

# Estrutura de Dados Básica

Daniel de Sousa Moraes  
danielmoraes14@gmail.com

# Vetores e Matrizes

- Vetor
  - Conjunto finito e limitado de elementos homogêneos.
- Forma de definição
  - Nome
  - Tipo
  - Tamanho
  - Limite final e Limite inicial
- Forma de acesso
  - Armazenamento e recuperação de qualquer posição dentro do vetor em qualquer tempo.

# Vetores e Matrizes

Vetor – unidimensional

Matriz – bidimensional

Volume - tridimensional

# Vetores e Matrizes

- Implementação de vetores
- *Em C: int a [100];*
  - *Reserva n posições sucessivas de memória*
  - *Cada posição contém um único elemento*
  - *O endereço da primeira posição é a base a*
  - *Intervalo de 0 a (n-1)*
  - *Forma geral:*  
$$a[pos] \rightarrow \text{conteúdo } \{ \text{base}(a) + pos * \text{size} - t \}$$

*Todos os elementos possuem o mesmo tamanho, facilitando as operações básicas e a geração de código compacto e veloz.*

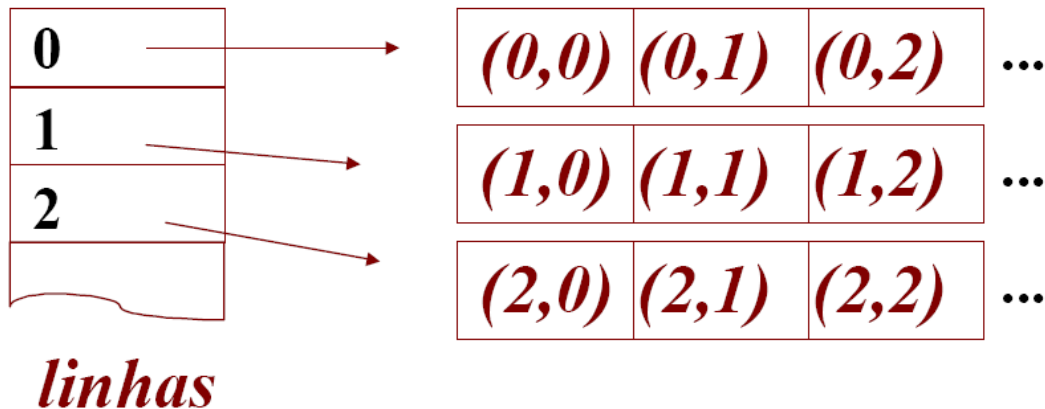
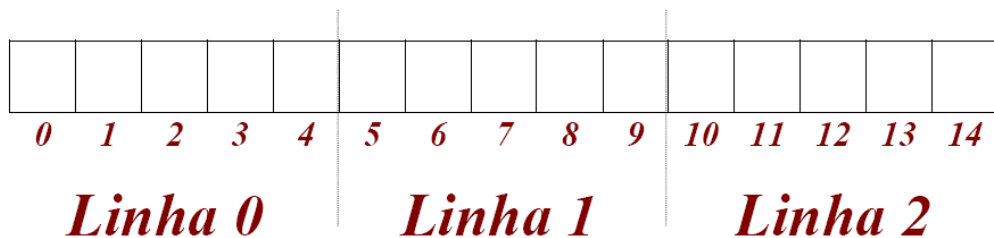
# Vetores e Matrizes

- *Matrizes*
  - Nome usual para definir vetores n-dimensionais
- Implementação de matrizes
  - `int a[10][10], int *a`

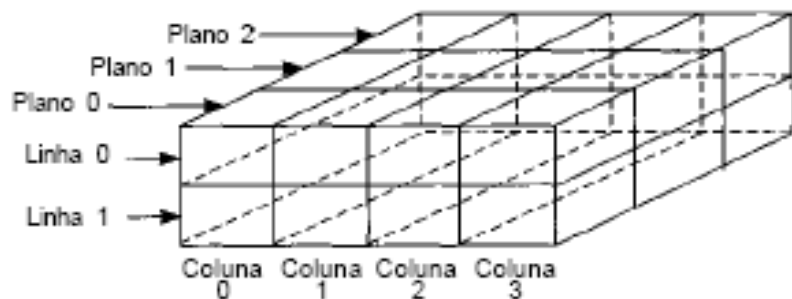
# Armazenamento de Matrizes em vetor

- Útil em aplicações de processamento de imagens e visualização volumétrica
- Matriz 2D
  - $K = @base(a) + i * numcol + j$
- Matriz 3D
  - $L = @base(a) + k * numcol * numlinha + i * numcol + j$

# Matrizes



# Matrizes



(a)

Cabeçalho		0	2
		0	1
		0	3
Plano 0	Linha 0	$b[0][0][0]$	← base (b)
		$b[0][0][1]$	
		$b[0][0][2]$	
	Linha 1	$b[0][1][0]$	
		$b[0][1][1]$	
		$b[0][1][2]$	
Plano 1	Linha 0	$b[1][0][0]$	
		$b[1][0][1]$	
		$b[1][0][2]$	
	Linha 1	$b[1][1][0]$	
		$b[1][1][1]$	
		$b[1][1][2]$	
Plano 2	Linha 0	$b[2][0][0]$	
		$b[2][0][1]$	
		$b[2][0][2]$	
	Linha 1	$b[2][1][0]$	
		$b[2][1][1]$	
		$b[2][1][2]$	

(b)



# Armazenamento de Matrizes em vetor

- Como percorrer uma matriz armazenada em um vetor?

```
mv = (int*)malloc(sizeof(int)*nl*nc);
```

```
for (i=0; i < nl; i++){  
    for (j=0; j < nc; j++){  
        mv[i*nc+j] = ...;  
    }  
}
```

# Tipos Abstratos de Dados

- Uma estrutura de dados e uma coleção de funções que operam sobre essa estrutura
- Uma nova forma de definir um tipo novo de dados juntamente com as operações que o manuseiam
- Exemplo: Programas sempre manuseiam coleções de itens (Analogia: Cofo de caranguejo)
  - Operações:
    - Criar → cria uma nova coleção
    - Inserir → adiciona um novo item à coleção
    - Remover → retira um item da coleção
    - Buscar → encontra um item na coleção atendendo algum critério
    - Destruir → Destroi a coleção

# O TAD *Coleção*

- Uma coleção pode ser implementada em uma linguagem de programação com:

- uma declaração de tipo

```
typedef struct _colecacao_ * Colecao;
```

- um conjunto de funções representando as operações

```
Colecao colCriar( int maxItems, int itemSize );
```

```
int colInserir( Colecao c, int item );
```

```
int colRetirar( Colecao c, int item );
```

```
int *colBuscar( Colecao c, int chave );
```

Essas declarações ficam em um arquivo de cabeçalho (header) *colecacao.h*, que deve ser incluído em todos os arquivos que utilizarem o TAD

Até agora não sabemos como esta implementada a coleção, apenas sabemos<sub>1</sub> sua especificação.

# Colecao.h

```
/*-----  
Colecao.h  
Arquivo com a especificação para o TAD  
Colecao, tipo de dado para  
uma coleção de inteiros  
Exemplo do curso: Estrutura de Dados  
-----  
  
-----*/  
#ifndef __COLECAO_H  
#define __COLECAO_H  
/*-----  
Definicoes locais  
-----*/  
typedef struct _colecao_ *Colecao;  
/*-----  
Funcoes que implementam as operacoes do  
TAD ColecaoInt  
-----*/
```

```
/* Cria um novo TAD Colecao  
Pre-condicao: max_items > 0  
Pos-condicao: retorna um ponteiro para uma  
novo TAD Colecao vazio*/  
Colecao colCriar( int max_items );  
/* Adiciona um item na Colecao  
Pre-condition: (c é um TAD Colecao criado  
por uma chamada a colCriar) e (o TAD  
Colecao nao esta cheio) e (item != NULL)  
Pos-condicao: item foi adicionado ao TAD c */  
int colInserir( Colecao c, int item );  
/* Retira um item da colecao  
Pre-condicao: (c é um TAD Colecao criado  
por uma chamada a colCriar) e &&  
(existe pelo menos um item no TAD  
Colecao) e (item != NULL)  
Pos-condicao: item foi eliminado do TAD c*/  
int colRetirar( Colecao c, int item );
```

# Colecao.h

```
/*  
    Encontra um item em um TAD  
    Colecao  
    Pre-condicao: (c é um TAD Colecao  
    criado por uma chamada a  
        colCriar) e  
        (key != NULL)  
    Pos-condicao: retorna um item  
    identificado por key se ele existir  
    no TAD c,  
        ou return NULL caso  
    contrário  
*/
```

```
int colBuscar( Colecao c, int key );
```

```
/*  
    Destroi um TAD Colecao  
    Pre-condicao: (c é um TAD Colecao  
    criado por uma chamada a  
        colCriar)  
    Pos-condicao: a memoria usada pelo  
    TAD foi liberada  
*/  
  
int colDestruir( Colecao c );  
  
#endif __COLECAO_H
```

# Questão

- Suponha que alguém forneça a você um TAD Coleção implementado, mas vc pode ler apenas o arquivo de cabeçalho, e ele permite que vc ligue com o seu programa um arquivo com a implementação desse TAD. Vc pode escrever um programa que utiliza este TAD como um tipo de dado?

Sim

- Vc conhece o nome do novo tipo de dado, o suficiente para declarar variáveis ponteiros para Colecao
- Vc também conhece os cabeçalhos e as especificações para cada operação

# Exemplo de Implementação - Vetor

Maneira mais simples de implementar uma **colecao** é usar um vetor para armazenar os itens da coleção.

```
/* Implementação do TAD Colecao como um vetor */  
#include "colecão.h"    /* inclui a especificação do TAD */  
typedef struct _colecão_ {  
    int numItens;  
    int maxItens;  
    int *itens;  
} Colecao;
```

Notas:

- a importação da especificação é para que o compilador verifique se estão compatíveis
- *item* pode ser definido como *int item[]* ou *int \*item*

A implementação dos métodos é encontrada em **Colecao.c**

# Exemplo de Implementação - Vetor

```
Colecao colCriar(int maxItens){  
    Colecao c;  
    if ( maxItens < 0 ) {    return NULL;    }  
    c = (Colecao)calloc( 1, sizeof(Colecao) );  
    if( c == NULL ) { return NULL; }  
    c→itens = (int *)calloc(maxItens,sizeof(int));  
    if ( c→itens == NULL ) {  
        free ( c );  
        return NULL;  
    }  
    c→maxItens = maxItens;  
    c→numItens = 0;  
    return c;  
}/* fim de colCriar */
```



# Exemplo de Implementação - Vetor

```
int colDestruir( Colecao c ) {  
    if ( c == NULL || c→itens == NULL ) {  
        return FALSE;  
    }  
    free(c→itens)  
    free(c);  
    return TRUE;  
} /* fim de colDestruir */
```

# Exemplo de Implementação - Vetor

```
int collInserir( Colecao c, int item ){  
    if(c == NULL || (c->numItens <  
        c->maxItens )) {  
        return FALSE;  
    }  
    c->items[c->numItens] = item;  
    c->numItens++;  
} /* fim de collInserir */
```

# Exemplo de Implementação - Vetor

```
int colRetirar( Colecao c, int item ) {  
    int i;  
    if( ( c == NULL ) || ( c→numItens < 1 ) ) return FALSE;  
    for(i= 0; i < c→numItens; i++) {  
        if ( item == c→itens[i] ) {  
            while( i < c→numItens ) {  
                c→itens[i] = c→itens[i+1];  
                i++;  
            }  
            c→numItens--;  
            return TRUE;  
        } /* if */  
    } /* for */  
} /* fim de colRetirar */
```

# Exemplo de Implementação - Vetor

```
int colBuscar( Collection c, int key ){
    int i;
    if (( c == NULL ) ){          return -1;      }
    for(i=0;i<c→numItens;i++) {
        if (c→itens[i] == key){
            return c→itens[i];
        }
    }
} /* fim de colBuscar */
```

# TAD Genérico

- Problema:
  - Necessário implementar um TAD Colecao para cada tipo de dados
- Solução:
  - reimplementar o TAD sem especificar que tipo de dados queremos colocar nele.
  - Usar um TAD de ponteiros para void.

# TAD Genérico

## *Especificacao das operacoes*

*Colecao colCriar( int maxItems );*

*int colInserir( Colecao c, void \* item );*

*int colRetirar( Colecao c, void \*item );*

*int \*colBuscar( Colecao c, void \*chave );*

## *Especificação do vetor*

*typedef struct \_colecao\_ {*

*int numItens;*

*int maxItens;*

*void \*item;*

*}Colecao;*

## *Reimplementar as funções*

Problema com consulta e remocao, necessario passar uma funcao como parametro.

# Funções como Parametros - Exemplo

- Quicksort

```
void qsort( void *base, size_t n, size_t size,  
           int (*compar)( void *, void * ) );
```

<b>base</b>	<b>endereço de um vetor</b>
<b>n</b>	<b>número de elementos</b>
<b>size</b>	<b>tamanho do elemento</b>
<b>compar</b>	<b>função de comparação</b>

- C permite passar uma função como parâmetro para uma outra função

# Funções como tipos de dados - declaração

- Funções como tipos de dados
  - Declaração

```
int (*compar)( void *, void * );
```

**Parênteses em torno do \* e do nome da função**  
**Define que é um ponteiro para função**  
diferente de:

```
int *compar( void *, void * );
```

Função retornando um ponteiro para inteiro



# Funções como tipos de dados - declaração

- Declaração

```
int (*compar)( void *, void * );
```



Função retorna um *int*



Possui dois argumentos *void \**

- Declara um **ponteiro** para uma função de nome *compar* que retorna *int* e recebe dois argumentos *void \**

# Funções como tipos de dados - declaração

- Declaração

```
int (*compar)( void *, void * );
```



Função retorna um *int*



Possui dois argumentos *void \**

- A função de comparação passada para *quicksort* retorna

-1      *\*arg1* < *\*arg2*

0        *\*arg1* == *\*arg2*

+1       *\*arg1* > *\*arg2*

Interpretado como definindo  
ordem não magnitude

# Funções como tipos de dados

- Uso
  - Funções de bibliotecas que precisam de uma função com objetivo especial
    - Ordenação precisa do conceito de ordem
    - função `compar` fornece isto
    - Conceito de ordem pode ser complexo
      - *eg* nomes em uma lista telefônica
    - Busca também necessita de ordem

# TAD Genérico

- Generalizando nosso TAD colecao
  - Usamos ponteiros para a estrutura (struct) que representa o elemento da coleção
    - Permite que o TAD colecao armazene qualquer tipo de objetos
  - *Mas* precisamos assumir que existem duas funções externas
    - `itemkey` e `itemcmp`
    - Restringindo a uma coleção por programa
- Coloque a função de ordenação como atributo
  - Regra de ordenação é armazenada com os atributos da coleção
  - Quantas coleções eu quiser.

# Coleção Genérica


- Redefina a função de criação

```
Collection Criar( int max_items,  
                int (*compar)(const void *,const void * ) );
```

- Redefina a estrutura da coleção

```
struct t_collection {  
    int max_items;  
    int cnt;  
    int (*compar)( const void *, const void * );  
    void *data;  
}Colecao;
```

*Compar é um ponteiro  
para uma função -  
em geral o endereço do  
início do código da função*



# Coleção Genérica

- Na função de construção trate a função parametro como se fosse um ponteiro qualquer

```
Colecao Criar( int max_items,  
              int (*compar)(const void *,const void * ) ) {  
    Collection c;  
    c = malloc( sizeof( struct collection ) );  
    if ( c != NULL ) {  
        c->max_items = max_items;  
        c->compar = compar;  
        .....  
    }  
    return c;  
}
```

# Usando um ponteiro para uma função

- Use o atributo passado como o nome de uma função

```
void *Busca( void *key )  
{  
    int k;  
    for(k=0;k<c->cnt;k++) {  
        if( *c->compar( key, c->data[k] ) == 0 )  
            return c->data[k];  
    }  
    return NULL;  
}
```

*c->compar é um ponteiro para uma função -  
acesse o endereço nele para  
executar a função*

*... e forneça os  
argumentos , como se  
fosse o nome de uma  
função que vc já esta  
acostumado a usar!*

# TAD - Resumo

- Um TAD consiste de um novo tipo de dados juntamente com operações que manipulam esses dados
- O TAD é colocado em um arquivo `.c` separado de sua especificação que fica em um arquivo `.h`
- Qualquer programa pode usar o TAD
- Se a implementação for modificada os programas que utilizam o TAD não precisam ser alterados
- TADs genéricos são uma importante característica



# Bibliografia

Adaptado do slide da disciplina ED1 – Prof Anselmo Paiva –  
Universidade Federal do Maranhão

TENENBAUM, Aaron M.; LANGSAM, Yedidyah; AUGENSTEIN, Moshe J. Estruturas de dados usando C. Pearson Makron Books, 2004.

KRUSE, Robert et al. Data structures and program design in C. Pearson Education India, 2007.