Programação 2

Matrizes e Tipos Abstratos

Rivera

Matrizes

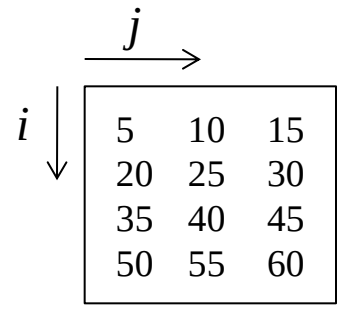
- Vetores bidimensionais – matrizes
  - Estáticos

int m [4][3]

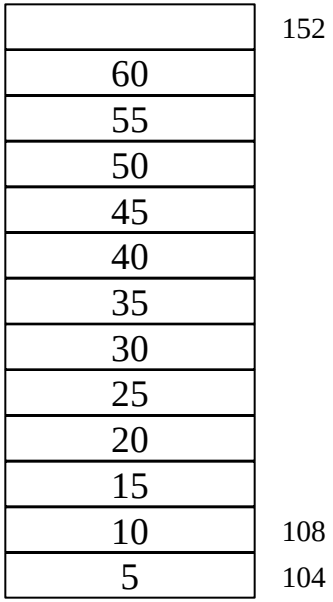
Reserva espaço contínuo de 12 elementos

Ex.

```
int m[4][3] = {   { 5, 10, 15 }
                  {20, 25, 30}
                  {35, 40, 45}
                  {50, 55, 60}  };
int m[  ][3] = {5, ....., 60};
int mv[12] = {5, 10, 15, 20, ..., 60};
```



m [i][j] = mv [3 * i + j]



Matrizes

- Dinâmica
  - Matriz representada por um vetor
    - Memória contínua suficiente para os elementos
    - Matriz M[n][m], onde m = num de colunas
      - Representada por vetor v[n x m]
      - M [i][j] = v [i * m + j]

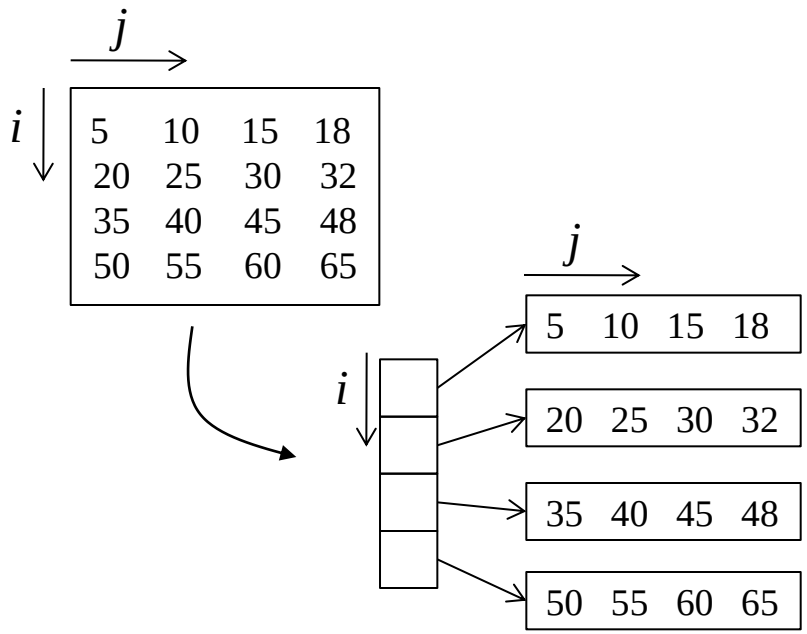
```
float *v;
...
v = (float*) malloc (n*m*sizeof(float));
```

Matrizes

- Dinâmica
  - Matriz representada por um vetor de ponteiros
    - Memória contínua para cada linha
    - Matriz M[n][m], onde m = num de colunas
      - M [i][j] elemento i e j

```
int i;
int **mat;
...
mat = (int**) malloc(m*sizeof(int*));
for (i=0; i<m; i++)
  mat[i] = (int*) malloc(n*sizeof(int));

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



Tipos Abstratos de Dados

- Módulos
  - Um arquivo com funções relacionados (parte de um programa completo)

str.c

```
int comprimento (char* str)
{
  ...
}

void copia (char* dest, char* or
{
  ...
}

void concatena (char* dest, char
{
  ...
}
...
```

vetor.c

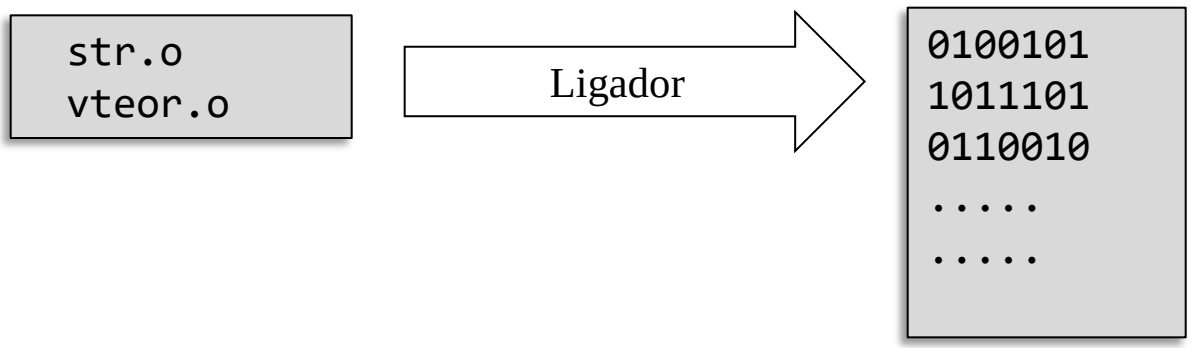
```
int compara (int n, int* v1, int* v2)
{
  ...
}

float norma(int n, int* v)
{
  ...
}

int elemento (int n, int* v)
{
  ...
}
...
```

Tipos Abstratos de Dados

- Arquivo objeto
  - Resultado de compilar um módulo
  - Geralmente com extensão *.o ou *.obj
- Ligador
  - Junta todos os arquivos objetos e transforma em um único arquivo executável.



Módulos

- Exemplo

  Prog1.c

```
#include <stdio.h>
int comprimento (char* str);
void copia (char* dest, char* orig);
void concatena (char* dest, char* orig);

int main (void)
{
  char str[101], atr1[51], str2[51];
  printf("Digite uma seqüência de caracteres: ");
  scanf(" %50[^\n]", str1);
  printf("Digite outra seqüência de caracteres: ");
  scanf(" %50[^\n]", str2);
  copia(str, str1);
  concatena(str, str2);
  printf("Comprimento da concatenação: %d\n",comprimento(str));
  return 0;
}
```

Módulos

- Exemplo

  Prog1.exe

  1) Compilar fontes str.c e prog1.c separadamente
  2) Ligar os arquivos resultantes em um único arquivo

```
> gcc –c str.c –o str.o
> gcc –c prog1.c –o prog1.o
> gcc –o prog.exe str.o prog1.o
```

Módulos

- Interfaces de um módulo de funções
  - Arquivo contendo apenas;
    - Os protótipos de funções oferecidas pelo módulo
    - Os tipos de dados exportados pelos módulos
  - Em geral possui:
    - Nome: o mesmo do módulo ao qual está associado
    - Extensão: *.h
  - Exemplo: str.h

```
int comprimento (char* str);
void copia (char* dest, char* orig);
void concatena (char* dest, char* orig);
```

Módulos

- Exemplo

  Prog1.c

```
#include <stdio.h>   // protótipos das funções da bib padrão de C

#include “str.h”     // protótipos de módulos do usuário

int main (void)
{
  char str[101], atr1[51], str2[51];
  printf("Digite uma seqüência de caracteres: ");
  scanf(" %50[^\n]", str1);
  printf("Digite outra seqüência de caracteres: ");
  scanf(" %50[^\n]", str2);
  copia(str, str1);
  concatena(str, str2);
  printf("Comprimento da concatenação: %d\n",comprimento(str));
return 0;
}
```

Tipo Abstrato de Dados

- TAD
  - Um TAD define
    - Um novo tipo de dado
    - O conjunto de operações para manipular dados desse tipo
  - Um TAD facilita
    - A manipulação e a reutilização de código
    - Abstrato: "forma de implementação não precisa ser conhecida"
      - Para usar, conhecer
        - » Funcionalidade, mas não a sua implementação

```c
/* TAD: Ponto.h */

 struct circulo { float x, y;} Ponto;
 Ponto* pto_cria (float, float);
 void pto_libera (Ponto* );
 void pto_acesso (Ponto*, float*, float*);
 void pto_atribui (Ponto*, float, float);
 float pto_distancia (Ponto*, Ponto*);
```

```c
#include “ponto.h”

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);
}

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

```c
...
void pto_libera (Ponto* p1)
{
  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;
}
```

```c
#include <stdio.h>
#include “ponto.h”

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

matriz.h

```
/* TAD: matriz m x n  como vetor */
typedef struct matriz
    {int lin; int col; float* v} 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);
```

Implementar as funções.

matriz.h

```
/* TAD: matriz m por n */
typedef struct matriz
    {int lin; int col; float** vv} 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);
```