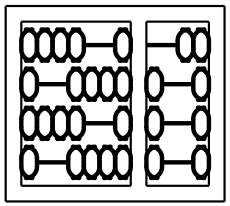


Universidade Estadual de Campinas - UNICAMP
Instituto de Computação - IC

# Introdução a Estruturas de Dados

August 3, 2011

# Conteúdo

# Conteúdo

- Alguns aspectos da linguagem C

# Conteúdo

- Alguns aspectos da linguagem C
- **Tipos Abstratos de Dados**

```c
/* Calculo da media de n numeros reais */

#include <stdio.h>

int main(void)
{
    int i ;
    int n ;
    float med = 0.0 ; /* valor da media */

    /* leitura do numero de valores */
    scanf("%d", &n) ;

    /* leitura de valores e calculo do somatorio */
    for (i = 0; i < n; i++) {
        float v ;
        scanf("%f", &v) ;    /* le cada valor */
        med = med + v ;      /* acumula soma dos valores */
    }
```

```
 /* calculo da media */
    med = med / n ;

    /* exibicao do resultado */
    printf("Valor da media = %f\n", med) ;

    return 0 ;

}
```

```
 /* calculo da media */
    med = med / n ;

    /* exibicao do resultado */
    printf("Valor da media = %f\n", med) ;

    return 0 ;

}
```

- Neste caso, o cálculo de med $= \frac{\sum v}{n}$ não exige o armazenamento do conjunto de valores $v$.

O cálculo da variância, *var*, exige que tal conjunto seja armazenado:

$$var = \frac{\sum (v - med)^2}{n}$$

O cálculo da variância, *var*, exige que tal conjunto seja armazenado:

$$var = \frac{\sum (v - med)^2}{n}$$

Neste caso, podemos utilizar um **vetor** de n elementos.

$\Rightarrow$ Armazenamento estático:

Declaracão: **int** v[10]

• O acesso a cada elemento do vetor é feito por meio de uma indexação da variável $v$: $v[0]$, $v[1]$, ... $v[9]$.

• O acesso a cada elemento do vetor é feito por meio de uma indexação da variável $v$: $v[0]$, $v[1]$, ... $v[9]$.

Assim:
v[0] $\rightarrow$ acessa o primeiro elemento de $v$

• O acesso a cada elemento do vetor é feito por meio de uma indexação da variável $v$: $v[0]$, $v[1]$, ... $v[9]$.

Assim:
v[0] $\rightarrow$ acessa o primeiro elemento de $v$
v[1] $\rightarrow$ acessa o segundo elemento de $v$

• O acesso a cada elemento do vetor é feito por meio de uma indexação da variável $v$: $v[0]$, $v[1]$, ... $v[9]$.

Assim:
v[0] $\rightarrow$ acessa o primeiro elemento de $v$
v[1] $\rightarrow$ acessa o segundo elemento de $v$
. . .

• O acesso a cada elemento do vetor é feito por meio de uma indexação da variável $v$: $v[0]$, $v[1]$, ... $v[9]$.

Assim:
v[0] $\rightarrow$ acessa o primeiro elemento de $v$
v[1] $\rightarrow$ acessa o segundo elemento de $v$
. . .
v[9] $\rightarrow$ acessa o décimo elemento de $v$

• O acesso a cada elemento do vetor é feito por meio de uma indexação da variável $v$: $v[0]$, $v[1]$, ... $v[9]$.

Assim:
v[0] $\rightarrow$ acessa o primeiro elemento de $v$
v[1] $\rightarrow$ acessa o segundo elemento de $v$
. . .
v[9] $\rightarrow$ acessa o décimo elemento de $v$
v[10] $\rightarrow$ invasão de memória (ERRO)

```c
/* Calculo da media e da variancia de 10 numeros reais */

#include <stdio.h>

int main (void) {

    float v[10] ;  /* vetor com 10 elementos */
    float med, var ;
    int i ;

    /* leitura dos valores */
    for (i = 0; i < 10; i++)
        scanf("%f", &v[i]) ; /* le elementos do vetor */

    /* calculo da media */
    med = 0.0 ;      /* inicializa media */
    for (i = 0; i < 10; i++)
        med = med + v[i] ;    /* acumula soma dos elementos *
    med = med / 10 ;          /* calcula a media */
```

```
/* calculo da variancia */
    var = 0.0 ;
    for (i = 0; i < 10; i++)
        var = var+(v[i]-med)*(v[i]-med) ;    /* acumula */
    var = var / 10 ;

    /* exibicao do resultado */
    printf("Media = %f   ; Variancia = %f \n", med, var) ;

    return 0 ;

} /* fim do main */
```

- Na declaração:

$$float \quad v[i];$$

o símbolo $v$ é uma constante que representa o endereço inicial do conjunto de dados do vetor.

*Sem indexação, $v$* **aponta** *para o primeiro elemento do vetor.*

- Na declaração:

$$float \quad v[i];$$

o símbolo $v$ é uma constante que representa o endereço inicial do conjunto de dados do vetor.

*Sem indexação, v* **aponta** *para o primeiro elemento do vetor.*

Logo:
v + 0 $\rightarrow$ aponta para o primeiro elemento do vetor

- Na declaração:

$$float \quad v[i];$$

o símbolo *v* é uma constante que representa o endereço inicial do conjunto de dados do vetor.

*Sem indexação, v* **aponta** *para o primeiro elemento do vetor.*

Logo:
v + 0 → aponta para o primeiro elemento do vetor
v + 1 → aponta para o segundo elemento do vetor

- Na declaração:

$$float \quad v[i];$$

o símbolo *v* é uma constante que representa o endereço inicial do conjunto de dados do vetor.

*Sem indexação, v* **aponta** *para o primeiro elemento do vetor.*

Logo:
v + 0 → aponta para o primeiro elemento do vetor
v + 1 → aponta para o segundo elemento do vetor
. . .

- Na declaração:

$$float \quad v[i];$$

o símbolo *v* é uma constante que representa o endereço inicial do conjunto de dados do vetor.

*Sem indexação, v* **aponta** *para o primeiro elemento do vetor.*

Logo:
v + 0 → aponta para o primeiro elemento do vetor
v + 1 → aponta para o segundo elemento do vetor
. . .
v + 9 → aponta para o décimo elemento do vetor

• Na declaração:

$$float \quad v[i];$$

o símbolo $v$ é uma constante que representa o endereço inicial do conjunto de dados do vetor.

*Sem indexação, v* **aponta** *para o primeiro elemento do vetor.*

Logo:
v + 0 → aponta para o primeiro elemento do vetor
v + 1 → aponta para o segundo elemento do vetor
. . .
v + 9 → aponta para o décimo elemento do vetor

• Assim, escrever $\&v[i]$ é equivalente a escrever $(v + i)$. Da mesma forma, escrever $v[i]$ é equivalente a escrever $*(v + i)$.

- Passagem de vetores para funções

$\Rightarrow$ Passar o endereço da primeira posição do vetor

```
/* Media e variancia (segunda versao) */

#include <stdio.h>

/* Funcao que calcula a media */
float media (int n, float *v)
{
      int i ;
      float s = 0.0 ;
      for (i = 0; i < n; i++)
          s += v[i] ;

      return s/n ;
}
```

- Passagem de vetores para funções

⇒ Passar o endereço da primeira posição do vetor

```
/* Media e variancia (segunda versao) */

#include <stdio.h>

/* Funcao que calcula a media */
float media (int n, float *v)
{
      int i ;
      float s = 0.0 ;
      for (i = 0; i < n; i++)
          s += v[i] ;

      return s/n ;
}
```

- Passagem de vetores para funções

$\Rightarrow$ Passar o endereço da primeira posição do vetor

```
/* Media e variancia (segunda versao) */

#include <stdio.h>

/* Funcao que calcula a media */
float media (int n, float *v)
{
      int i ;
      float s = 0.0 ;
      for (i = 0; i < n; i++)
          s += v[i] ;

      return s/n ;
}
```

```
/* Funcao que calcula a variancia */
float variancia(int n, float *v, float m)
{
      int i ;
      float s = 0.0 ;
      for (i = 0; i < n; i++)
          s += (v[i] - m) * (v[i] - m) ;
      return s/n ;
}
```

```
/* Programa principal */

int main(void)  {
    float v[10] ;
    float med, var ;
    int i ;
    /* leitura dos valores */
    for (i = 0; i < 10; i++)
        scanf("%f", &v[i]) ;

    med = media(10, v) ;
    var = variancia(10, v, med) ;
    printf("Media = %f   Variancia = %f  \n", med, var) ;
    return 0 ;
}
```

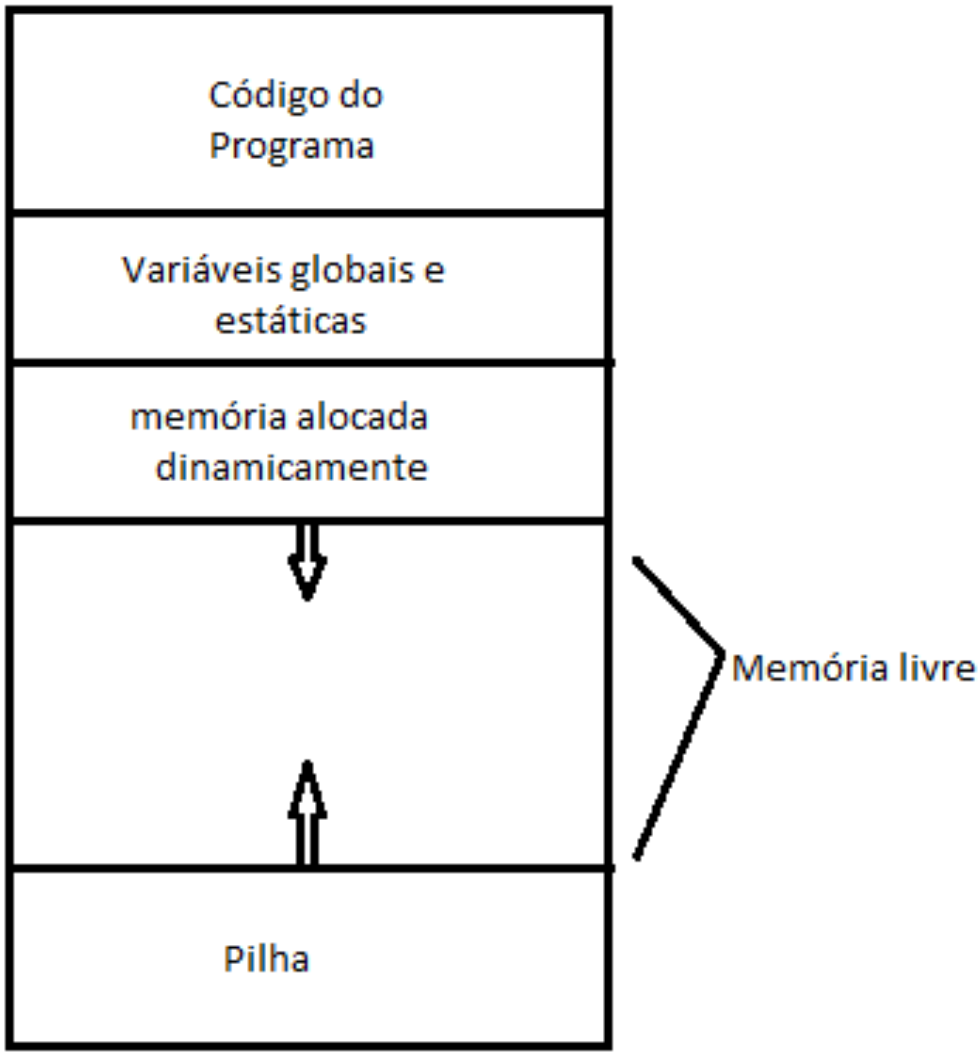
- Outro exemplo:

```
/* Incrementa os elementos de um vetor */

#include <stdio.h>

void incr_vetor(int n, int *v){
int i ;
for (i = 0; i < n; i++)
 v[i]++ ;
}
int main (void) {
int a[ ] = {1, 3, 5} ;
incr_vetor(3, a) ;
printf("%d %d %d \n", a[0], a[1], a[2]) ;
return 0 ;
}
```

- Alocação dinâmica



• Alocação do vetor v de 10 inteiros:

```
int  *v ;
v = (int *) malloc(10*sizeof(int)) ;
```

– Declaração *int *v ;*


– Comando *v = (int *) malloc(10*sizeof(int)) ;*

• A função *malloc* retorna um endereço nulo (NULL) caso não haja espaço para a alocação dinâmica.

• Exemplo de controle de execução:

```
...
v = (int *) malloc(10*sizeof(int)) ;
if (v == NULL) {
  printf(''Memoria insuficiente\n'') ;
  exit(1) /* aborta o programa */
}
...
```

• A função *malloc* retorna um endereço nulo (NULL) caso não haja espaço para a alocação dinâmica.

• Exemplo de controle de execução:

```
...
v = (int *) malloc(10*sizeof(int)) ;
if (v == NULL) {
  printf(''Memoria insuficiente\n'') ;
  exit(1) /* aborta o programa */
}
...
```

• Liberação do espaço **alocado dinamicamente**:
*free*($v$)

# Matrizes

- Vetores bidimensionais - Matrizes

- Vetores bidimensionais - Matrizes

– Alocação estática:

```
float mat[4][3] ;
float mat[4][3] = {{1,2,3},{4,5,6},{7,8,9},{10,11,12}} ;
float mat[ ][3] = {1,2,3,4,5,6,7,8,9,10,11,12} ;
```

• Vetores bidimensionais - Matrizes

– Alocação estática:

float mat[4][3] ;
float mat[4][3] = {{1,2,3},{4,5,6},{7,8,9},{10,11,12}} ;
float mat[ ][3] = {1,2,3,4,5,6,7,8,9,10,11,12} ;

• O acesso se dá por indexação dupla, apesar do **armazenamento contíguo na memória**.
a = mat[i][j] ;

• Vetores bidimensionais - Matrizes

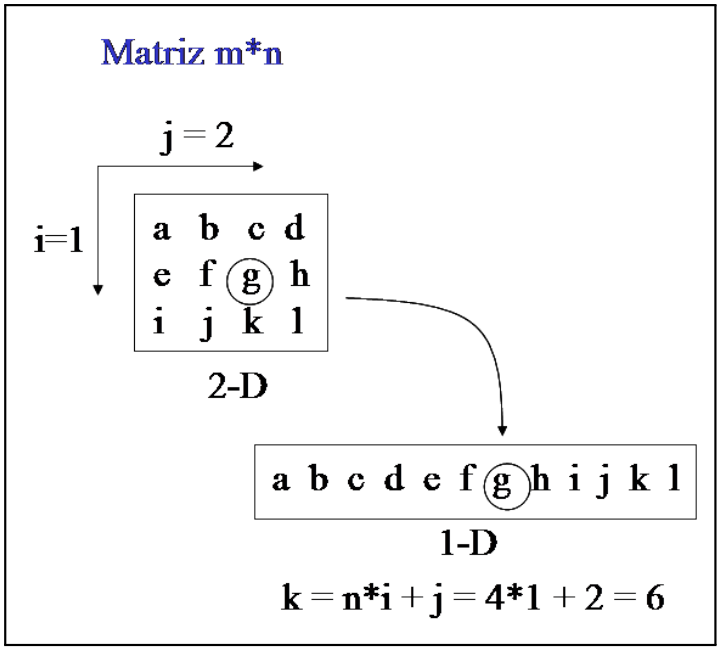
– Alocação estática:

float mat[4][3] ;
float mat[4][3] = {{1,2,3},{4,5,6},{7,8,9},{10,11,12}} ;
float mat[ ][3] = {1,2,3,4,5,6,7,8,9,10,11,12} ;

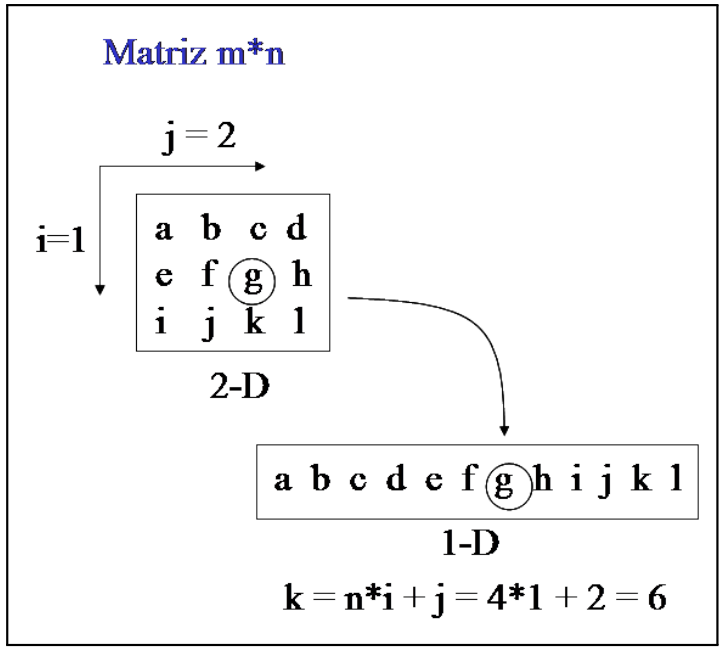
• O acesso se dá por indexação dupla, apesar do **armazenamento contíguo na memória**.
a = mat[i][j] ;

• Protótipo de uma função que recebe a matriz
void func(..., float mat[ ][3],...) ;

- Matrizes dinâmicas: alocadas em tempo de execução.
- *C* só permite alocar conjuntos unidimensionais.



- Matrizes dinâmicas: alocadas em tempo de execução.
- *C* só permite alocar conjuntos unidimensionais.




$\Rightarrow$ A alocação da "matriz" representa uma alocação de um vetor com $m * n$ elementos.

```
float *mat ; /* matriz representada por um vetor */
...
mat = (float *) malloc(m*n*sizeof(float)) ;
...
```

- Matrizes representadas por um vetor de ponteiros:

- Matrizes representadas por um vetor de ponteiros:

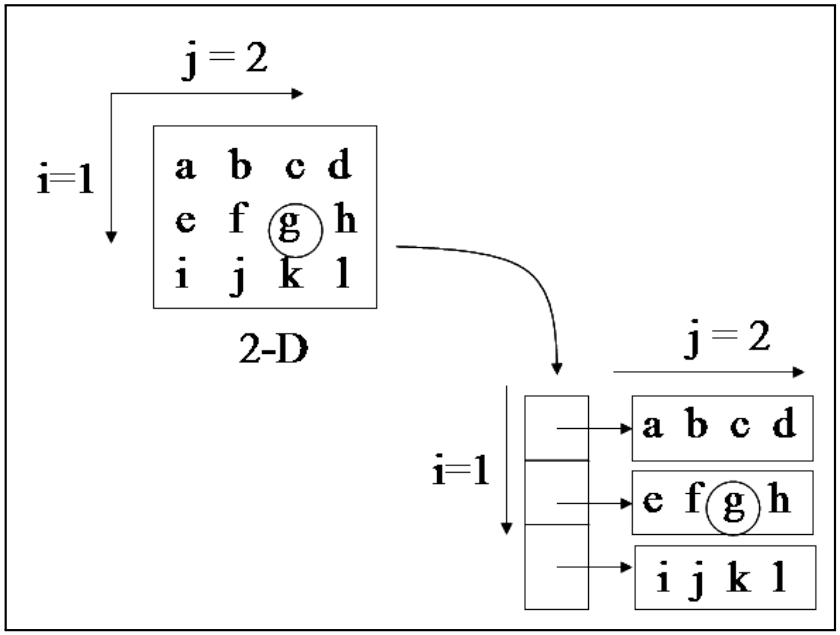
$\rightarrow$ Cada linha é representada por um vetor independente.

$\rightarrow$ A matriz é representada por um vetor de ponteiros, no qual cada componente armazena o endereço do primeiro elemento de cada linha.

• Matrizes representadas por um vetor de ponteiros:

→ Cada linha é representada por um vetor independente.

→ A matriz é representada por um vetor de ponteiros, no qual cada componente armazena o endereço do primeiro elemento de cada linha.



- A alocação da matriz:

```
int i ;
float **mat  /* matriz indicada por um vetor  de ponteiros */
...
mat = (float **)malloc(m*sizeof(float *)) ;
for (i=0; i <m; i++)
  mat[i] = (float *)malloc(n*sizeof(float)) ;
...
```

- Liberação do espaço alocado

```
...
for (i=0; i<m; i++)
   free(mat[i]) ;
free(mat) ;
...
```

- **Exemplo:** A matriz transposta com vetor simples

```
float *transposta(int m, int n, float *mat) {
      int i, j;
      float *trp ;

      /* aloca matriz transposta */
      trp = (float *)malloc(n*m*sizeof(float)) ;

      /* calcula a transposta */
      for(i=0; i<m; i++)
        for(j=0; j<n; j++)
          trp[j*m+i] = mat[i*n+j] ;

      return trp ;
}
```

- **Exemplo:** A matriz transposta com vetor de ponteiros

```
float **transposta(int m, int n, float **mat) {
      int i, j;
      float **trp ;

      /* aloca matriz transposta: m linhas, n colunas */
      trp = (float **)malloc(m*sizeof(float *)) ;
        for(i=0; i<m; i++)
          trp[i] = (float *)malloc(n*sizeof(float)) ;

      /* calcula transposta */
      for(i=0 ; i<m; i++)
        for(j=0; j<n; j++)
          trp[j][i] = mat[i][j] ;

      return trp ;
}
```

# Tipos Estruturados

• Tipo Estrutura: tipo de dados cujos campos são compostos de vários valores de tipos mais simples.

```
struct ponto {
  float x ;
  float y ;
} ;

struct ponto p ;
...
p.x = 5.0 ;
p.y = 7.0 ;
```

# Tipos Estruturados

• Tipo Estrutura: tipo de dados cujos campos são compostos de vários valores de tipos mais simples.

```
struct ponto {
  float x ;
  float y ;
} ;

struct ponto p ;
...
p.x = 5.0 ;
p.y = 7.0 ;
```

• Os elementos da estrutura são manipulados da mesma forma que as variáveis simples.

```c
/* le e imprime as coordenadas de um ponto */

#include <stdio.h>

struct ponto {
  float x ;
  float y ;
} ;

int main (void) {
  struct ponto p ;

  printf("Digite as coordenadas do ponto (x,y): ") ;
  scanf("%f %f", &p.x, &p.y) ;
  printf("O ponto fornecido foi: (%.2f,%.2f)\n", p.x, p.y) ;
  return 0 ;
}
```

- Variável do tipo ponteiro para estrutura: *struct ponto *pp ;*

- Variável do tipo ponteiro para estrutura: *struct ponto *pp ;*

Acesso aos campos:

*(*pp).x = 12.0* ou *pp->x = 12.0*

- Variável do tipo ponteiro para estrutura: *struct ponto *pp ;*

Acesso aos campos:

*(*pp).x = 12.0* ou *pp->x = 12.0*

Acesso ao endereço de um campo:

*&pp->x*

- Passagem de estruturas para funções

- Passagem de estruturas para funções

→ **Por valor** (uma cópia da estrutura):

```
void imprime (struct ponto p) {
printf("O ponto tem coordenadas: (%.2f, %.2f)\n", p.x, p.y) ;
}
```

- Passagem de estruturas para funções

→ **Por valor** (uma cópia da estrutura):

```
void imprime (struct ponto p) {
printf("O ponto tem coordenadas: (%.2f, %.2f)\n", p.x, p.y) ;
}
```

→ **Por referência** (o endereço da estrutura; menos custosa):

```
void imprime (struct ponto *pp) {
printf("O ponto tem coordenadas:(%.2f, %.2f)\n", pp->x, p->y) ;
}
```

- **Alocação dinâmica de estruturas**

```
struct ponto *p ;
p = (struct ponto *)malloc(sizeof(struct ponto)) ;
```

- **Alocação dinâmica de estruturas**

```
struct ponto *p ;
p = (struct ponto *)malloc(sizeof(struct ponto)) ;
```

Acesso aos campos:

...
*p->x = 12.0*
...

- Definição de tipos

```
struct ponto {
  float x ;
  float y ;
} ;

typedef struct ponto Ponto, *PPonto ;
```

- Definição de tipos

```
struct ponto {
  float x ;
  float y ;
} ;

typedef struct ponto Ponto, *PPonto ;
```

Declaração de variáveis:

```
Ponto p ;        /* Variavel p armazena um ponto */
PPonto pp ;      /* Um ponteiro para a estrutura ponto */
```

- Vetores de estruturas

```
Ponto centro_geom(int n, Ponto *v) {

int i ;
Ponto p = {0.0, 0.0}   /* declara e inicializa ponto */
for (i=0; i<n; i++) {
  p.x += v[i].x ;
  p.y += v[i].y ;
}

p.x /= n ;
p.y /= n ;
return p ;
}
```

```
#define MAX 10000

struct aluno{
 int mat ;
 char nome[81] ;
 char end[121] ;
 char tel[21] ;
 ...
} ;
typedef struct aluno Aluno ;
...
Aluno tab[MAX] ;
...
tab[i].mat = 1023456 ;
...
}
```

• Vetores de ponteiros para estruturas

$\Rightarrow$ Útil no tratamento de grandes conjuntos de estruturas complexas

• No caso anterior, gasta-se muito espaço de memória no armazenamento das MAX estruturas

• Vetores de ponteiros para estruturas

$\Rightarrow$ Útil no tratamento de grandes conjuntos de estruturas complexas

• No caso anterior, gasta-se muito espaço de memória no armazenamento das MAX estruturas

Alternativa:

Aluno *tab[MAX] ;

• Vetores de ponteiros para estruturas

$\Rightarrow$ Útil no tratamento de grandes conjuntos de estruturas complexas

• No caso anterior, gasta-se muito espaço de memória no armazenamento das MAX estruturas

Alternativa:

Aluno *tab[MAX] ;

• Neste caso, cada elemento do vetor ocupa apenas o espaço necessário para armazenar um ponteiro.

- Exemplos de operações:

```
void inicializa(int n, Aluno **tab) {
  int i ;
  for(i=0; i<n; i++)
    tab[i] = NULL ;
}
...

void preenche(int n, Aluno **tab, int i) {
  if (tab[i] == NULL)
    tab[i] = (Aluno *)malloc(sizeof(Aluno)) ;

  scanf(''%d'', &tab[i]->mat) ;
  scanf(''%80[^\n]'', tab[i]->nome) ;
  scanf(''%120[^\n]'', tab[i]->end) ;
  scanf(''%20[^\n]'', tab[i]->tel) ;
}
```

- Exemplo de programa principal:

```
#include <stdio.h>

int main (void) {
  Aluno  *tab[10] ;

  inicializa(10, tab) ;
  preenche(10, tab, 0) ;  /* insere aluno 0 */
  preenche(10, tab, 1) ;  /* insere aluno 1 */
  ...
  return 0 ;
}
```

# Tipos Abstratos de Dados

• TAD: Encapsula/"esconde" a forma como um determinado tipo é implementado.

# Tipos Abstratos de Dados

• TAD: Encapsula/"esconde" a forma como um determinado tipo é implementado.

• Desacopla a implementação do uso
$\Rightarrow$ *Maior nível de abstração*

# Tipos Abstratos de Dados

• TAD: Encapsula/"esconde" a forma como um determinado tipo é implementado.

• Desacopla a implementação do uso
⇒ *Maior nível de abstração*

A linguagem C e a implementação de um TAD:

⇒ **Módulos** *.c* **e interfaces** *.h* **em separado**.

• Um **módulo** *.c* contém o conjunto das funções associadas à implementação do tipo.

• Um **módulo** *.c* contém o conjunto das funções associadas à implementação do tipo.

• A **interface** *.h* contém os protótipos das funções exportadas e oferecidas pelo módulo.

• Um **módulo** *.c* contém o conjunto das funções associadas à implementação do tipo.

• A **interface** *.h* contém os protótipos das funções exportadas e oferecidas pelo módulo.

**A subdivisão de um programa em módulos e a criação de TAD constitui uma técnica de programação muito importante !**

- **Exemplo 1:** TAD Ponto

– Definir o tipo de dados *Ponto* para representar um ponto em $\Re^2$.

- **Exemplo 1:** TAD Ponto

– Definir o tipo de dados *Ponto* para representar um ponto em $\Re^2$.

Funções operando sobre o tipo *Ponto*:

- **Exemplo 1:** TAD Ponto

– Definir o tipo de dados *Ponto* para representar um ponto em $\Re^2$.

Funções operando sobre o tipo *Ponto*:

- *cria*: operação que cria um ponto com coordenadas $x$ e $y$.

- **Exemplo 1:** TAD Ponto

– Definir o tipo de dados *Ponto* para representar um ponto em $\Re^2$.

Funções operando sobre o tipo *Ponto*:

- *cria*: operação que cria um ponto com coordenadas $x$ e $y$.
- *libera*: operação que libera a memória alocada por um ponto.

- **Exemplo 1:** TAD Ponto

– Definir o tipo de dados *Ponto* para representar um ponto em $\Re^2$.

Funções operando sobre o tipo *Ponto*:

- *cria*: operação que cria um ponto com coordenadas $x$ e $y$.
- *libera*: operação que libera a memória alocada por um ponto.
- *acessa*: operação que retorna as coordenadas de um ponto.

- **Exemplo 1:** TAD Ponto

– Definir o tipo de dados *Ponto* para representar um ponto em $\Re^2$.

Funções operando sobre o tipo *Ponto*:

- *cria*: operação que cria um ponto com coordenadas $x$ e $y$.
- *libera*: operação que libera a memória alocada por um ponto.
- *acessa*: operação que retorna as coordenadas de um ponto.
- *atribui*: operação que atribui novos valores às coordenadas de um ponto.

- **Exemplo 1:** TAD Ponto

– Definir o tipo de dados *Ponto* para representar um ponto em $\Re^2$.

Funções operando sobre o tipo *Ponto*:

- *cria*: operação que cria um ponto com coordenadas $x$ e $y$.
- *libera*: operação que libera a memória alocada por um ponto.
- *acessa*: operação que retorna as coordenadas de um ponto.
- *atribui*: operação que atribui novos valores às coordenadas de um ponto.
- *distancia*: operação que calcula a distância entre dois pontos.
- ...

- A interface *ponto.h*:

```
/* TAD: Ponto (x,y) */
/* Tipo exportado */
typedef struct ponto Ponto ;
/* Funcoes exportadas */
/* funcao cria: bla-bla-bla */
Ponto *pto_cria(float x, float y) ;
/* funcao libera */
void pto_libera(Ponto *p) ;
/* funcao acessa */
void pto_acessa(Ponto *p, float *x, float *y) ;
/* funcao atribui */
void pto_atribui(Ponto *p, float x, float y) ;
/* funcao distancia */
float pto_distancia(Ponto *p1, Ponto *p2) ;
```

• **A partir da interface anterior, pode-se considerar diferentes programas que utilizem as funcionalidades exportadas.**

• Um programa principal, *prog*1.*c*, que usa o TAD deve incluir o arquivo interface .*h*. Exemplo:

```
#include <stdio.h>
#include "ponto.h"

int main (void) {
    Ponto *p = pto_cria(2.0,1.0) ;
    Ponto *q = pto_cria(3.4,2.1) ;
    float d = pto_distancia(p, q) ;
    printf("Distancia entre pontos: %f\n", d) ;
    pto_libera(p) ;
    pto_libera(q) ;
    return 0 ;
}
```

• O **módulo** *ponto.c* contém a implementação das funções para o referido TAD.

```
/* interfaces importadas */
#include <stdlib.h>   /* malloc, free, exit, etc */
#include <stdio.h>    /* printf, etc */
#include <math.h>     /* sqrt, etc */
#include "ponto.h"    /funcoes do TAD Ponto */

struct ponto{
       float x ;
       float y ;
} ;
```

- ... continuação de *ponto.c*

```
Ponto *pto_cria (float x, float y) {
      Ponto *p = (Ponto *)malloc(sizeof(Ponto)) ;
      if (p == NULL) {
            printf("Memoria insuficiente!\n") ;
            exit(1) ;
      }
      p->x = x ;
      p->y = y ;
      return p ;
}
```

- ... continuação de *ponto.c*

```
void pto_libera(Ponto *p) {
     free(p) ;
}

void pto_acessa(Ponto *p, float *x, float *y) {
     *x = p->x ;
     *y = p->y ;
}

void pto_atribui(Ponto *p, float x, float y) {
     p->x = x ;
     p->y = y ;
}
```

- ... continuação de *ponto.c*

```
float pto_distancia(Ponto *p1, Ponto *p2) {
      float dx = p2->x - p1->x ;
      float dy = p2->y - p1->y ;
      return sqrt(dx*dx + dy*dy) ;
}
```

- ... continuação de *ponto.c*

```
float pto_distancia(Ponto *p1, Ponto *p2) {
      float dx = p2->x - p1->x ;
      float dy = p2->y - p1->y ;
      return sqrt(dx*dx + dy*dy) ;
}
```

Compilação:
*gcc -o prog1 prog1.c ponto.c -lm*

- ... continuação de *ponto.c*

```
float pto_distancia(Ponto *p1, Ponto *p2) {
      float dx = p2->x - p1->x ;
      float dy = p2->y - p1->y ;
      return sqrt(dx*dx + dy*dy) ;
}
```

Compilação:
*gcc -o prog1 prog1.c ponto.c -lm*

Do ponto de vista do usuário, o trabalho aqui consiste em acessar as informações relativas a um ponto sem que se conheça concretamente a forma de manipulação dos dados envolvidos na operação, assim como a implementação das respectivas funções associadas ao TAD.

• O ato de esconder a implementação de um TAD flexibiliza o projeto das suas funções, sem que isto afete o trabalho do usuário.

• O ato de esconder a implementação de um TAD flexibiliza o projeto das suas funções, sem que isto afete o trabalho do usuário.

• **Exemplo:** TAD matriz
– Representa matrizes de valores reais, alocadas dinamicamente, e de dimensões $m$ por $n$ fornecidas em tempo de execução.

• O ato de esconder a implementação de um TAD flexibiliza o projeto das suas funções, sem que isto afete o trabalho do usuário.

• **Exemplo:** TAD matriz
– Representa matrizes de valores reais, alocadas dinamicamente, e de dimensões $m$ por $n$ fornecidas em tempo de execução.

Operações sobre o TAD Matriz:

• O ato de esconder a implementação de um TAD flexibiliza o projeto das suas funções, sem que isto afete o trabalho do usuário.

• **Exemplo:** TAD matriz
– Representa matrizes de valores reais, alocadas dinamicamente, e de dimensões $m$ por $n$ fornecidas em tempo de execução.

Operações sobre o TAD Matriz:

1. *cria*: cria uma matriz de dimensão $m$ por $n$.

• O ato de esconder a implementação de um TAD flexibiliza o projeto das suas funções, sem que isto afete o trabalho do usuário.

• **Exemplo:** TAD matriz
– Representa matrizes de valores reais, alocadas dinamicamente, e de dimensões $m$ por $n$ fornecidas em tempo de execução.

Operações sobre o TAD Matriz:

1. *cria*: cria uma matriz de dimensão $m$ por $n$.
2. *libera*: libera a memória alocada para a matriz.

• O ato de esconder a implementação de um TAD flexibiliza o projeto das suas funções, sem que isto afete o trabalho do usuário.

• **Exemplo:** TAD matriz
– Representa matrizes de valores reais, alocadas dinamicamente, e de dimensões $m$ por $n$ fornecidas em tempo de execução.

Operações sobre o TAD Matriz:

1. *cria*: cria uma matriz de dimensão $m$ por $n$.
2. *libera*: libera a memória alocada para a matriz.
3. *acessa*: acessa o elemento da linha $i$ e da coluna $j$ da matriz.

• O ato de esconder a implementação de um TAD flexibiliza o projeto das suas funções, sem que isto afete o trabalho do usuário.

• **Exemplo:** TAD matriz
– Representa matrizes de valores reais, alocadas dinamicamente, e de dimensões $m$ por $n$ fornecidas em tempo de execução.

Operações sobre o TAD Matriz:

1. *cria*: cria uma matriz de dimensão $m$ por $n$.
2. *libera*: libera a memória alocada para a matriz.
3. *acessa*: acessa o elemento da linha $i$ e da coluna $j$ da matriz.
4. *atribui*: atribui o elemento da linha $i$ e da coluna $j$ da matriz.

• O ato de esconder a implementação de um TAD flexibiliza o projeto das suas funções, sem que isto afete o trabalho do usuário.

• **Exemplo:** TAD matriz
– Representa matrizes de valores reais, alocadas dinamicamente, e de dimensões $m$ por $n$ fornecidas em tempo de execução.

Operações sobre o TAD Matriz:

1. *cria*: cria uma matriz de dimensão $m$ por $n$.
2. *libera*: libera a memória alocada para a matriz.
3. *acessa*: acessa o elemento da linha $i$ e da coluna $j$ da matriz.
4. *atribui*: atribui o elemento da linha $i$ e da coluna $j$ da matriz.
5. *linhas*: retorna o número de linhas da matriz.

• O ato de esconder a implementação de um TAD flexibiliza o projeto das suas funções, sem que isto afete o trabalho do usuário.

• **Exemplo:** TAD matriz
– Representa matrizes de valores reais, alocadas dinamicamente, e de dimensões $m$ por $n$ fornecidas em tempo de execução.

Operações sobre o TAD Matriz:

1. *cria*: cria uma matriz de dimensão $m$ por $n$.
2. *libera*: libera a memória alocada para a matriz.
3. *acessa*: acessa o elemento da linha $i$ e da coluna $j$ da matriz.
4. *atribui*: atribui o elemento da linha $i$ e da coluna $j$ da matriz.
5. *linhas*: retorna o número de linhas da matriz.
6. *colunas*: retorna o número de colunas da matriz.
7. ...

- A interface *Matriz.h*:

```
/* TAD: Matriz m por n */

typedef struct matriz Matriz ;

Matriz *mat_cria(int m, int n) ;
void mat_libera(Matriz *mat) ;
float mat_acessa(Matriz *mat, int i, int j) ;
void mat_atribui(Matriz *mat, int i, int j, float v) ;
int mat_linhas(Matriz *mat) ;
int mat_colunas(Matriz *mat) ;
```

• A implementação pode ser feita por matrizes dinâmicas representadas por **vetores simples** ou matrizes dinâmicas representadas por **vetores de ponteiros**.

• A interface do módulo independe da estratégia de implementação adotada (pode-se mudar a implementação sem afetar o uso do TAD).

• Se usarmos vetores simples, a estrutura da matriz pode ser:

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

- Se usarmos vetores de ponteiros:

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

Exercício:

- Implementar as funções que compõem o TAD Matriz.

• Resumidamente:

– Um TAD é um tipo de dados (conjunto de valores e de operações sobre os dados) que é acessado apenas através de uma **interface**. O programa que usa o TAD é denominado **cliente** e o programa que especifica o tipo de dados, **implementação**.

- Bibliografia:

Waldemar Celes et alli. *Introdução a Estruturas de Dados*, Elsevier, 2004