armazenamento e a busca de dados **pelo próprio conteúdo dos dados** (através de chaves ou valores). O principal dicionário é:
 - **Dictionary** (dicionário):
 - `search(k, dictionary)`
 - `insert(k, dictionary)`
 - `delete(k, dictionary)`
- Podem ser implementados usando as seguintes estruturas de dados:
 - arrays
 - lists
 - árvores binárias de busca
 - hash tables

Principais TADs na computação

- **Filas de Prioridade**: são TADs que permitem o processamento de itens de acordo com uma **ordem pré-estabelecida (prioridade)**. A principal fila de prioridade é:
 - **Priority Queue** (fila de prioridade):
 - `search_min(k, pq)`
 - `search_max(k, pq)`
 - `insert(k, pq)`
 - `delete_min(k, pq)`
 - `delete_max(k, pq)`
- Podem ser implementados usando as seguintes estruturas de dados:
 - arrays
 - árvores binárias de busca balanceadas

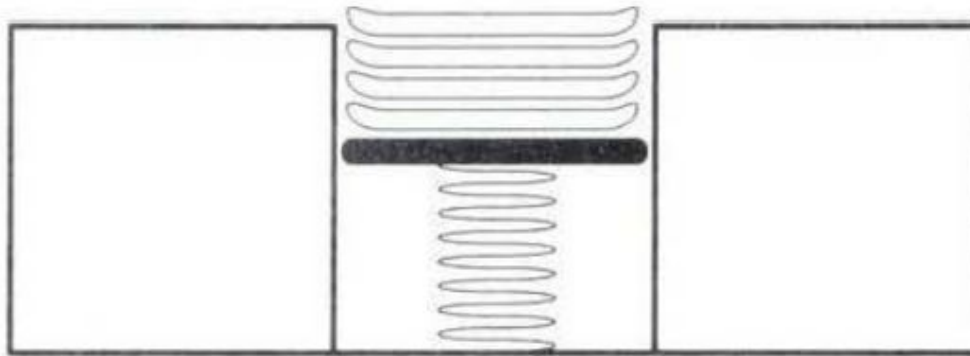
Principais TADs na computação

- Existem muitos outros TADs importantes na computação (além dos TADs que você mesmo vai criar para suas aplicações).
- Veremos diversos desses TADs agora (e nos próximos capítulos!).
- Vamos prosseguir com a seguinte abordagem:
 - Estudaremos um TAD e, no momento de implementar esse TAD, escolheremos uma estrutura de dados apropriada (e nesse momento estudaremos as estruturas de dados).

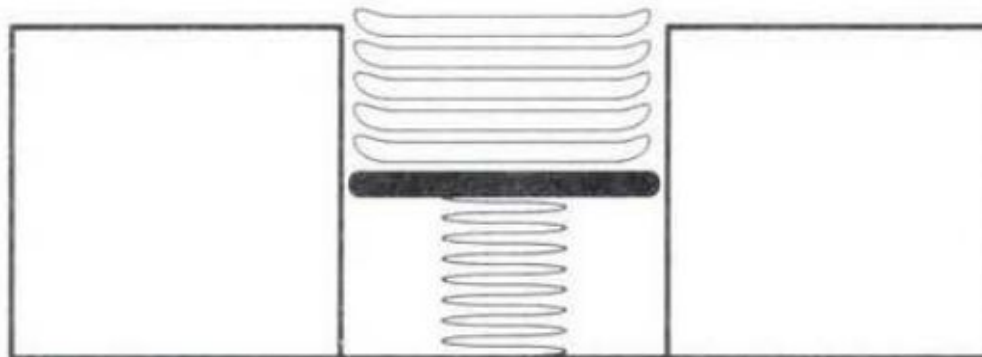
Stack (pilha)

- É um TAD da categoria dos **containers**, ou seja, é um TAD que **permite armazenar e recuperar dados independentemente de seu valor/conteúdo**.
- A principal característica do comportamento de uma pilha é que **os dados só podem ser retirados da pilha na ordem inversa em que foram adicionados**, ou seja, de modo **LIFO** (last in, first out).
- O comportamento da pilha é dado por 2 subprogramas principais e outros acessórios:
 - **push(item, pilha)** insere o item na pilha
 - **pop(pilha)** remove o item no topo da pilha
 - **vazia(pilha)** verifica se a pilha está vazia
 - **cheia(pilha)** verifica se a pilha está cheia
 - **quantidade(pilha)** verifica quantos elementos estão na pilha

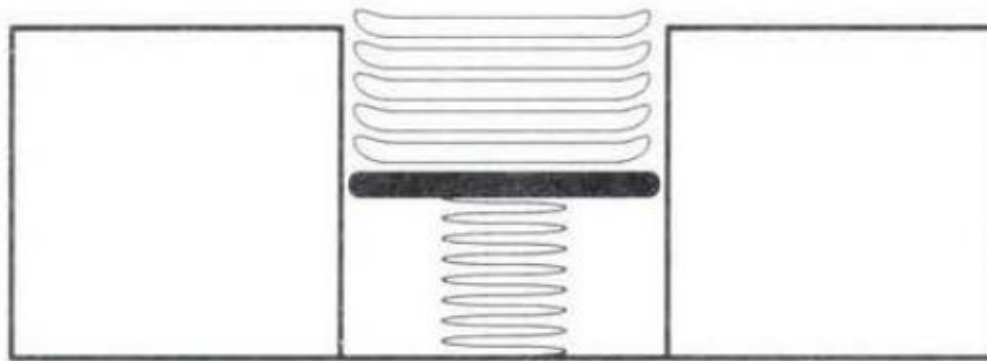
Stack (pilha)



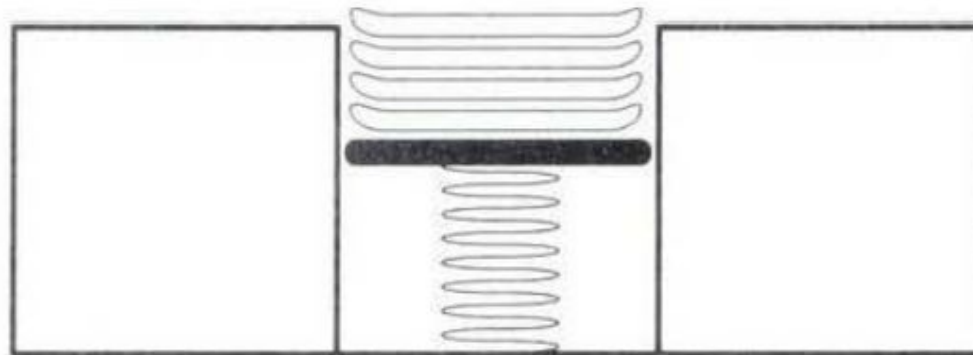
push(prato, pilha)



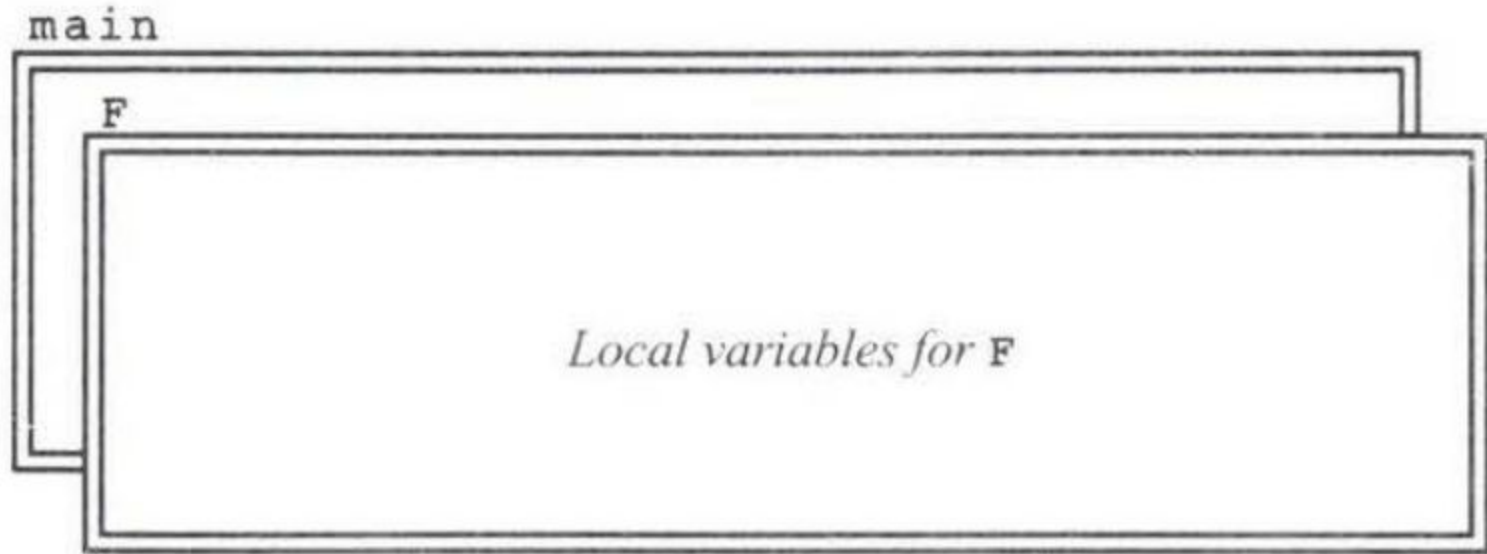
Stack (pilha)



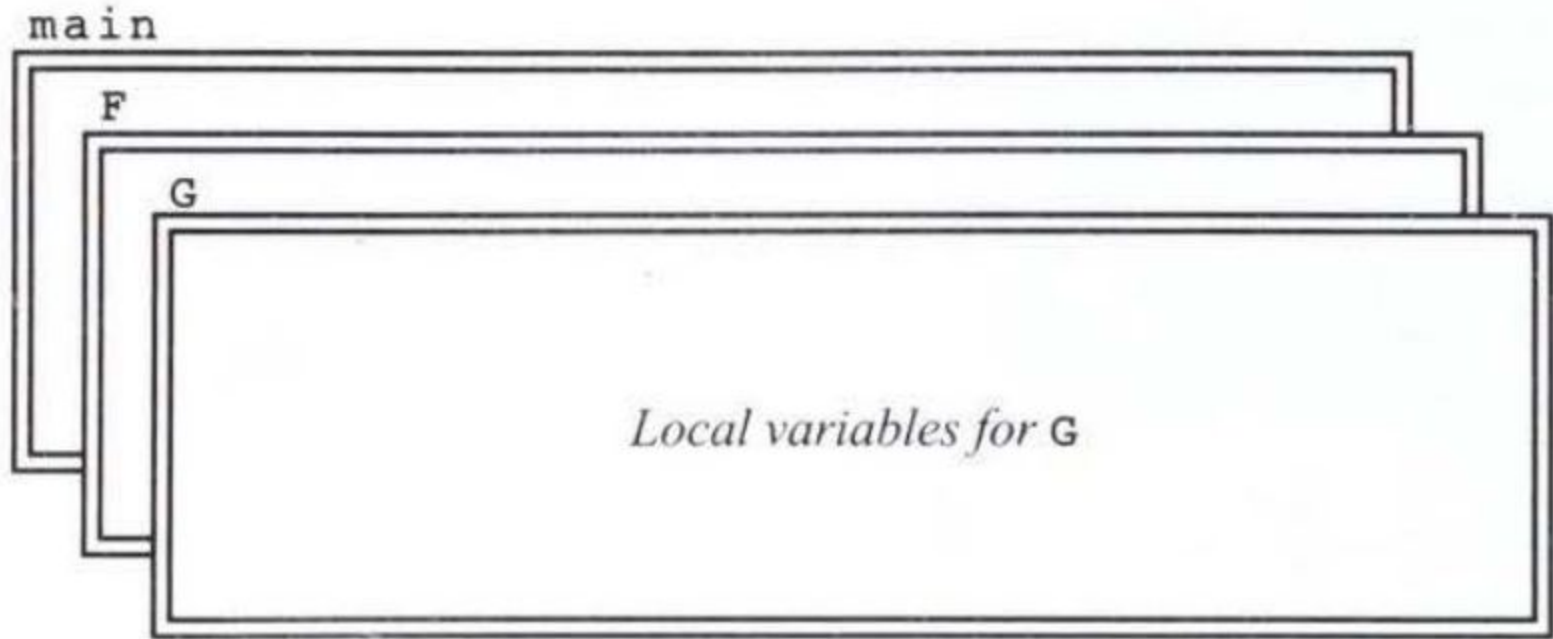
pop(pilha)



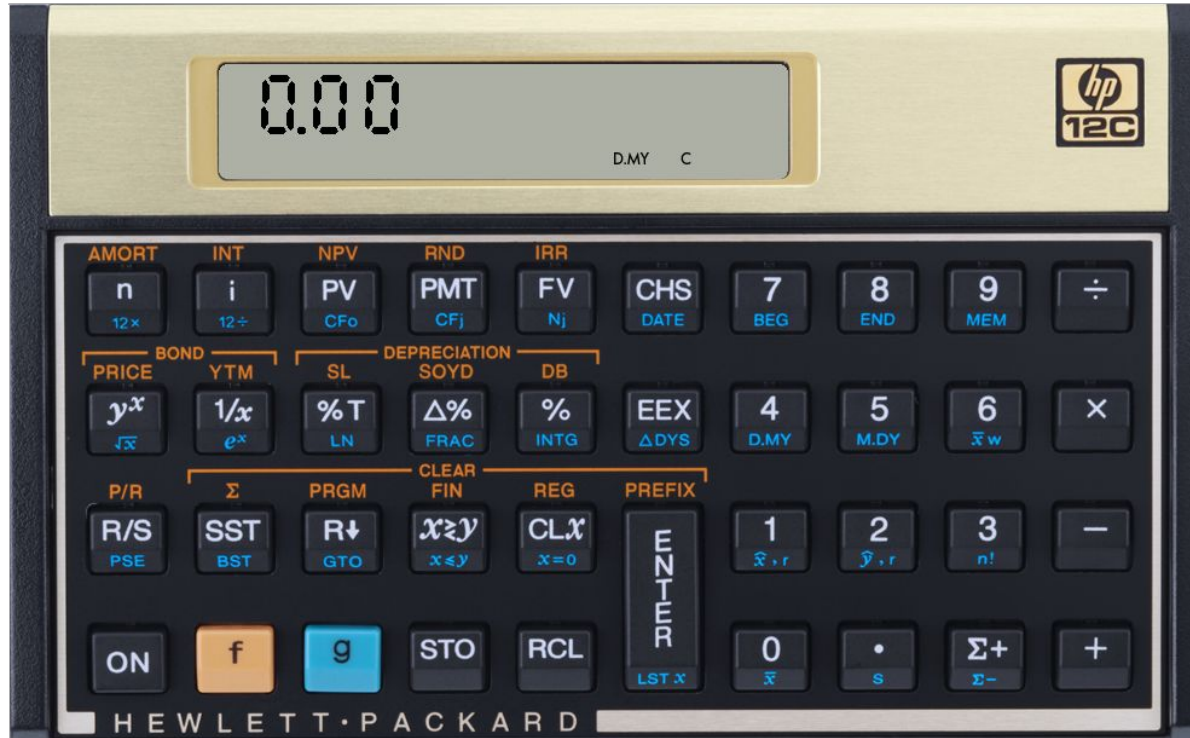
Stack (pilha) e chamadas de função



Stack (pilha) e chamadas de função



Stack (pilha) e calculadoras RPN

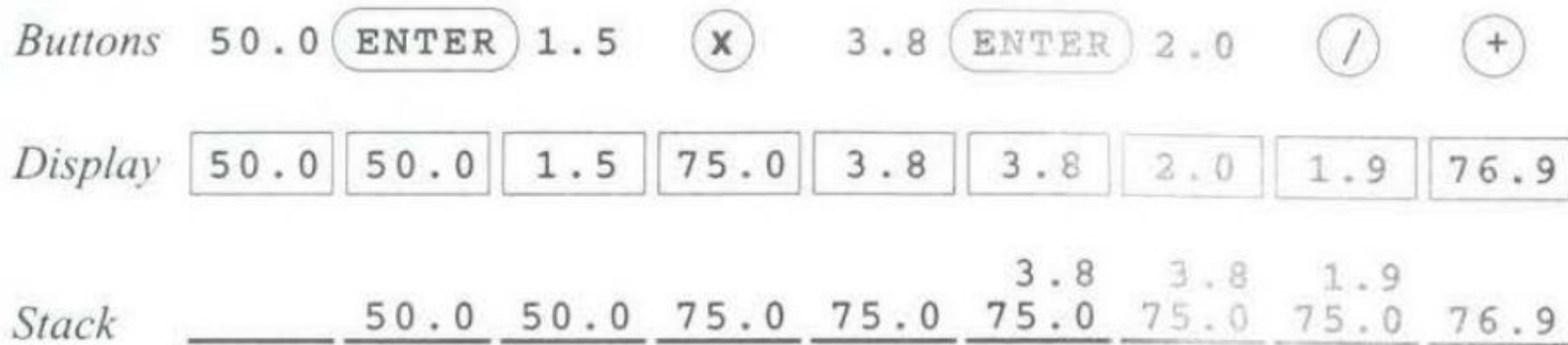


Stack (pilha) e calculadoras RPN

$$50.0 \times 1.5 + \frac{3.8}{2.0}$$

50.0 ENTER 1.5 × 3.8 ENTER 2.0 / +

Stack (pilha) e calculadoras RPN



- Se o usuário digitar ENTER o valor do display é colocado no stack
- Se o usuário digitar algum operador:
 - Se o usuário digitou um valor novo no display, coloca no stack
 - Faz o pop dos dois valores do topo
 - Aplica a operação matemática
 - Faz o push do resultado no stack

Criando um TAD Stack (pilha)

- Como representar um stack? Como criar uma abstração que seja genérica, para uso por diversas aplicações clientes?
 - Precisamos de uma **interface** (por exemplo, stack.h ou stackTAD.h) que exporte:
 - Os **tipos de dados usados no stack**; e
 - Os **subprogramas** que definem seu comportamento.

Criando um TAD Stack (pilha): tipos de dados

- Ao criar um stack existem duas coisas que temos que considerar:
 - Qual o **tipo de dado do elemento** a ser armazenado na pilha
 - Qual o **tipo de dado para a pilha** em si mesma (concreto ou abstrato).
- Para responder, temos que considerar se essas duas coisas estão no **domínio do cliente** ou no **domínio da implementação**.

Criando um TAD Stack (pilha): tipo de dado dos elementos

- O **tipo de dado do elemento** a ser armazenado na pilha está no **domínio do cliente**, ou seja, é a aplicação cliente que determina como a pilha será utilizada. Não importa para a “abstração stack” quais valores estão armazenados, pois as operações e comportamentos do stack serão os mesmos.
- O ideal seria criar um **stack genérico**, que aceita **elementos de qualquer tipo de dado que o cliente precisa**. O problema é que, em C, é necessário especificar um tipo de dado exato, para que as funções utilizem esse tipo de dado como parâmetro. Isso seria possível se a linguagem C tivesse um tipo de dado “**any**” que serviria para qualquer coisa. Isso não existem em C, o mais próximo que conseguimos chegar é o tipo **void ***, que é um tipo **compatível com ponteiros de qualquer tipo** (não com tipos de dados).

Criando um TAD Stack (pilha): tipo de dado dos elementos

- Se o **tipo de dado do elemento** for o tipo `void *`, podemos criar um stack genérico no sentido de que esse stack pode armazenar dados de qualquer tipo de ponteiro:
 - ponteiros para char (string) em um stack
 - ponteiros para inteiros em outro stack
 - ponteiros para diferentes tipos no mesmo stack (a aplicação deve ter informações que permitam saber que tipos estão sendo usados no momento)
- O problema é que essa abordagem, apesar de realmente criar um stack genérico usando ponteiros genéricos, cria um **overhead para a aplicação cliente**, forçando o uso de ponteiros e alocação de memória ao invés do uso dos dados de modo direto.

Criando um TAD Stack (pilha): tipo de dado dos elementos

- Se o **tipo de dado do elemento** for um tipo específico, como int ou double, o cliente não terá overhead usando ponteiros, mas o stack não será genérico (só poderá armazenar elementos desse tipo específico).
- É a solução mais simples (TALVEZ a mais eficiente), mas é a alternativa menos genérica.

Criando um TAD Stack (pilha): tipo de dado dos elementos

- Se o **tipo de dado do elemento** for especificado através de um `typedef` que define um alias para algum tipo específico, podemos **aumentar a flexibilidade da interface**: você pode especificar um tipo padrão genérico, como o `void *`, mas o cliente pode alterar o tipo de dado do elemento conforme a necessidade, se você fornecer o código fonte. Ex.:

```
typedef void * elementoT;
```

- Se o cliente não quiser trabalhar com ponteiros, ele pode remover o `void *` e utilizar um outro tipo de sua preferência. O problema é que essa abordagem **quebra a interface** e **quebra a barreira da abstração**! Você precisará fornecer o código fonte (implementação) para que o cliente recompile a interface.

Criando um TAD Stack (pilha): tipo de dado dos elementos

- Na verdade, em C, não há uma estratégia ótima para definir o **tipo de dado do elemento**, devido às limitações de C:
 - Gostaríamos de que o stack fosse genérico, que pudesse ser utilizado para elementos de qualquer tipo de dado, mas isso complica a interface para o cliente
 - Ao mesmo tempo, se utilizarmos aliases e fornecermos o código para que o cliente compile uma versão apropriada, quebramos a barreira da abstração e comprometemos a estabilidade da interface
- TALVEZ a melhor (?) alternativa seja:
 - Criar uma interface usando **typedef <tipo> elementoT** para criar aliases para algum tipo de dado específico, e fornecer acesso ao código fonte para que o cliente altere o tipo, se necessário; e
 - Exportar uma versão “padrão” da interface com **typedef void * elementoT**, para que a interface seja genérica e prontamente utilizável por clientes que não se importem em utilizar ponteiros.

Criando um TAD Stack (pilha): tipo de dado da pilha

- Uma vez definido o tipo de dado dos elementos (que está no domínio dos clientes), temos que decidir sobre o **tipo de dado para a pilha em si**, o que está no **domínio da implementação**. É você que deve escolher uma representação adequada para o stack (se será concreto ou abstrato) e implementar seu comportamento; depois, é você que deve escolher uma estrutura de dados adequada para armazenar os elementos da pilha.
- Já vimos que um tipo de dado, como o stack, pode ser **concreto** ou **abstrato**, dependendo do grau de acesso do cliente à representação interna: se toda a representação interna do stack está **opaco, escondido**, então temos um tipo abstrato de dado.

Criando um TAD Stack (pilha): tipo de dado da pilha

- Para criar um tipo abstrato de dado em C, na **interface** usaremos esse modelo didático:

```
typedef nomeTCD *nomeTAD;
```

- Termos:
 - TCD: tipo concreto de dado
 - TAD: tipo abstrato de dado
- Na interface nós criamos um TAD que é apenas um PONTEIRO para o TCD (que será especificado na implementação da interface). Exemplo:

```
typedef struct stackTCD *stackTAD;
```

Criando um TAD Stack (pilha): tipo de dado da pilha

- Na linha abaixo, o tipo `stackTAD` é um **ponteiro** para uma estrutura que tem o nome de `stackTCD`, que ainda não foi definida (só será definida durante a implementação da interface). Uma estrutura que ainda não foi definida é chamada de um **tipo incompleto**. Na **interface**:

```
typedef struct stackTCD *stackTAD;
```

- Na **implementação** da interface nós completaremos o tipo:

```
struct stackTCD {  
    . . .  
};
```

Criando um TAD Stack (pilha): tipo de dado da pilha

- As duas definições são “ligadas” pelo identificador **nomeTCD**, que passa a ser conhecido por **structure tag**.

stackTAD.h

```
typedef struct stackTCD *stackTAD;  
  
. . .
```

stackTAD.c

```
struct stackTCD  
{  
    . . .  
};
```

Criando um TAD Stack (pilha): tipos de dados

```
41 /**
42  * TIPO: elementoT
43  * -----
44  * O tipo "elementoT" é utilizado nesta interface para indicar o tipo de dado
45  * dos valores que serão armazenados no stack, ou seja, representa o tipo de
46  * dado dos elementos. Nesta interface, por padrão, o tipo de dado armazenado é
47  * "double", mas isso pode ser alterado ao se editar a linha da definição, a
48  * seguir (é necessário compilar a implementação da interface se esse tipo
49  * padrão for alterado).
50  */
51
52 typedef double elementoT;
53
54 /* Seção 2: tipos abstratos de dados */
55
56 /**
57  * TIPO ABSTRATO: stackTAD
58  * -----
59  * O tipo stackTAD representa um tipo abstrato que é uma pilha, para armazenar
60  * os elementos do tipo "elementoT". Como o stackTAD é definido apenas como um
61  * ponteiro para uma estrutura concreta que não está definida nesta interface,
62  * os clientes não têm acesso à implementação interna.
63  */
64
65 typedef struct stackTCD *stackTAD;
66
```

Criando um TAD Stack (pilha): comportamento

- Agora que discutimos sobre os tipos de dados envolvidos em nossa pilha, precisamos **especificar seu comportamento** (os subprogramas que manipularão a pilha). Alguns comportamentos comuns:
 - `criar_stack`
 - `remover_stack`
 - `push`
 - `pop`
 - `vazia`
 - `cheia`
 - `tamanho`
 - `qtd_elementos`
 - `espaco_restante`
 - `ver_elemento`
 - `ver_pilha`
 - `. . .`

Criando um TAD Stack (pilha): comportamento

```
69  /**
70   * FUNÇÃO: criar_stackTAD
71   * Uso: stackTAD = criar_stackTAD( );
72   * -----
73   * Esta função aloca e retorna um novo stack, que está inicialmente vazio. Se
74   * ocorrer algum erro na criação do stack, será retornado NULL.
75   */
76
77  stackTAD criar_stackTAD (void);
78
79  /**
80   * PROCEDIMENTO: remover_stackTAD
81   * Uso: remover_stackTAD(&stackTAD);
82   * -----
83   * Este procedimento libera o armazenamento alocado para o stackTAD. Note que o
84   * argumento é um PONTEIRO para um stack.
85   */
86
87  void remover_stackTAD (stackTAD *stack);
88
```

Criando um TAD Stack (pilha): comportamento

```
89 /**
90  * PROCEDIMENTO: push
91  * Uso: push(stack, elemento);
92  * -----
93  * Este procedimento coloca o elemento especificado no topo do stack.
94  */
95
96 void push (stackTAD stack, elementoT elemento);
97
98 /**
99  * FUNÇÃO: pop
100  * Uso: elemento = pop(stack);
101  * -----
102  * Esta função retira o elemento que está no topo da pilha e retorna esse
103  * elemento. O valor a ser retirado é sempre o último que entrou na pilha por
104  * último. Se o stack for null ou estiver vazio, é exibida uma mensagem de erro
105  * e o valor retornado será um valor NAN, ou seja, o valor padrão que representa
106  * um "Not a Number" conforme o padrão IEEE-754. Seu programa deve verificar se
107  * o valor retornado é, ou não, um NaN e tomar ações apropriadas:
108  *
109  *     if (isnan(elemento = pop(stack)))
110  *         ... código para tratar o erro do retorno do pop ...
111  *     else
112  *         ... código para prosseguir com seu programa ...
113  *
114  * Obs.: você precisa de math.h para usar a função isnan(x);
115  */
116
117 elementoT pop (stackTAD stack);
118
```

Criando um TAD Stack (pilha): comportamento

```
119 /**
120  * PREDICADOS: vazia, cheia
121  * Uso: if (vazia(stack)) . . .
122  *       if (cheia(stack)) . . .
123  * -----
124  * Estes predicados retornam TRUE caso a pilha esteja vazia ou cheia,
125  * respectivamente. Se o stack for dinâmico, ou seja, sem um tamanho máximo
126  * pré-definido, o predicado "cheia" sempre retornará FALSE.
127  */
128
129 bool vazia (stackTAD stack);
130 bool cheia (stackTAD stack);
131
132 /**
133  * FUNÇÃO: tamanho
134  * Uso: n = tamanho(stack);
135  * -----
136  * Esta função retorna o tamanho do stack, ou seja, quantos elementos este stack
137  * pode armazenar. Três situações especiais podem ocorrer:
138  *
139  *     a) A função retorna normalmente o tamanho do stack;
140  *     b) Ocorre um erro na função: será retornado o valor -1;
141  *     c) Se o stack é dinâmico, sem tamanho máximo, será retornado o valor 0.
142  */
143 long int tamanho (stackTAD stack);
144
```

Criando um TAD Stack (pilha): comportamento

```
145 /**
146  * FUNÇÃO: qtd_elementos
147  * Uso: n = qtd_elementos(stack);
148  * -----
149  * Esta função retorna a quantidade de elementos atualmente dentro da pilha, ou
150  * o valor -1 caso ocorra algum erro.
151  */
152
153 long int qtd_elementos (stackTAD stack);
154
155 /**
156  * FUNÇÃO: espaco_restante
157  * Uso: n = espaco_restante(stack);
158  * -----
159  * Esta função é utilizada geralmente para DEBUG, não é um comportamento padrão
160  * de um TAD pilha. Ela retorna a quantidade de "slots" ainda disponíveis na
161  * pilha. Caso o stack seja dinâmico, ou seja, sem um tamanho máximo
162  * pré-definido, retorna 0. Se houver algum erro, retorna -1.
163  *
164  * TODO: o retorno 0 (zero) pode indicar tanto uma pilha dinâmica sem limite
165  *       pré-definido, ou uma pilha cheia. Isso está errado, é preciso ajustar
166  *       esse comportamento.
167  */
168
169 #ifdef debug
170 long int espaco_restante (stackTAD stack);
171 #endif
172
```

Criando um TAD Stack (pilha): comportamento

```
173 /**
174  * FUNÇÃO: ver_elemento
175  * Uso: elemento = ver_elemento(stack, posicao);
176  * -----
177  * Esta função é apenas para DEBUG, não é comportamento padrão de um TAD pilha.
178  * Ela retorna o elemento especificado em uma posição qualquer da pilha, sem
179  * fazer o pop de nenhum elemento. Se a posição especificada não for uma posição
180  * válida, ou se o stack for null ou estiver vazio, é exibida uma mensagem de
181  * erro e o valor retornado será um valor NAN, ou seja, o valor padrão que
182  * representa um "Not a Number" conforme o padrão IEEE-754. Seu programa deve
183  * verificar se o valor retornado é, ou não, um NaN e tomar ações apropriadas:
184  *
185  *     if (isnan(elemento = ver_elemento(stack, posicao)))
186  *         ... código para tratar o erro ...
187  *     else
188  *         ... código para prosseguir com seu programa ...
189  *
190  * Obs.: você precisa de math.h para usar a função isnan(x);
191  */
192
193 #ifndef debug
194 elementoT ver_elemento (stackTAD stack, int posicao);
195 #endif
196
```

Criando um TAD Stack (pilha): comportamento

```
197 /**
198  * PROCEDIMENTO: imprimir_stack
199  * Uso: imprimir_stack(stack, limite);
200  * -----
201  * Este procedimento é geralmente utilizado para DEBUG, não é comportamento
202  * padrão de um TAD pilha. Ele imprime os elementos atuais da pilha, até uma
203  * quantidade limite (para evitar que o terminal do usuário receba várias e
204  * várias linhas).
205  */
206
207 #ifdef debug
208 void imprimir_stack (stackTAD stack, int limite);
209 #endif
210
```

Criando um TAD Stack (pilha): comportamento

- Após analisar a definição da interface, responda o seguinte:
 - Como os dados serão armazenados?
 - Quais os nomes das variáveis que serão utilizadas para armazenar os dados?
 - Onde os dados serão armazenados?
- Note que o TAD pilha é totalmente especificado pelo seu comportamento! Não sabemos nada da representação interna! É exatamente isso que queremos.

Criando um TAD Stack (pilha): implementação

- E como implementar o TAD stack? Precisamos tomar algumas decisões:
 - **Quantos elementos** a pilha irá suportar?
 - Limite máximo
 - Ilimitado (o limite é a memória disponível)
 - **Que estrutura de dados** utilizaremos para armazenar os elementos?
 - Array
 - Lista encadeada
 - Variáveis

Criando um TAD Stack (pilha): 1ª implementação

- E como implementar o TAD stack? Precisamos tomar algumas decisões:
 - **Quantos elementos** a pilha irá suportar?
 - Limite máximo
 - **Que estrutura de dados** utilizaremos para armazenar os elementos?
 - Array

Criando um TAD Stack (pilha): 1ª implementação

```
38 /**
39  * CONSTANCE: TAMMAX
40  * -----
41  * Esta constante especifica o tamanho máximo de espaço a ser alocado para o
42  * array que armazenará os elementos do stack. Se o usuário fizer um push de
43  * elementos além deste limite, receberá um erro. Se TAMMAX estiver definido
44  * como 0 (zero), indica que o array é dinâmico e não tem tamanho máximo
45  * limitante.
46  */
47
48 #define TAMMAX 1000
49
```

Criando um TAD Stack (pilha): 1ª implementação

```
52  /**
53   * TIPO: stackTCD
54   * -----
55   * O tipo stackTCD é a representação concreta do tipo abstrato de dado stackTAD
56   * definido na interface. Nesta implementação os elementos serão armazenados em
57   * um array. Como a definição do stackTCD aparece apenas na implementação, e não
58   * na interface, podemos alterar esta definição à vontade, desde que a interface
59   * não seja alterada e o comportamento do stack seja mantido. A variável inteira
60   * "contagem" manterá o número atual de elementos no stack.
61   */
62
63  struct stackTCD
64  {
65      elementoT dados[TAMMAX];
66      int contagem;
67  };
68
```

Criando um TAD Stack (pilha): 1ª implementação

```
71 /**
72  * FUNÇÃO: criar_stackTAD
73  * Uso: stackTAD = criar_stackTAD( );
74  * -----
75  * Aloca memória suficiente para um stackTCD e retorna um ponteiro para esse
76  * objeto, através do tipo abstrato stackTAD. Retorna NULL se não for possível
77  * alocar a memória.
78  */
79
80 stackTAD criar_stackTAD (void)
81 {
82     stackTAD S = malloc(sizeof(struct stackTCD));
83     if (S == NULL)
84     {
85         fprintf(stderr, "Erro: não foi possível alocar o stack.\n");
86         return NULL;
87     }
88     S->contagem = 0;
89     return S;
90 }
91
```

Criando um TAD Stack (pilha): 1ª implementação

```
92  /**
93   * PROCEDIMENTO: remover_stackTAD
94   * Uso: remover_stackTAD(&stackTAD);
95   * -----
96   * Ao receber um PONTEIRO para um stackTAD, ou seja, um ponteiro para um
97   * ponteiro para stackTCD, libera a memória alocada para o stackTCD e atribui
98   * NULL para o ponteiro original (para evitar dangling pointer). Se o
99   * ponteiro original já aponta para NULL, não faz nada.
100  */
101
102 void remover_stackTAD (stackTAD *stack)
103 {
104     if (*stack != NULL)
105     {
106         free(*stack);
107         *stack = NULL;
108     }
109 }
110
```

Criando um TAD Stack (pilha): 1ª implementação

```
111 /**
112  * PROCEDIMENTO: push
113  * Uso: push(stack, elemento);
114  * -----
115  * Este procedimento coloca o elemento especificado no topo do stack.
116  */
117
118 void push (stackTAD stack, elementoT elemento)
119 {
120     if (stack == NULL)
121         fprintf(stderr, "Erro: push em stack null.\n");
122     else if (cheia(stack))
123         fprintf(stderr, "Erro: o stack está cheio.\n");
124     else
125         stack->dados[stack->contagem++] = elemento;
126 }
127
```

Criando um TAD Stack (pilha): 1ª implementação

```
128 /**
129  * FUNÇÃO: pop
130  * Uso: elemento = pop(stack);
131  * -----
132  * Retorna o elemento do topo da pilha, ou o valor NAN ("Not a Number") em caso
133  * de erro (situação na qual uma mensagem de erro apropriada também é exibida).
134  */
135
136 elementoT pop (stackTAD stack)
137 {
138     if (stack == NULL)
139     {
140         fprintf(stderr, "Erro: pop em stack null.\n");
141         return NAN;
142     }
143     else if (vazia(stack))
144     {
145         fprintf(stderr, "Erro: stack vazio.\n");
146         return NAN;
147     }
148
149     return stack->dados[--stack->contagem];
150 }
151
```

Criando um TAD Stack (pilha): 1ª implementação

```
152 /**
153  * PREDICADOS: vazia, cheia
154  * Uso: if (vazia(stack)) . . .
155  *      if (cheia(stack)) . . .
156  * -----
157  * Estes predicados retornam TRUE caso a pilha esteja vazia ou cheia,
158  * respectivamente. Se o stack for dinâmico, ou seja, sem um tamanho máximo
159  * pré-definido, o predicado "cheia" sempre retornará FALSE.
160  */
161
162 bool vazia (stackTAD stack)
163 {
164     if (stack == NULL)
165     {
166         printf("Erro: stack null.\n");
167         exit(1);
168     }
169     return (stack->contagem == 0);
170 }
171
172 bool cheia (stackTAD stack)
173 {
174     if (stack == NULL)
175     {
176         printf("Erro: stack null.\n");
177         exit(1);
178     }
179     return (stack->contagem == TAMMAX);
180 }
181
```

Criando um TAD Stack (pilha): 1ª implementação

```
182  /**
183   * FUNÇÃO: tamanho
184   * Uso: n = tamanho(stack);
185   * -----
186   * Retorna o tamanho do stack. Situações:
187   *
188   *     TAMMAX > 0: se tamanho limitado;
189   *     TAMMAX = 0: se tamanho ilimitado;
190   *     -1       : se ocorrer algum erro.
191   */
192
193 long int tamanho (stackTAD stack)
194 {
195     if (stack == NULL)
196     {
197         fprintf(stderr, "Erro: tamanho de stack null.\n");
198         return -1;
199     }
200     return (long int) TAMMAX;
201 }
202
```

Criando um TAD Stack (pilha): 1ª implementação

```
203 /**
204  * FUNÇÃO: qtd_elementos
205  * Uso: n = qtd_elementos(stack);
206  * -----
207  * Esta função retorna a quantidade de elementos atualmente dentro da pilha,
208  * ou -1 se ocorrer algum erro.
209  */
210
211 long int qtd_elementos (stackTAD stack)
212 {
213     if (stack == NULL)
214     {
215         fprintf(stderr, "Erro: qtd_elementos de stack null.\n");
216         return -1;
217     }
218     return (stack->contagem);
219 }
220
```

Criando um TAD Stack (pilha): 1ª implementação

```
221 /**
222  * FUNÇÃO: espaco_restante
223  * Uso: n = espaco_restante(stack);
224  * -----
225  * Retorna a quantidade de espaço restante na pilha. Se a pilha é dinâmica,
226  * retorna 0; se houver algum erro, retorna -1.
227  *
228  * TODO: o retorno 0 (zero) pode indicar tanto uma pilha dinâmica sem limite
229  *        pré-definido, ou uma pilha cheia. Isso está errado, é preciso ajustar
230  *        esse comportamento.
231  */
232
233 #ifdef debug
234 long int espaco_restante (stackTAD stack)
235 {
236     if (stack == NULL)
237     {
238         fprintf(stderr, "Erro: espaco_restante de stack null.\n");
239         return -1;
240     }
241     else if (TAMMAX == 0)
242         return 0;
243
244     return (TAMMAX - stack->contagem);
245 }
246 #endif
247
```

Criando um TAD Stack (pilha): 1ª implementação

```
248 /**
249  * FUNÇÃO: ver_elemento
250  * Uso: elemento = ver_elemento(stack, posicao);
251  * -----
252  * Retorna o elemento que está em uma determinada posição do stack, mesmo que
253  * não seja o topo, SEM fazer nenhum pop do stack. Se houver algum erro é
254  * retornado o valor NAN (Not a Number).
255  */
256
257 #ifdef debug
258 elementoT ver_elemento (stackTAD stack, int posicao)
259 {
260     if (stack == NULL)
261     {
262         fprintf(stderr, "Erro: ver_elemento de stack null.\n");
263         return NAN;
264     }
265     else if (vazia(stack))
266     {
267         fprintf(stderr, "Erro: stack vazio.\n");
268         return NAN;
269     }
270     else if (posicao < 0 || posicao >= stack->contagem)
271     {
272         fprintf(stderr, "Erro: posição inválida.\n");
273         return NAN;
274     }
275
276     return (stack->dados[posicao]);
277 }
```

Criando um TAD Stack (pilha): 1ª implementação

```
280 /**
281  * PROCEDIMENTO: imprimir_stack
282  * Uso: imprimir_stack(stack, limite);
283  * -----
284  * Imprime os elementos da pilha, até um certo limite.
285  */
286
287 #ifdef debug
288 void imprimir_stack (stackTAD stack, int limite)
289 {
290     if (stack == NULL)
291         fprintf(stderr, "Erro: imprimir_stack de stack null.\n");
292     else if (vazia(stack))
293         fprintf(stderr, "Erro: stack está vazio.\n");
294     else if (limite < 0 || limite > stack->contagem)
295         limite = stack->contagem;
296
297     for (int i = 0; i < stack->contagem && i < limite; i++)
298         printf("%g\n", (double) stack->dados[i]);
299 }
300 #endif
```

Criando um TAD Stack (pilha): 1ª implementação

- Nossa 1ª implementação está boa, mas contém algumas coisas que só estão ali apenas para fins didáticos, por exemplo:
 - Impressão de mensagens de erro no terminal
 - Dependência do tipo double
 - Tratamento de erro ruim nos predicados (uso de exit)
 - Tamanho limitado da pilha
- Que outras melhorias você faria na interface e na implementação?

Criando um TAD Stack (pilha): 2ª implementação

- E como implementar o TAD stack? Precisamos tomar algumas decisões:
 - **Quantos elementos** a pilha irá suportar?
 - Sem limite máximo (suportará tantos elementos quanto o tamanho de memória alocada para o programa)
 - **Que estrutura de dados** utilizaremos para armazenar os elementos?
 - Array

Criando um TAD Stack (pilha): 2ª implementação

```
37 /** Constantes Simbólicas: **/
38
39 /**
40  * CONSTANTE: TAMMAX
41  * -----
42  * Esta constante especifica o tamanho máximo de espaço a ser alocado para o
43  * array que armazenará os elementos do stack. Se o usuário fizer um push de
44  * elementos além deste limite, receberá um erro. Se TAMMAX estiver definido
45  * como 0 (zero), indica que o array é dinâmico e não tem tamanho máximo
46  * limitante. Se o array é dinâmico, TAMINI indica o tamanho inicial.
47  */
48
49 #define TAMMAX 0
50 #define TAMINI 2
```

Criando um TAD Stack (pilha): 2ª implementação

```
58 /**
59  * TIPO: stackTCD
60  * -----
61  * O tipo stackTCD é a representação concreta do tipo abstrato de dado stackTAD
62  * definido na interface. Nesta implementação os elementos serão armazenados em
63  * um array. Como a definição do stackTCD aparece apenas na implementação, e não
64  * na interface, podemos alterar esta definição à vontade, desde que a interface
65  * não seja alterada e o comportamento do stack seja mantido. A variável inteira
66  * "contagem" manterá o número atual de elementos no stack, e a variável inteira
67  * "tamanho" manterá o tamanho atual do stack (que pode ser aumentado de forma
68  * dinâmica durante o andamento do programa).
69  */
70
71 struct stackTCD
72 {
73     elementoT *dados;
74     int contagem;
75     int tamanho;
76 };
77
```

Criando um TAD Stack (pilha): 2ª implementação

```
80 /**
81  * FUNÇÃO: criar_stackTAD
82  * Uso: stackTAD = criar_stackTAD( );
83  * -----
84  * Aloca memória suficiente para um stackTCD e retorna um ponteiro para esse
85  * objeto, através do tipo abstrato stackTAD. Retorna NULL se não for possível
86  * alocar a memória.
87  */
88
89 stackTAD criar_stackTAD (void)
90 {
91     stackTAD S = malloc(sizeof(struct stackTCD));
92     if (S == NULL)
93     {
94         fprintf(stderr, "Erro: não foi possível alocar o stack.\n");
95         return NULL;
96     }
97     S->dados = malloc(sizeof(elementoT) * TAMINI);
98     if (S->dados == NULL)
99     {
100         fprintf(stderr, "Erro: impossível alocar array dinâmico para stack.\n");
101         free(S);
102         S = NULL;
103         return NULL;
104     }
105     S->contagem = 0;
106     S->tamanho = TAMINI;
107     return S;
108 }
```

Criando um TAD Stack (pilha): 2ª implementação

```
110 /**
111  * PROCEDIMENTO: remover_stackTAD
112  * Uso: remover_stackTAD(&stackTAD);
113  * -----
114  * Ao receber um PONTEIRO para um stackTAD, ou seja, um ponteiro para um
115  * ponteiro para stackTCD, libera a memória alocada para o stackTCD e atribui
116  * NULL para o ponteiro original (para evitar dangling pointer). Se o
117  * ponteiro original já aponta para NULL, não faz nada.
118  */
119
120 void remover_stackTAD (stackTAD *stack)
121 {
122     if (*stack != NULL)
123     {
124         free((*stack)->dados);
125         (*stack)->dados = NULL;
126         free(*stack);
127         *stack = NULL;
128     }
129 }
```

Criando um TAD Stack (pilha): 2ª implementação

```
131 /**
132  * PROCEDIMENTO: push
133  * Uso: push(stack, elemento);
134  * -----
135  * Este procedimento coloca o elemento especificado no topo do stack. Se o stack
136  * estiver cheio, duplica o tamanho antes da inserção.
137  */
138
139 void push (stackTAD stack, elementoT elemento)
140 {
141     if (stack == NULL)
142         fprintf(stderr, "Erro: push em stack null.\n");
143
144     if (stack->contagem == stack->tamanho)
145         aumentar_stack(stack);
146
147     stack->dados[stack->contagem++] = elemento;
148 }
```

Criando um TAD Stack (pilha): 2ª implementação

```
52 /** Declarações de subprogramas */
53
54 static void aumentar_stack (stackTAD stack);
55
319 /**
320  * PROCEDIMENTO: aumentar_stack
321  * Uso: aumentar_stack(stack);
322  * -----
323  * Este procedimento recebe um stackTAD e dobra o tamanho do array utilizado
324  * para armazenar os elementos. Depois disso, copia todos os elementos do
325  * array antigo e retorna o novo array preenchido e expandido. Não é exportado
326  * na interface. O array anterior é desalocado.
327  */
328
329 static void aumentar_stack (stackTAD stack)
330 {
331     elementoT *temp;
332     int novo_tamanho = stack->tamanho * 2;
333
334     temp = realloc(stack->dados, sizeof(elementoT) * novo_tamanho);
335     if (temp == NULL)
336     {
337         fprintf(stderr, "Erro: impossível aumentar stack!\n");
338         exit(1);
339     }
340
341     stack->dados = temp;
342     stack->tamanho = novo_tamanho;
343 }
```

Criando um TAD Stack (pilha): 2ª implementação

```
194 bool cheia (stackTAD stack)
195 {
196     return FALSE;
197 }
```

Criando um TAD Stack (pilha): comparação da implement.

- Quando implementamos a interface `stackTAD.h`, o que mudou de uma implementação para a outra?
- O que mudou na interface?

Criando um TAD Stack (pilha): comparação da implement.

- Quando implementamos a interface `stackTAD.h`, o que mudou de uma implementação para a outra?
 - Quantidade máxima de elementos!
- O que mudou na interface?
 - Nada!

Uso clássico de TAD Stack: calculadora RPN

```
[abrantestasf@ideapad ~/ed1/cap08]$ rpncalc
Simulação de Calculadora RPN (digite A para ajuda)
> 50
> 1.5
> *
75
> 3.8
> 2
> /
1.9
> +
76.9
> 100
> 30
> m
Pilha:
76.9
100
30
> +
130
> m
Pilha:
76.9
130
> -
-53.1
> s
```

```
[abrantestasf@ideapad ~/ed1/cap08]$ █
```

Quando usar um TAD pilha?

- Em qualquer situação onde você precise de comportamento LIFO.
 - Ex.: calculadora RPN
- Para substituir programas que utilizam estado encapsulado.
 - São programas que **utilizam variáveis globais para manter o estado** entre as chamadas de funções, geralmente inicializadas como estáticas.
 - **Estado encapsulado** é como chamamos os **dados que estão mantidos privadamente dentro de um módulo através de variáveis globais**. Sinônimo: estado interno.
 - **LIMITAÇÕES:**
 - só mantém uma única cópia do estado disponível em qualquer instante de tempo
 - não podem ser utilizados em dois “locais” ao mesmo tempo
 - abstrações clientes impedem o uso do programa original

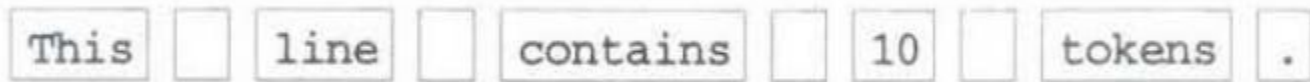
Importante de uso de TADs: remover estado encapsulado

- Um uso muito importante de TADs é na substituição de programas que utilizam estado encapsulado.
 - São programas que **utilizam variáveis globais para manter o estado** entre as chamadas de funções, geralmente inicializadas como estáticas.
 - **Estado encapsulado** é como chamamos os **dados que estão mantidos privadamente dentro de um módulo através de variáveis globais**. Sinônimo: estado interno.
 - **LIMITAÇÕES:**
 - só mantém uma única cópia do estado disponível em qualquer instante de tempo
 - não podem ser utilizados em dois “locais” ao mesmo tempo
 - abstrações clientes impedem o uso do programa original

Ex. de problemas com estado encapsulado: scanner

- Um **scanner** é um programa que “lê” textos sem prestar atenção aos caracteres individuais: ele agrupa letras para formar palavras que são reconhecidas como unidades independentes, dividindo a string em seus **tokens** componentes.
- Um token é uma seqüência de caracteres que forma uma unidade coerente:
 - Uma seqüência de caracteres alfanuméricos consecutivos (letras ou números); OU
 - Uma string de apenas um caractere contendo um espaço; OU
 - Uma string de apenas um caractere contendo um sinal de pontuação.

This line contains 10 tokens.



Ex. de problemas com estado encapsulado: scanner.h

```
62  /**
63   * PROCEDIMENTO: iniciar_scanner
64   * Uso: iniciar_scanner(string);
65   * -----
66   * Este procedimento inicializa o scanner e o configura de modo a permitir que
67   * ele leia tokens a partir de uma linha (string). Após a inicialização do
68   * scanner, a chamada da função "obter_proximo_token" retornará uma string que
69   * contém o primeiro token da linha; a próxima chamada da função
70   * "obter_proximo_token" retornará a string que contém o segundo token da linha
71   * e assim por diante.
72   */
73
74 void iniciar_scanner (string linha);
75
```

Ex. de problemas com estado encapsulado: scanner.h

```
76 /**
77  * FUNÇÃO: obter_proximo_token
78  * Uso: str = obter_proximo_token( );
79  * -----
80  * Esta função retorna o próximo token da linha.
81  */
82
83 string obter_proximo_token (void);
84
85 /**
86  * PREDICADO: final_da_linha
87  * Uso: if(final_da_linha( )) . . .
88  * -----
89  * Este predicado retorna TRUE se o scanner alcançou o final da linha.
90  */
91
92 bool final_da_linha (void);
93
```

Ex. de problemas com estado encapsulado: scanner.c

```
27  /** Estado Interno do Scanner: **/
28
29  static string buffer;
30  static int tamanho_buffer;
31  static int posicao;
32
```

Ex. de problemas com estado encapsulado: scanner.c

```
35 /**
36  * PROCEDIMENTO: iniciar_scanner
37  * Uso: iniciar_scanner(string);
38  * -----
39  * Este procedimento apenas inicializa as variáveis do estado interno do scanner
40  * (o estado encapsulado).
41  */
42
43 void iniciar_scanner (string linha)
44 {
45     buffer = linha;
46     tamanho_buffer = StringLength(linha);
47     posicao = 0;
48 }
49
```

Ex. de problemas com estado encapsulado: scanner.c

```
50 /**
51  * FUNÇÃO: obter_proximo_token
52  * Uso: str = obter_proximo_token( );
53  * -----
54  * Se o próximo caractere é alfanumérico (letra ou dígito), a função continua
55  * lendo para encontrar um "pedaço" contínuo de caracteres e retorna esse token.
56  * Se o caractere atual não for alfanumérico (se for um espaço ou pontuação),
57  * uma string contendo apenas esse caractere é retornada.
58  */
59
60 string obter_proximo_token (void)
61 {
62     char c;
63     int inicio;
64
65     if (posicao >= tamanho_buffer)
66         Error("Não há mais tokens.");
67
68     c = IthChar(buffer, posicao);
69     if (isalnum(c))
70     {
71         inicio = posicao;
72         while (posicao < tamanho_buffer && isalnum(IthChar(buffer, posicao)))
73             posicao++;
74         return SubString(buffer, inicio, posicao - 1);
75     }
76     else
77     {
78         posicao++;
79         return CharToString(c);
80     }
81 }
```

Ex. de problemas com estado encapsulado: scanner.c

```
83  /**
84   * PREDICADO: final_da_linha
85   * Uso: if(final_da_linha( )) . . .
86   * -----
87   * Compara se a posição atual do buffer com o comprimento.
88   */
89
90 bool final_da_linha (void)
91 {
92     return posicao >= tamanho_buffer;
93 }
```

Ex. de problemas com estado encapsulado: teste

```
62     string frase1 = "o rato roeu a roupa do rei de roma.";
63     string frase2 = "THE DESIGN OF THE UNIX OPERATING SYSTEM.";
64
65     iniciar_scanner(frase1);
66     printf("Frase 1: %s\n", frase1);
67     printf("    Token: %s\n", obter_proximo_token());
68     printf("    Token: %s\n", obter_proximo_token());
69     printf("    Token: %s\n", obter_proximo_token());
70     printf("\n");
71
72     iniciar_scanner(frase2);
73     printf("Frase 2: %s\n", frase2);
74     printf("    Token: %s\n", obter_proximo_token());
75     printf("    Token: %s\n", obter_proximo_token());
76     printf("    Token: %s\n", obter_proximo_token());
77     printf("\n");
78
79     printf("Frase 1: %s\n", frase1);
80     printf("    Token: %s\n", obter_proximo_token());
81     printf("    Token: %s\n", obter_proximo_token());
82     printf("    Token: %s\n", obter_proximo_token());
83     printf("\n");
```

Ex. de problemas com estado encapsulado: scanner

```
[abrantestasf@ideapad ~/ed1/cap08]$ ./scanner_encapsulado
```

Frase 0: Este e um teste!

Token: Este

Token:

Token: e

Token:

Token: um

Token:

Token: teste

Token: !

OK, scanner funcionando!

Frase 1: o rato roeu a roupa do rei de roma.

Token: o

Token:

Token: rato

OK, scanner funcionando!

Frase 2: THE DESIGN OF THE UNIX OPERATING SYSTEM.

Token: THE

Token:

Token: DESIGN

OK, scanner funcionando!

Frase 1: o rato roeu a roupa do rei de roma.

Token:

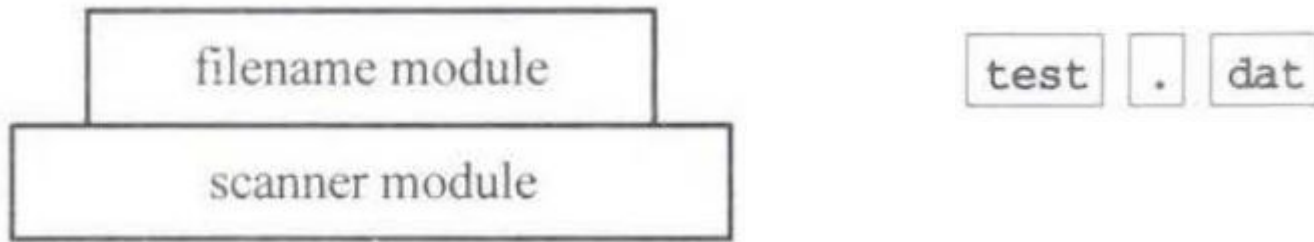
Token: OF

Token:

ERRO!!!

Ex. de problemas com estado encapsulado: scanner

- O problema não se limita a tentar utilizar 2 instâncias do scanner com estado encapsulado: se você criar outra abstração usando o scanner, também teremos a perda do estado!
 - Por exemplo: você pode querer usar o scanner para uma função que retorna os tokens de um nome de arquivo. Abstrações “em camadas” são geralmente úteis.



- Mas, ao basear a leitura do nome do arquivo em um scanner com estado encapsulado, você não pode usar as duas abstrações ao mesmo tempo.
- Em geral, cuidado com variáveis globais para manter estado, mesmo que estáticas.

Solução para o problema do estado encapsulado: TADs

- Uma das soluções possíveis para o problema do estado encapsulado é utilizar TADs, pois:
 - **o tipo abstrato armazena o estado**
 - **nenhuma informação de estado é mantida globalmente**
 - **várias instâncias do tipo de dado podem armazenar estados diferentes ao mesmo tempo, sem que uma instância interfira na outra**

TAD para scanner: scannerTAD.h

```
77  /** Inicia Boilerplate da Interface */
78
79  #ifndef _SCANNERTAD_H
80  #define _SCANNERTAD_H
81
82  /** Includes */
83
84  #include "genlib.h"
85
86  /** Tipos de Dados */
87
88  /**
89   * TIPO: scannerTAD
90   * -----
91   * Este é um tipo abstrato de dado utilizado para representar uma instância
92   * individual de um scanner. Como qualquer outro TAD, os detalhes da estrutura
93   * interna estão ocultos do cliente.
94   */
95
96  typedef struct scannerTCD *scannerTAD;
97
```

TAD para scanner: scannerTAD.h

```
101  /**
102   * FUNÇÃO: crar_scanner
103   * Uso: scanner = criar_scanner( );
104   * -----
105   * Esta função cria uma nova instância de um scanner. Todos os outros
106   * subprogramas desta interface receberão o scanner criado como argumento,
107   * para identificar qual instância particular do scanner o cliente está
108   * utilizando. Isso permite que um mesmo cliente utilize simultaneamente
109   * diversos scanners ao mesmo tempo.
110   */
111
112 scannerTAD criar_scanner (void);
113
```

TAD para scanner: scannerTAD.h

```
114 /**
115  * PROCEDIMENTO: remover_scanner
116  * Uso: remover_scanner(&scanner);
117  * -----;
118  * Este procedimento remove uma instância de um scanner e libera todas as
119  * estruturas de memória associadas a ele. Note que este procedimento precisa
120  * receber um PONTEIRO para um scanner.
121  */
122
123 void remover_scanner(scannerTAD *scanner);
```

TAD para scanner: scannerTAD.h

```
125 /**
126  * PROCEDIMENTO: ler_string
127  * Uso: ler_string(scanner, str);
128  * -----
129  * Este procedimento inicializa a instância do scanner e faz com que essa
130  * instância receba uma string (str) a ser processada para a extração dos
131  * tokens.
132  */
133
134 void ler_string (scannerTAD scanner, string str);
```

TAD para scanner: scannerTAD.h

```
136 /**
137  * FUNÇÃO: obter_token
138  * Uso: token = obter_token(scanner);
139  * -----
140  * Esta função retorna o próximo token lido pelo scanner. Se não houver mais
141  * nenhum token disponível, é retornada a string vazia "". O token retornado
142  * por esta função é sempre alocado na Heap, o que significa que os clientes
143  * devem liberar os tokens quando eles não forem mais necessários.
144  */
145
146 string obter_token (scannerTAD scanner);
```

TAD para scanner: scannerTAD.h

```
148 /**
149  * PROCEDIMENTO: liberar_token
150  * Uso: liberar_token(&token);
151  * -----
152  * Este procedimento recebe um PONTEIRO para um token armazenado na Heap e
153  * libera essa área de memória.
154  */
155
156 void liberar_token (string *token);
```

TAD para scanner: scannerTAD.h

```
158 /**
159  * PREDICADO: existe_outro_token
160  * Uso: if (existe_outro_token( )) . . .
161  * -----
162  * Este predicado retorna TRUE enquanto existirem tokens adicionais para o
163  * scanner ler e retornar.
164  */
165
166 bool existe_outro_token (scannerTAD scanner);
167
168 /**
169  * PROCEDIMENTO: salvar_token
170  * Uso: salvar_token(scanner, token);
171  * -----
172  * Este procedimento armazena o token na estrutura de dados do scanner de tal
173  * forma que, na próxima vez que obter_token for chamado, esse token armazenado
174  * será retornado sem que nenhuma leitura adicional dos caracteres da string
175  * lida pelo scanner.
176  */
177
178 void salvar_token (scannerTAD scanner, string token);
```

TAD para scanner: scannerTAD.h

```
181  * PROCEDIMENTO: configurar_tratamento_de_espacos
182  * Uso: configurar_tratamento_de_espacos(scanner, opcao);
183  * -----
184  * Este procedimento controla se o scanner ignorará caracteres em branco ou se
185  * irá considerá-los tokens válidos. Por padrão o scanner trata caracteres em
186  * branco (espaços, tabs, etc.) como qualquer outro caractere de pontuação e
187  * retorna tokens para esses caracteres. Em algumas aplicações (parsers,
188  * compiladores, etc.) isso não é desejável. Para fazer com que o scanner
189  * ignore caracteres em branco:
190  *
191  *     configurar_tratamento_de_espacos(scanner, 1);
192  *
193  * Para fazer com que o scanner volte a considerar os espaços em branco como
194  * tokens válidos, faça:
195  *
196  *     configurar_tratamento_de_espacos(scanner, 0);
197  *
198  * As opções para o controle do tratamento de espaços são representadas por
213 void configurar_tratamento_de_espacos (scannerTAD scanner, int opcao);
```

TAD para scanner: scannerTAD.h

```
215 /**
216  * FUNÇÃO: obter_tratamento_de_espacos
217  * Uso: opcao = obter_tratamento_de_espacos(scanner);
218  * -----
219  * Esta função retorna a opção atual de tratamento de espaços configurada para o
220  * scanner.
221  */
222
223 int obter_tratamento_de_espacos (scannerTAD scanner);
224
225 /** Finaliza Boilerplate da Interface */
226
227 #endif
```

TAD para scanner: scannerTAD.c

```
21  /** Includes **/
22
23  #include <ctype.h>
24  #include "genlib.h"
25  #include "scannerTAD.h"
26  #include <stdio.h>
27  #include <stdlib.h>
28  #include "strlib.h"
29
30  /** Declarações de Subprogramas Privados **/
31
32  static void pular_espacos (scannerTAD scanner);
33  static int achar_final_do_token (scannerTAD scanner);
34
```

TAD para scanner: scannerTAD.c

```
37 /**
38  * TIPO: scannerTCD
39  * -----
40  * Esta struct é a representação concreta do tipo abstrato scannerTAD, exportado
41  * pela interface. O objetivo deste tipo de dado é manter o estado do scanner
42  * entre as chamadas ao scanner. Os membros da struct são:
43  *
44  *      str          -- cópia da string passada ao scanner
45  *      tam          -- tamanho da string
46  *      pa           -- posição do caractere atual na string
47  *      token_salvo  -- token salvo pelo cliente (NULL se não houver)
48  *      opcao_de_espacos -- opção para ignorar/considerar espaços como tokens
49  */
50
51 struct scannerTCD
52 {
53     string str;
54     int tam;
55     int pa;
56     string token_salvo;
57     int opcao_de_espacos;
58 };
```

TAD para scanner: scannerTAD.c

```
62  /**
63   * FUNÇÃO: criar_scanner
64   * Uso: scanner = criar_scanner( );
65   * -----
66   * Cria uma instância do scanner. Retorna NULL se erro.
67   */
68
69 scannerTAD criar_scanner (void)
70 {
71     scannerTAD S = malloc(sizeof(struct scannerTCD));
72     if (S == NULL)
73     {
74         fprintf(stderr, "Erro: impossível criar o scanner.\n");
75         return NULL;
76     }
77     S->str = NULL;
78     S->tam = 0;
79     S->pa = 0;
80     S->token_salvo = NULL;
81     S->opcao_de_espacos = 0;
82     return S;
83 }
```

TAD para scanner: scannerTAD.c

```
85  /**
86   * PROCEDIMENTO: remover_scanner
87   * Uso: remover_scanner(&scanner);
88   * -----
89   * Recebe um PONTEIRO para um scannerTAD (um ponteiro para ponteiro para
90   * scannerTCD) e libera todas as estruturas de memória alocadas.
91   */
92
93 void remover_scanner(scannerTAD *scanner)
94 {
95     if (*scanner != NULL)
96     {
97         free((*scanner)->str);
98         free((*scanner)->token_salvo);
99         free(*scanner);
100         *scanner = NULL;
101     }
102 }
```

TAD para scanner: scannerTAD.c

```
104 /**
105  * PROCEDIMENTO: ler_string
106  * Uso: ler_string(scanner, str);
107  * -----
108  * Aloca a string str para o scannerTAD scanner, fazendo uso das funções da
109  * biblioteca CSLIB, e "zerando" a condição inicial do scanner (exceto a
110  * opção de tratamento de espaços).
111  *
112  * A alocação da string passada pelo usuário é feita através da função
113  * CopyString, para preservar a separação entre o cliente e a implementação.
114  * Como a implementação não tem nenhum controle sobre o quê o usuário passará
115  * como string, para evitar receber um ponteiro para str (que pode ser alterado
116  * pelo usuário a qualquer momento) fazemos uma cópia da string. Nesse caso o
117  * usuário pode alterar a string original, sem causar efeito no scanner (nesse
118  * caso, se o usuário alterar a string original, será forçado a passar a nova
119  * string para o scanner).
120  */
121
122 void ler_string (scannerTAD scanner, string str)
123 {
124     if (scanner == NULL)
125     {
126         fprintf(stderr, "Erro: impossível atribuir string a scanner NULL.\n");
127         exit(1);
128     }
129
130     if (scanner->str != NULL)
131     {
132         free(scanner->str);
133         if (scanner->token_salvo != NULL)
134             free(scanner->token_salvo);
135     }
136
137     scanner->str = CopyString(str);
138     scanner->tam = StringLength(str);
139     scanner->pa = 0;
140     scanner->token_salvo = NULL;
141 }
```

TAD para scanner: scannerTAD.c

```
143 /**
144  * FUNÇÃO: obter_token
145  * Uso: token = obter_token(scanner);
146  * -----
147  * Retorna o próximo token da string armazenada no scannerTAD. Se já houver um
148  * token salvo, retorna o token salvo. Se não houver mais nenhum token a ser
149  * lido, retorna a string vazia.
150  *
151  * No caso da função retornar a string vazia, também é utilizada a função
152  * CopyString. Por quê? Por dois motivos principais:
153  *
154  * 1. Proteger a implementação de clientes descuidados ou maliciosos.
155  *    Retornar um ponteiro para memória alocada em seu próprio domínio
156  *    é uma falha de segurança pois permite ao cliente sobrescrever o
157  *    valor apontado por esse ponteiro.
158  *
159  * 2. Para permitir que o cliente passar utilizar o procedimento
160  *    liberar_token em qualquer token retornado por esta função. Note que
161  *    esta função utilizar a SubString para retornar um token, e essa função
162  *    sempre aloca um novo espaço de armazenamento (fora do domínio desta
163  *    implementação). Alocando um espaço também para a string vazia
164  *    garantimos que TODOS os tokens que podem ser retornados por esta
165  *    função estão alocados da mesma maneira e podem ser liberados da mesma
166  *    maneira (com a função liberar_token).
167  *
168  */
```

```
150 string obter_token (scannerTAD scanner)
151 {
152     char c;
153     string token;
154     int inicio, fim;
155
156     if (scanner == NULL)
157     {
158         fprintf(stderr, "Erro: lendo token de scanner NULL.\n");
159         exit(1);
160     }
161     else if (scanner->str == NULL)
162     {
163         fprintf(stderr, "Erro: scanner não recebeu linha.\n");
164         exit(1);
165     }
166     else if (scanner->token_salvo != NULL)
167     {
168         token = scanner->token_salvo;
169         scanner->token_salvo = NULL;
170         return token;
171     }
172
173     if (scanner->opcao_de_espacos == 1)
174         pular_espacos(scanner);
175
176     inicio = fim = scanner->pa;
177     if (inicio >= scanner->tam)
178         return CopyString("");
179
180     c = scanner->str[scanner->pa];
181     if (isalnum(c))
182         fim = achar_final_do_token(scanner);
183     else
184         scanner->pa++;
185
186     return SubString(scanner->str, inicio, fim);
187 }
```

TAD para scanner: scannerTAD.c

```
209 /**
210  * PROCEDIMENTO: liberar_token
211  * Uso: liberar_token(&token);
212  * -----
213  * Este procedimento recebe um PONTEIRO para um token (um ponteiro para um
214  * ponteiro para char) e libera a memória alocada para o token.
215  */
216
217 void liberar_token (string *token)
218 {
219     if (*token != NULL)
220     {
221         free(*token);
222         *token = NULL;
223     }
224 }
```

TAD para scanner: scannerTAD.c

```
226 /**
227  * PREDICADO: existe_outro_token
228  * Uso: if (existe_outro_token( )) . . .
229  * -----
230  * Retorna TRUE se ainda houver algum token para ser retornado.
231  */
232
233 bool existe_outro_token (scannerTAD scanner)
234 {
235     if (scanner == NULL)
236     {
237         fprintf(stderr, "Erro: scanner NULL.\n");
238         exit(1);
239     }
240     else if (scanner->str == NULL)
241     {
242         fprintf(stderr, "Erro: scanner não recebeu linha.\n");
243         exit(1);
244     }
245
246     if (scanner->token_salvo != NULL)
247         return (!StringEqual(scanner->token_salvo, ""));
248
249     if (scanner->opcao_de_espacos == 1)
250         pular_espacos(scanner);
251
252     return (scanner->pa < scanner->tam);
253 }
```

TAD para scanner: scannerTAD.c

```
255  /**
256   * PROCEDIMENTO: salvar_token
257   * Uso: salvar_token(scanner, token);
258   * -----
259   */
260
261 void salvar_token (scannerTAD scanner, string token)
262 {
263     if (scanner == NULL)
264     {
265         fprintf(stderr, "Erro: scanner NULL.\n");
266         exit(1);
267     }
268     else if (scanner->str == NULL)
269     {
270         fprintf(stderr, "Erro: scanner não recebeu linha.\n");
271         exit(1);
272     }
273     else if (scanner->token_salvo != NULL)
274     {
275         fprintf(stderr, "Erro: já existe um token salvo.\n");
276         exit(1);
277     }
278
279     scanner->token_salvo = token;
280 }
```

TAD para scanner: scannerTAD.c

```
282 /**
283  * PROCEDIMENTO: configurar_tratamento_de_espacos
284  * Uso: configurar_tratamento_de_espacos(scanner, opcao);
285  * -----
286  * As opções para o controle do tratamento de espaços são representadas por
287  * números inteiros:
288  *
289  *     0 = considerar os espaços em branco como tokens
290  *     1 = ignorar os espaços em branco
291
292  * Atenção: qualquer valor diferente de 0 ou 1 é entendido como inválido e,
293  * nesse caso, o scanner volta para seu comportamento padrão (considerar os
294  * espaços como tokens válidos).
295  */
296
297 void configurar_tratamento_de_espacos (scannerTAD scanner, int opcao)
298 {
299     if (scanner == NULL)
300     {
301         fprintf(stderr, "Erro: scanner NULL.\n");
302         exit(1);
303     }
304
305     if (opcao == 1)
306         scanner->opcao_de_espacos = 1;
307     else
308         scanner->opcao_de_espacos = 0;
309 }
```

TAD para scanner: scannerTAD.c

```
311  /**
312   * FUNÇÃO: obter_tratamento_de_espacos
313   * Uso: opcao = obter_tratamento_de_espacos(scanner);
314   * -----
315   */
316
317 int obter_tratamento_de_espacos (scannerTAD scanner)
318 {
319     if (scanner == NULL)
320     {
321         fprintf(stderr, "Erro: scanner NULL.\n");
322         exit(1);
323     }
324
325     return scanner->opcao_de_espacos;
326 }
```

TAD para scanner: scannerTAD.c

```
328 /** Definições de Subprogramas Privados */
329
330 /**
331  * PROCEDIMENTO: pular_espacos
332  * Uso: pular_espacos(scanner);
333  * -----
334  * Este procedimento "pula" os espaços em branco em uma string, avançando a
335  * posição atual até que ela não esteja sob um caractere branco.
336  */
337
338 static void pular_espacos (scannerTAD scanner)
339 {
340     while (isspace(scanner->str[scanner->pa]))
341         scanner->pa++;
342 }
```

TAD para scanner: scannerTAD.c

```
344 /**
345  * FUNÇÃO: achar_final_do_token
346  * Uso: final = achar_final_do_token(scanner);
347  * -----
348  * Esta função avança a posição atual do scanner até que ela alcance o final de
349  * uma sequência de letras ou dígitos que forma um identificador (token). O
350  * valor de retorno é a posição (índice) do último caractere no identificador;
351  * o valor da posição atual que ficará no scanner será o primeiro caractere
352  * posterior ao retorno.
353  */
354
355 static int achar_final_do_token (scannerTAD scanner)
356 {
357     while (isalnum(scanner->str[scanner->pa]))
358         scanner->pa++;
359
360     return (scanner->pa - 1);
361 }
```

TAD para scanner: cliente

```
34 // Frases sem acentuação, nosso scanner não lida bem com isso!
35 string frase0 = "Este e um teste!";
36 string frase1 = "o rato roeu a roupa do rei de roma.";
37 string frase2 = "THE DESIGN OF THE UNIX OPERATING SYSTEM.";
38
39 // Iniciar scanners para ler as frases:
40 scannerTAD scanner0 = criar_scanner();
41 ler_string(scanner0, frase0);
42
43 scannerTAD scanner1 = criar_scanner();
44 ler_string(scanner1, frase1);
45
46 scannerTAD scanner2 = criar_scanner();
47 configurar_tratamento_de_espacos(scanner2, 1);
48 ler_string(scanner2, frase2);
49
50 // Token obtido:
51 string token;
52
```

TAD para scanner: cliente

```
53 // Imprimir os tokens:
54 printf("Frase 0: %s\n", frase0);
55 for (int i = 0; i < 8; i++)
56 {
57     token = obter_token(scanner0);
58     printf("    Token: %s\n", token);
59     liberar_token(&token);
60 }
61 printf("\n");
62
63 printf("Frase 1: %s\n", frase1);
64 for (int i = 0; i < 3; i++)
65 {
66     token = obter_token(scanner1);
67     printf("    Token: %s\n", token);
68     liberar_token(&token);
69 }
70 printf("\n");
71
72 printf("Frase 2: %s\n", frase2);
73 for (int i = 0; i < 3; i++)
74 {
75     token = obter_token(scanner2);
76     printf("    Token: %s\n", token);
77     liberar_token(&token);
78 }
79 printf("\n");
80
```

```
81 printf("Frase 1: %s\n", frase1);
82 for (int i = 0; i < 3; i++)
83 {
84     token = obter_token(scanner1);
85     printf("    Token: %s\n", token);
86     liberar_token(&token);
87 }
88 printf("\n");
89
90 printf("Frase 2: %s\n", frase2);
91 for (int i = 0; i < 3; i++)
92 {
93     token = obter_token(scanner2);
94     printf("    Token: %s\n", token);
95     liberar_token(&token);
96 }
97 printf("\n");
98
99 // Remover os scanners:
100 remover_scanner(&scanner0);
101 remover_scanner(&scanner1);
102 remover_scanner(&scanner2);
103 }
```

TAD para scanner: cliente

```
[abrantesasf@ideapad ~/ed1/cap08]$ ./scanner_nao_encapsulado
```

Frase 0: Este e um teste!

Token: Este

Token:

Token: e

Token:

Token: um

Token:

Token: teste

Token: !

Frase 1: o rato roeu a roupa do rei de roma.

Token: o

Token:

Token: rato

Frase 2: THE DESIGN OF THE UNIX OPERATING SYSTEM.

Token: THE

Token: DESIGN

Token: OF

Frase 1: o rato roeu a roupa do rei de roma.

Token:

Token: roeu

Token:

Frase 2: THE DESIGN OF THE UNIX OPERATING SYSTEM.

Token: THE

Token: UNIX

Token: OPERATING

Em resumo

- Diferença entre TCD e TAD
- Estruturas de dados
- TADs x Estruturas de Dados
- TADs x Estado Encapsulado
- TAD Stack
 - genérica (void *) x não genérica
- TAD Scanner
 - Obter tokens (importante para compiladores!)