Linguagem C

Arquivos

Clodoaldo A. M. Lima

Arquivos

- Podem armazenar grande quantidade de informação;
- Dados são persistentes (gravados em disco).

Tipos de Arquivos

- Texto
  - Armazena caracteres que podem ser mostrados diretamente na tela ou modificados por um editor de textos simples.
  - Conhecido por arquivo seqüencial.
  - Tipo especial de arquivo que pode ser editado normalmente através de um editor de textos qualquer.
  - É dito seqüencial porque a leitura tem que ser feita seqüencialmente do início ao fim do arquivo.

Tipos de Arquivos

- Binário
  - Seqüência de bits sujeita às convenções dos programas que o gerou.
  - Conhecido por Arquivo randômico, ou de acesso aleatório.
    - É possível buscar uma determinada informação em qualquer posição, sem necessidade de percorrer todo o arquivo até alcançar a informação desejada.
    - O acesso a informação é direto.
  - Exemplos:
    - Arquivos executáveis, arquivos compactados, arquivos de registros.

Manipulação de Arquivos

- Passos:
  - Criar um ponteiro para a estrutura FILE (definida em stdio.h)
  - Abrir o arquivo.
  - Ler ou gravar dados no arquivo.
  - Fechar o arquivo.

Funções

- fopen("nome", "modo")
  - Abre um arquivo e retorna um ponteiro para ele.
  - Caso não consiga abrir o arquivo, retorna NULL.
- fprintf(fp, "formato", arg1, ...)
  - Escreve para um arquivo
- fclose(fp)
  - Fecha um arquivo
- getc(fp)
  - Obtém um caractere.
- fscanf(fp, "formato", arg1, ...)
  - Lê de um arquivo.
- fwrite(*prt, tamanho_bloco , n, fp)
  - Escreve n blocos de *ptr para o arquivo.

Funções

- fread(*ptr, tamanho_bloco, n, fp)
- fseek(fp, posicao, modo)
  - Altera a posição no arquivo.
  - Deslocamento relativo ao:
    - SEEK_SET – início do arquivo
    - SEEK_CUR – ponto inicial
    - SEEK_END – final do arquivo
- fflush(fp)
  - Realiza a gravação efetiva do arquivo
- ftell(fp)
  - Indica a posição corrente do ponteiro do arquivo.

Modos de Operação

modo	operações	ponto no arquivo
r	leitura	início
r+	leitura e escrita	início
w	escrita	início
w+	leitura e escrita	início
a	escrita	final
a+	leitura	início
	escrita	final

Escrita de Arquivo Texto

```c
#include <stdio.h>

int main() {
   FILE *arq;
   arq = fopen("teste1.txt", "w");
   fprintf(arq, "Ola, Mundo!!!\n");
   fclose(arq);
   return 0;
}
```

Leitura de Arquivo Texto

```c
#include <stdio.h>

int main() {
   FILE *arq;
   char c;
   arq = fopen("teste.txt", "r");
   c = getc(arq);
   while (c != EOF) {
      printf("%c", c);
      c = getc(arq);
   }
   fclose(arq);
   return 0;
}
```

Somatório dos números de um arquivo

```c
#include <stdio.h>

int main() {
   FILE *arq;
   int numero, soma = 0;
   arq = fopen("numeros.txt", "r");
    If
   while (fscanf(arq, "%d", &numero) != EOF) {
      soma = soma + numero;
   }
   printf("A soma e: %d\n", soma);
   fclose(arq);
   return 0;
}
```

Entrada de dados

```
#include <stdio.h>

int main(int argc, char *argv[]) {
   FILE *arq;
    int a=1;
   char b[30]="Clodoaldo";
   float c=2.3;
   if (argc==2){
      arq = fopen(argv[1], "w");
      fprintf(arq,"%d %s %.1f",a,b,c);
     fclose(fp)
    else
      printf("É necessário especificar o nome
       do arquivo\n );
   return 0;
}
```

Modificando fgets

```c
#include <stdio.h>
#include <string.h>

int main() {
   FILE *arq;
   char str[80];
    arq = fopen("numero.txt", "r");
    fgets(str,79,arq);
     printf("%s\n",str);
     return 0;
}
```

Salvando uma estrutura

```c
#include <stdio.h>
#include <string.h>

struct pessoa {
  char nome[50];
  int idade;
};

typedef struct pessoa Pessoa;

int main() {
  FILE *arq;
  arq = fopen("pessoas.txt", "ab");
  Pessoa p;
  printf("Digite um nome: ");
  gets(p.nome);
  printf("Digite a idade: ");
  scanf("%d", &p.idade);
  fwrite(&p, sizeof(Pessoa), 1, arq);
  fclose(arq);
  return 0;
}
```

Lendo todo o arquivo

```c
#include <stdio.h>
#include <string.h>

struct pessoa {
  char nome[50];
  int idade;
};

typedef struct pessoa Pessoa;

int main() {
  FILE *arq;
  arq = fopen("pessoas.txt", "r");
  Pessoa p;
  while(fread(&p, sizeof(Pessoa), 1, arq) != 0) {
    printf("Nome: %s - Idade: %d\n", p.nome, p.idade);
  }
  fclose(arq);
  return 0;
}
```

Registros

Clodoaldo A. M. Lima

Registros

- São variáveis compostas heterogêneas.
- São conjuntos de dados logicamente relacionados, mas de tipos diferentes (inteiro, real, string, etc.)
- Os elementos dos registros são chamados de campos.
- **Exemplo:** Dados sobre funcionários de uma empresa:
  - Nome
  - Idade
  - Salário

Declaração

```
struct nome_do_tipo_do_registro {
    tipo1 campo1;
    tipo2 campo2;
    tipo3 campo3;
    // ...
    tipon campon;
};
```

```
struct funcionario {
    char nome[50];
    int idade;
    float salario;
};
```

Acesso a campos de um registro

- Pode ser realizado através da seguinte sintaxe:
  - **`nome_do_registro.nome_do_campo`**
- Para uma variável f do tipo funcionario:
  - **`struct funcionario f;`**
- O campo nome é acessado assim:
  - **`f.nome`**

Exemplo

```c
#include <stdio.h>
#include <string.h>

struct funcionario {
    char nome[50];
    int idade;
    float salario;
};

int main() {
    struct funcionario f;
    strcpy(f.nome,"Clodoaldo");
    f.idade = 18;
    f.salario = 1000;
    printf("Nome: %s\n", f.nome);
    printf("Idade: %d\n", f.idade);
    printf("Salario: %.2f\n", f.salario);
    return 0;
}
```

Vetor de Registros

- Declaração:
  - `struct nome_do_registro nome_da_variavel[tamanho_do_vetor];`
- Uso:
  - **`nome_da_variavel[indice].nome_do_campo`**

Exemplo

```c
#include <stdio.h>
#include <string.h>

struct pessoa {
  char nome[50];
  int idade;
};

int main() {
  struct pessoa p[2];
  strcpy(p[0].nome,"Jose");
  p[0].idade = 18;
  strcpy(p[1].nome,"Maria");
  p[1].idade = 25;
  printf("Nome: %s - Idade: %d\n", p[0].nome, p[0].idade);
  printf("Nome: %s - Idade: %d\n", p[1].nome, p[1].idade);
  return 0;
}
```

Exercício 2

- Considerando o registro de um produto de uma loja contendo as seguintes informações:
  - descricao, valor

- Fazer um programa que, dado o registro de 50 produtos, exiba-os na ordem inversa em que foram digitados.

Resolução Exercício

```c
#include <stdio.h>
#include <string.h>
#define MAX 5
int main() {
   struct produto {
     char descricao[40];
     float valor;
   };
   struct produto prods[MAX];
   int i;
   for (i=0; i<MAX; i++) {
      puts("Nome do Produto?");
      gets(prods[i].descricao);
      puts("Valor do Produto?");
      scanf("%f", &prods[i].valor);
      getchar();
   }
   for (i=MAX-1; i>=0; i--) {
      printf("Descricao: %s - Valor: R$ %.2f\n",
             prods[i].descricao, prods[i].valor);
   }
   return 0;
}
```

Definição de Tipos

Clodoaldo A M LIma

Definição de Tipos

- Em C é possível criar um tipo que faz exatamente a mesma coisa de um outro tipo já existente.

- Vantagens:
  - Facilitar a legibilidade do código;
  - Evitar alterações em vários pontos do código caso o tipo seja modificado.

Declaração de Tipos

- A declaração de um novo tipo é realizada a partir do comando **`typedef`**:
  - **`typedef <tipo_existente> <novo_tipo>;`**
- Usualmente a declaração de um novo tipo é feita fora da função main() para ficar global.
- Exemplo:
  - **`typedef float nota;`**

Exemplo

```c
#include <stdio.h>

typedef float nota;

int main () {
   nota n1;
   printf ("Digite a primeira nota: ");
   scanf ("%f", &n1);
   printf ("A primeira nota foi %f\n", n1);
   return 0;
}
```

Exemplo

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

struct pessoa {
  char nome[50];
  int idade;
};

typedef struct pessoa Pessoa;

int main() {
  Pessoa p;
  strcpy(p.nome,"Clodoaldo");
  p.idade = 18;
  printf("Nome: %s - Idade: %d\n", p.nome, p.idade);
  return 0;
}
```

Definindo uma estrutura

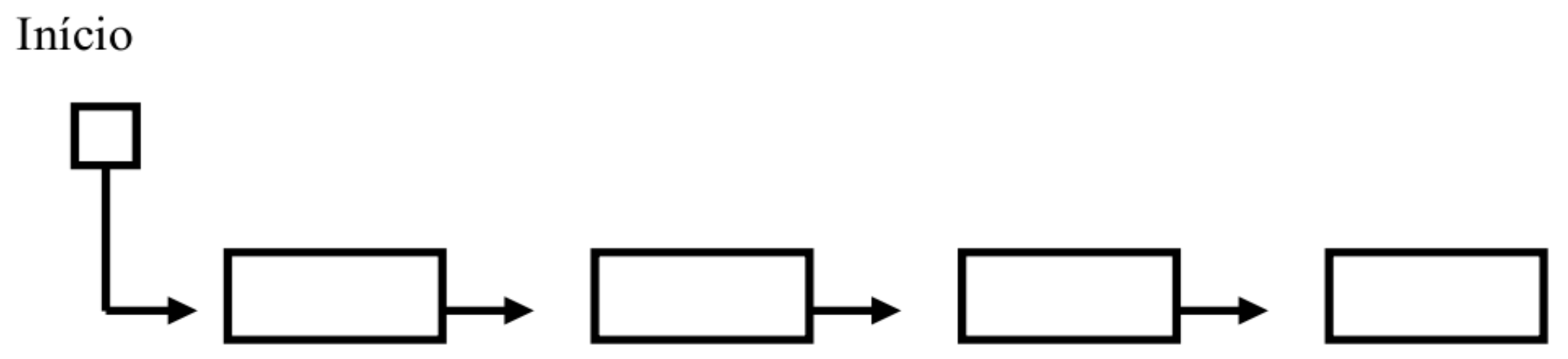
```c
#include <stdio.h>
#include <string.h>

struct pessoa {
  char nome[50];
  int idade;
};

typedef struct pessoa Pessoa;

int main() {
  Pessoa p[2];
  strcpy(p[0].nome,"Clodoaldo");
  p[0].idade = 18;
  strcpy(p[1].nome,"Maria");
  p[1].idade = 25;
  printf("Nome: %s - Idade: %d\n", p[0].nome, p[0].idade);
  printf("Nome: %s - Idade: %d\n", p[1].nome, p[1].idade);
  return 0;
}
```

Linguagem C

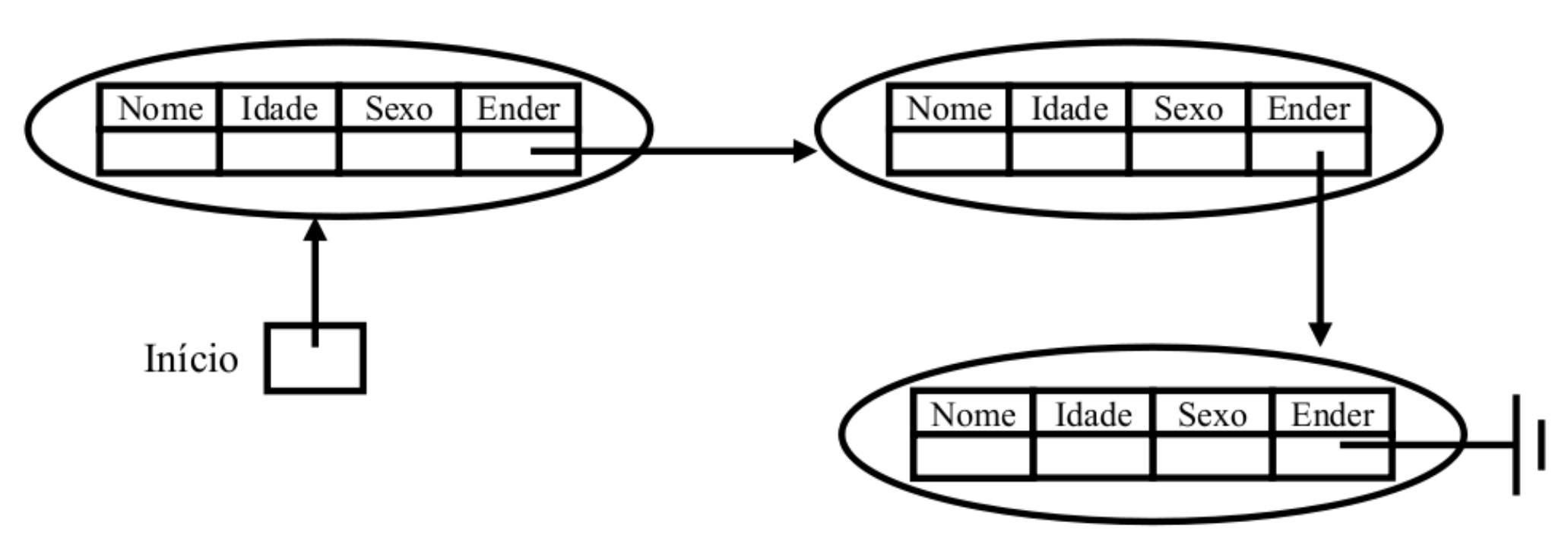
Lista Encadeada Simples

Clodoaldo A. M. Lima

Lista Encadeada Simples



Lista Encadeada Simples



Lista Encadeada Simples

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct pessoa Pessoa;

struct pessoa {
  char nome[50];
  int idade;
  Pessoa *proximo;
};

int main() {
  Pessoa *pAtual, *pInicio = NULL, *pAnterior = NULL;
  char continua;
  do {
    pAtual = malloc(sizeof(Pessoa));
    printf("Digite um nome: ");
    gets(pAtual->nome);
    printf("Digite a idade: ");
    scanf("%d", &pAtual->idade); getchar();
    if (pInicio == NULL) {
      pInicio = pAtual;
    }
    if (pAnterior != NULL) {
      pAnterior->proximo = pAtual;
    }
    pAnterior = pAtual;
    printf("Insere mais (S/N)? ");
    continua = getchar(); getchar();
  } while (toupper(continua) != 'N');
  pAtual = pInicio;
  while (pAtual != NULL) {
    printf("Nome: %s - Idade: %d\n", pAtual->nome, pAtual->idade);
    pAtual = pAtual->proximo;
  }
}
```

Exercicio

- Com base na estrutura criada anteriormente, fazer uma função que apaga o registro de uma determinada pessoa.

- Entrega 12/03 via COL

Modularização

Clodoaldo A. M. Lima

Modularização em C

- Programa em C pode ser dividido em vários arquivos
  - Arquivos fonte com extensão .c
  - Denominados de módulos
- Cada módulo deve ser compilado separadamente
  - Para tanto, usa um compilador
  - Resultado: arquivos objetos não executáveis
    - Arquivos em linguagem de máquina com extensão .o ou .obj
- Arquivos oibjeto devem ser juntados em um executável
  - Para tanto, usa-se um ligador ou link-editor
  - Resultado: um único arquivo em linguagem de máquina
    - Usualmente com extensão .exe

Modularização em C

- Módulos são muito úteis para construir bibliotecas de funções inter-relacionadas. Por exemplo:
  - Módulos de funções para manipulação de strings
  - Módulos de funções matemáticas
  - etc
- Em C, é preciso listar no início de cada módulo aquelas funções de outros módulos que serão utilizadas:
- Isso é feito através de uma lista denominada cabeçalho
- Exemplo: considere um arquivo STR.c contendo funções para manipulação de strings, dentre elas:
  - **int** comprimento (char* strg)
  - void copia (char* dest, char* orig)
  - void concatena (char* dest, char* orig)

Modularização em C

- Exemplo (cont): Qualquer módulo que utilizar essas funções deverá incluir no início o cabeçalho das mesmas, como abaixo.

```
/* Programa Exemplo.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], str1[51], str2[51];
    printf("Entre com uma seqüência de caracteres: ");
    scanf(" %s\n", str1);
    printf("Entre com outra seqüência de caracteres: ");
    scanf(" %s\n", str2);
    copia(str, str1);
    concatena(str, str2);
    printf("Comprimento total: %d\n",comprimento(str));
    return 0; }
```

Modularização em C

- A partir desses dois fontes (Exemplo.c e STR.c), podemos gerar um executável compilando cada um separadamente e depois ligando-os
- Por exemplo, com o compilador Gnu C (gcc) utilizaríamos a seguinte seqüência de comandos para gerar o arquivo executável Teste.exe:
  - > gcc –c STR.c
  - > gcc –c Exemplo.c
  - > gcc –o Teste.exe STR.o Exemplo.o
- Questão:
  - É preciso inserir manualmente e individualmente todos os cabeçalhos de todas as funções usadas por um módulo?
  - E se forem muitas e de diferentes módulos?

Modularização em C

- Solução
- Arquivo de cabeçalhos associado a cada módulo, com:
  - cabeçalhos das funções oferecidas pelo módulo e, eventualmente, os tipos de dados que ele exporta typedefs, structs, etc.
- Segue o mesmo nome do módulo ao qual está associado
  - porém com a extensão .h
- Exemplo:
  - Arquivo STR.h para o módulo STR.c do exemplo anterior

Modularização em C

- O programa Exemplo.c pode então ser rescrito como:
- /* Programa Exemplo.c */
- #include <stdio.h> /* Módulo da Biblioteca C Padrão */
- #include "STR.h" /* Módulo Próprio */
- int main (void) {
  - char str[101], str1[51], str2[51];
  - printf("Entre com uma seqüência de caracteres: ");
  - scanf(" %s\n", str1);
  - printf("Entre com outra seqüência de caracteres: ");
  - scanf(" %s\n", str2);
  - copia(str, str1);
  - concatena(str, str2);
  - printf("Comprimento total: %d\n",comprimento(str));
  - return 0; }
- Nota: O uso dos delimitadores < > e " " indica onde o compilador deve procurar os arquivos de cabeçalho, na biblioteca interna (<>) ou no diretório indicado (" " – default se ausente).

Modularização em C

- Módulos podem ser usados para definir um novo tipo de dado e o conjunto de operações para manipular dados desse tipo:
  - Tipo Abstrato de Dados (TAD)

- Definindo um tipo abstrato, pode “esconder” a implementação
  - Quem usa o tipo abstrato precisa apenas conhecer a funcionalidade que ele implementa, não a forma como ele é implementado

  - Facilita manutenção e re-uso de códigos, entre outras vantagens

Modularização em C

- /* Matriz m por n (m e n >= 1)*/
- struct Matriz {
  - int lin;
  - int col;
  - float* v;
- };
- /* Tipo Exportado */
- typedef struct matriz Matriz;
- /* Funções Exportadas */
- /* Função cria - Aloca e retorna matriz m por n */
  - Matriz* cria (int m, int n);
- /* Função libera - Libera a memória de uma matriz */
  - void libera (Matriz* mat);
- Arquivo matriz.h

Modularização em C

- /* Continuação... */
- /* Função acessa - Retorna o valor do elemento [i][j]*/
- float acessa (Matriz* mat, int i, int j);

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

- /* Função linhas - Retorna o no. de linhas da matriz*/
- int linhas (Matriz* mat);

- /* Função colunas - Retorna o no. de colunas da matriz */
- int colunas (Matriz* mat);

Matriz.h

```
struct matriz {
int lin;
int col;
float* v;
};
typedef struct matriz Matriz;
Matriz* cria (int m, int n) {
   Matriz* mat = (Matriz*) malloc(sizeof(Matriz));
   if (mat == NULL) {
        printf("Memória insuficiente!\n");
        exit(1);}
mat->lin = m;
mat->col = n;
mat->v = (float*) malloc(m*n*sizeof(float));
return mat;
}
```

Matriz.h

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

float acessa (Matriz* mat, int i, int j) {
int k; /* índice do elemento no vetor – armazenamento por linha*/
if ((i<1) || (i>mat->lin) || (j<1) || (j>mat->col)) {
    printf("Acesso inválido!\n");
    exit(1);}
k = (i-1)*mat->col + j - 1;
return mat->v[k];}

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

Matriz.h

```
void atribui (Matriz* mat, int i, int j, float v) {
    int k; /* índice do elemento no vetor */
    if ((i<1) || (i>mat->lin) || (j<1) || (j>mat->col)) {
        printf("%d %d",i,j);
        printf("Atribuição inválida!\n");
        exit(1);
    }
    k = (i-1)*mat->col + j -1;
    mat->v[k] = v;
    }

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

Programa Principal

```
#include <stdio.h>
#include <stdlib.h>
#include "matriz.h"
int main()
{
    float a,b,c,d;
    Matriz *M;
    // criação de uma matriz
    M = cria(5,5);
    // inserção de valores na matriz
    atribui(M,1,2,40);
    atribui(M,2,3,3);
    atribui(M,3,5,15);
    atribui(M,5,1,21);
```

Programa Principal

```
/* Continuação... */
// verificando se a inserção foi feita corretamente
    a = acessa(M,1,2);
    b = acessa(M,2,3);
    c = acessa(M,3,5);
    d = acessa(M,5,1);
    printf ("M[1][2]:%4.2f\n", a);
    printf ("M[2][3]:%4.2f\n", b);
    printf ("M[3][5]:%4.2f\n", c);
    printf ("M[5][1]:%4.2f\n", d);
    getchar();
    return 0;
}
```