SPOEDD1 – Estrutura de Dados

IFSP – Campus São Paulo

Antonio Ferreira Viana

2024

Agenda

1. Revisão de Linguagem C

Revisão de Linguagem C

- **Estrutura geral de um programa em linguagem C**

```
#include <stdio.h>                  // #include serve para incluir as bibliotecas necessárias
#include <stdlib.h>

#define PI 3.141618                 // definicao de uma constante chamada PI

typedef  int  inteiro ;             // exemplo de criação de um tipo novo, chamado inteiro

int main( ) {
inteiro x;
printf ("\nDigite um numero inteiro positivo => ");
scanf ("%d",  &x);                  // no scanf é obrigatório o uso do &
                                    // o %d  indica que será lido um numero inteiro decimal
                                    // o & indica que está sendo passado parâmetro por referência

if ( x % 2 == 0 )
     printf("\nVoce digitou o numero %d, que eh par.\n" ,  x);   // usa-se \n para pular uma linha
else
     printf("\nVoce digitou o numero %d, que eh impar.\n" ,  x);
}
```

Revisão de Linguagem C

- **Comandos para leitura de dados pelo teclado**
  - **getchar()** // ler um caractere
  - **getc** // ler um caractere
  - **gets** // ler uma string que pode ter espaços em branco
  - **scanf** // ler variáveis indicando o formato, indicados no primeiro // parâmetro do comando

- **Comandos para escrita/exibição de dados na tela**
  - **putchar()** // exibir um caractere
  - **putc** // exibir um caractere
  - **puts** // exibir uma string
  - **printf** // exibir conteúdo de variáveis indicando o formato, indicados // no primeiro do comando

Revisão de Linguagem C

- **Exemplo de uso das funções de leitura e exibição na tela**

```
#include <stdio.h>
int main()
{
  char ch;
  printf("Digite algum caracter: ");
  ch = getchar();
  printf("\n A tecla pressionada eh %c.\n", ch);
  return (0);
}
```

```
#include <stdio.h>
 int main()
{
    char ch = 'G';          // Get the character to be written
    putchar(ch);            // Write the Character to stdout
    return (0);
}
```

Revisão de Linguagem C

- **Exemplo de uso das funções de leitura e exibição na tela**

```
#include <stdio.h>
int main()
{
   char buffer[20];        // declaração de um string de tamanho 10, vetor de tamanho 10 de caracteres
   printf("\nEntre com o seu nome completo: ");

   gets(buffer);           // leitura do string, que pode conter espaço em branco

   printf("O nome é: %s.\n", buffer);          // exibição com printf

   puts("O nome é: ");
   puts(buffer);                               // exibição com puts
   return 0;
}
```

Revisão de Linguagem C

- **Comandos de repetição**

Comando while

O comando while permite que um certo trecho de programa seja executado ENQUANTO uma certa condição for verdadeira. A forma do comando while é a seguinte:

```
while (condição)
{
        // comandos a serem repetidos
        // comandos a serem repetidos
}
// comandos após o 'while'
```

Revisão de Linguagem C

- **Comandos de repetição**

Comando while

O funcionamento é o seguinte:

1. Testa a condição;
2. Se a condição for falsa então pula todos os comandos do bloco subordinado ao while e passa a executar os comandos após o bloco do while.
3. Se condição for verdadeira então executa cada um dos comandos do bloco subordinado ao while.
4. Após executar o último comando do bloco do while volta ao passo 1.

O comando while deve ser usado sempre que:

- **não soubermos exatamente quantas vezes o laço deve ser repetido**;
- **o teste deve ser feito antes de iniciar a execução** de um bloco de comandos;
- houver casos em que o laço não deva ser repetido nenhuma vez.

Revisão de Linguagem C

- **Comandos de repetição**

Comando do..while

O comando **do..while** permite que um certo trecho de programa seja executado ENQUANTO uma certa condição for verdadeira. A forma do comando do..while é a seguinte:

```
do
{
        // comandos a serem repetidos
        // comandos a serem repetidos
}
 while (condição);
// comandos após o 'do..while'
```

Revisão de Linguagem C

- **Comandos de repetição**

Comando do..while

O funcionamento é o seguinte:

1. Executa os comando dentro do bloco do-while;
2. Testa a condição;
3. Se a condição for falsa então sai da repetição, executa o comando que está logo após o bloco subordinado ao do-while;
4. Se condição for verdadeira então volta ao passo 1.

O comando do..while deve ser usado sempre que:

- **que não soubermos exatamente quantas vezes o laço deve ser repetido**;
- o **teste deve ser feito depois da execução** de um bloco de comandos;
- o bloco de comandos **deve se executado pelo menos 1 vez**;

Revisão de Linguagem C

- **Comandos de repetição**

Comando for

O comando for permite que um certo trecho de programa seja executado um determinado número de vezes. A forma do comando for é a seguinte:

```
for (comandos de inicialização; condição de teste; incremento/decremento)
{
        // comandos a serem repetidos
        // comandos a serem repetidos
}

// comandos após o 'for'
```

Revisão de Linguagem C

- **Comandos de repetição**

Comando for

O funcionamento é o seguinte:

1. Executa os comandos de inicialização;
2. Testa a condição;
3. Se a condição for falsa então executa o comando que está logo após o bloco subordinado ao for .
4. Se condição for verdadeira então executa os comandos que estão subordinados ao for;
5. Executa os comandos de incremento/decremento;
6. Volta ao passo 2.

O comando **for** deve ser usado sempre que:

- **soubermos exatamente quantas vezes o laço deve ser repetido**;
- o teste deve ser feito **antes da execução** de um bloco de comandos;
- houver casos em que o laço não deva ser repetido nenhuma vez.

Revisão de Estrutura

- Exemplo de uso de typedef e uso de Estrutura (struct)

```
#include <stdio.h>
#define alturaMaxima 2.25

typedef struct {
        float peso;                 /* peso em quilogramas    */
        float altura;               /* altura em metros       */
} PesoAltura;

int main()  {

 PesoAltura      pessoa1;      // PesoAltura é o novo tipo de dados...

 pessoa1.peso    = 80;
 pessoa1.altura  = 1.85;

 printf ( "Peso: %.2f, Altura %.2f " ,pessoa1.peso, pessoa1.altura );

 if ( pessoa1.altura > alturaMaxima )
     printf("Altura acima da maxima.\n");
 else
     printf("Altura abaixo da maxima.\n");
 return 0;
}
```

Revisão de Estrutura

- Exemplo de uso de typedef e uso de Estrutura (struct)

```
#include <stdio.h>
#define alturaMaxima 2.25

typedef struct {
        float peso;                 /* peso em quilogramas    */
        float altura;               /* altura em metros       */
} PesoAltura;

int main()  {

 PesoAltura      pessoa[10];                 // PesoAltura é o novo tipo de dados.  A variável pessoa é um vetor de
                                                10 posições onde, cada posição é do tipo PesoAltura. Modo de
                                                tratar um vetor cujos elementos são elementos de uma struct

 pessoa[0].peso    = 80;
 pessoa[0].altura  = 1.85;
 printf("Peso: %.2f, Altura %.2f ", pessoa[0].peso, pessoa[0].altura);

 if ( pessoa[0].altura > alturaMaxima )
     printf("Altura acima da maxima.\n");
 else
     printf("Altura abaixo da maxima.\n");
 return 0;
}
```

Revisão de Ponteiros

- Na Linguagem C existe uma distinção bastante explícita entre um tipo (ou uma estrutura) e um **endereço**:

```
int x;          /* significa que x é uma variável do tipo inteiro */

int * y;        /* significa que y é uma variável que contém um endereço
                   memória onde está armazenado (que aponta para) um
                   número inteiro */
```

O asterisco (*) após a palavra int indica que estamos falando de um endereço para inteiro e não mais de um inteiro.

Revisão de Ponteiros

- Também usamos ponteiros quando manipulamos arquivos:

```
FILE  * Arq;  /* a variável Arq trata-se de um ponteiro (apontador) */
              /* para endereços de uma estrutura FILE                */
```

```c
/* Bibliotecas */
#include <stdio.h>
#include <locale.h>
#include <stdlib.h>
/* corpo do programa */
int main()
{
   /* Variáveis locais ao main */
   FILE * Arq; /* a variável Arq trata-se de um ponteiro (apontador) */
               /* para endereços de uma estrutura FILE               */
   setlocale (LC_ALL, "");
   system ("mode 100");

   Arq = fopen ("TestePonteiro.txt", "w");
   if ( Arq == NULL)
   {
      printf ("Erro ao abrir TestePonteiro.txt!");
      getch();
      exit(0);
   }
   fprintf(Arq, "Endereço de Arq: [%p]", &Arq);
   fclose (Arq);
   return 0;
}
```

Revisão de Ponteiros

- O que será exibido ao executar o programa abaixo, 25 ou 30 ?

```c
#include <stdio.h>

int   main()
{
    int    x  = 25;
    int *  y;
     y    = &x;
     *y   = 30;
     printf("Valor atual de x: %i \n", x);
     return 0;
}
```

Revisão de Ponteiros

- **O que será exibido ao executar o programa abaixo, 25 ou 30 ?**

```
#include <stdio.h>

int   main()
{
    int    x  = 25;
    int *  y;
    y         = &x;
    *y        = 30;
     printf("Valor atual de x: %i\n", x);
     return 0;
}
```

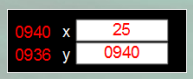
A variável x é inicializada com valor 25 (e está alocada no endereço de memória 0940).

Revisão de Ponteiros

- **O que será exibido ao executar o programa abaixo, 25 ou 30 ?**

```
#include <stdio.h>

int   main()
{
    int    x  = 25;
    int *  y;
     y        = &x;
     *y       = 30;
     printf("Valor atual de x: %i\n", x);
     return 0;
}
```



A variável x é inicializada com valor 25

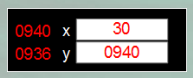
A variável **y** (que está alocada no **endereço** de memória **0936**) recebe o endereço de onde está a variável x.

Revisão de Ponteiros

- O que será exibido ao executar o programa abaixo, 25 ou 30 ?

```
#include <stdio.h>

int   main()
{
    int    x  = 25;
    int *  y  = &x;
     *y       = 30;
     printf("Valor atual de x: %i\n", x);
     return 0;
}
```



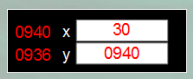
Coloca-se o valor 30 na posição de memória referenciada (apontada) pelo endereço armazenado em y (o conteúdo do endereço apontado por y (0940) recebe o valor 30).

Revisão de Ponteiros

- O que será exibido ao executar o programa abaixo, 25 ou 30 ?

```
#include <stdio.h>

int   main()
{
    int    x  = 25;                    // valor atual de x é 30....
    int *  y  = &x;
     *y       = 30;
     printf ( "Valor atual de x: %d \n " , x );
     return 0;
}
```



Coloca-se o valor 30 na posição de memória referenciada (apontada) pelo endereço armazenado em y (o conteúdo do endereço apontado por y (0940) recebe o valor 30).

Revisão de Ponteiros

- Para alocarmos memória dinamicamente na Linguagem C utilizamos a função malloc (memory allocation).
- Para usarmos a função **malloc** precisamos saber que:
  - Devemos incluir a biblioteca **stdlib.h** em nosso programa;
  - Devemos passar para malloc, como parâmetro, o número de bytes que se deseja alocar;
- A função **malloc retorna o endereço inicial** do bloco de bytes que foi alocado, porém esse retorno tem o tipo void * (ponteiro para void);
- Existe a função chamada **sizeof** que recebe como parâmetro um tipo (simples ou composto) e **retorna a quantidade de bytes necessária para tratar esse tipo**.

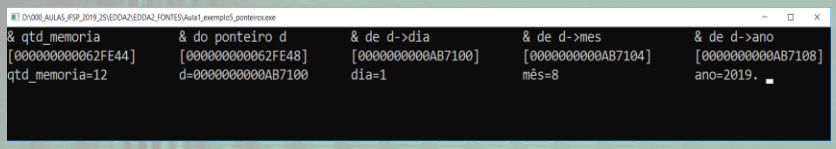
Revisão de Ponteiros

- Exemplo de uso da função malloc

```c
/* Bibliotecas */
#include <stdio.h>
#include <conio.h>
#include <locale.h>
#include <stdlib.h>
/* Tipos de dados definidos pelo programador */
typedef struct
{
   int dia, mes, ano;
}
data;
/* Corpo do programa */
int main()
{ /* Variáveis locais ao main */
  data *d;
  int  qtd_memoria;
  setlocale (LC_ALL, "");
  system ("mode 115,6");
  /* sizeof retornará a quantidade de bytes da estrutura data */
  qtd_memoria = sizeof (data);
  /* o ponteiro d apontará para o endereço de onde começa a memória alocada por malloc */
  d = (data *) malloc (qtd_memoria);
  /* os conteúdos da memória reservada (d) são preenchidos...*/
  d->dia = 1;    /*...com dia... */
  d->mes = 8;    /*...com mês... */
  d->ano = 2019; /*... e com ano.*/
  /* Monta cabeçalho para exibição dos dados */
  printf("& qtd_memoria\t\t& do ponteiro d\t\t& de d->dia\t\t& de d->mes\t\t& de d->ano\n");
  /* Exibe os endereços de memória de: qtd_memoria, d, d->dia, d->mes e d->ano */
  printf("[%p]\t[%p]\t[%p]\t[%p]\t[%p]\n", &qtd_memoria, &d, &d->dia, &d->mes, &d->ano);
  /* Exibe os conteúdos das variáveis: qtd_memoria, d, d->dia, d->mes e d->ano */
  printf("qtd_memoria=%d\t\td=%p\tdia=%d\t\t\tmês=%d \t\t\tano=%d. ", qtd_memoria, d, d->dia, d->mes, d->ano);
  getch();
  return 0;
}
```

Revisão de Ponteiros

- **Exemplo de uso da função malloc**



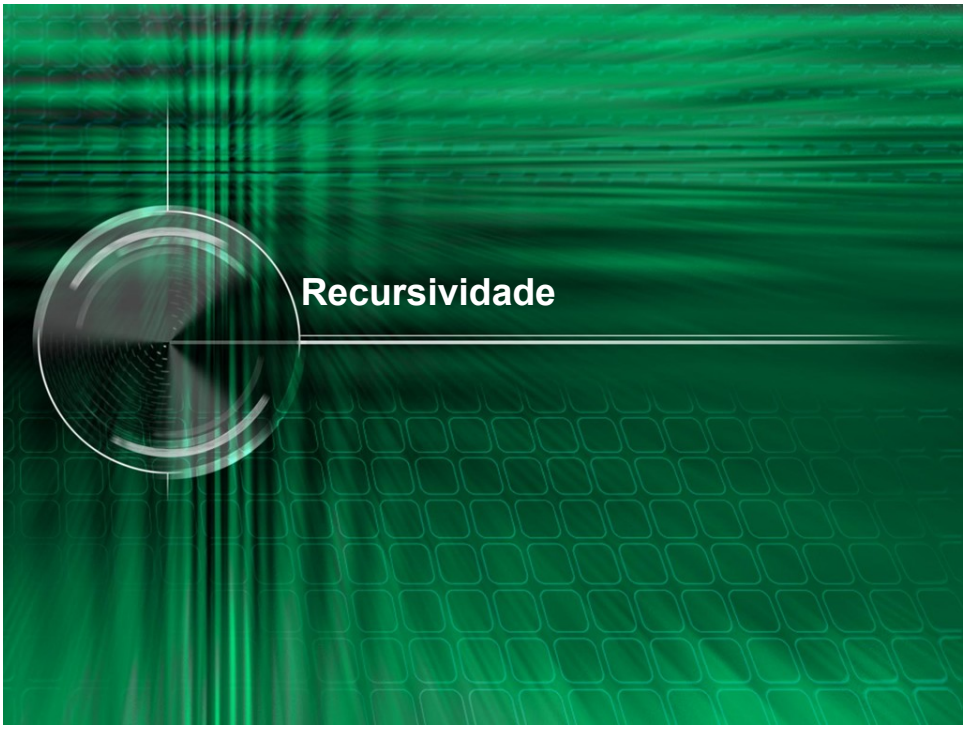
Revisão de Ponteiros

- **Exercícios sugeridos**

1. Discuta, passo a passo, o efeito do seguinte fragmento de código:

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

2. Elabore um programa em Linguagem C para testar esse fragmento de código identificando os endereços e conteúdos envolvidos.



Recursividade

- Recursividade é uma técnica de programação na qual uma função (ou método) pode chamar a si mesma.
- Ela é bastante usada para simplificar a resolução de problemas que demandariam muitos passos.

Recursividade

- **O fatorial é um número natural inteiro positivo, sendo representado por n!, na matemática.**

  O cálculo do fatorial de um número é obtido pela multiplicação desse número por todos os seus antecessores até chegar ao número 1.

  Assim: **3! = 3 * 2 * 1 = 6**
  4! = 4 * 3 * 2 * 1 = 24
  5! = 5 * 4 * 3 * 2 * 1 = 120

  Além disso, por definição:
  1! = 1
  0! = 1
  n! = n * (n-1)!

Recursividade

- **Exemplo de implementação em C de uma função recursiva para cálculo do fatorial de um número:**

```
int  fatorial_recursivo (int n)
{
      if ( (n == 0) || (n == 1) )   /* CASO BASE ou de PARADA*/
                return 1;
      else                          /* CASO RECURSIVO */
                return ( n * fatorial_recursivo (n-1) );
}
```

Recursividade

- Em geral, uma função definida recursivamente pode ser também definida de uma forma iterativa (através de estruturas de repetição).
- Em geral, a definição recursiva é mais "declarativa" – explicita o que se pretende obter e não a forma como se obtém (ou seja, o algoritmo que é usado).

Recursividade

- Por outro lado, uma definição iterativa, embora não permita uma compreensão tão imediata, é geralmente mais eficiente, dado que a implementação recursiva precisa registrar o estado atual do processamento para continuar de onde parou após a conclusão de cada nova execução, e isso consome tempo e memória.

Recursividade

- **Implementação de funções para cálculo do fatorial de um número (uma recursiva e outra iterativa)**

```c
/* Bibliotecas */
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <conio.h>
/* Funções */
int fatorial_recursivo(int n)
{
     if (n == 0 || n == 1)
          return 1;
     else
          return ( n * fatorial_recursivo (n-1) );
}

int fatorial_nao_recursivo (int n)
{
    int resultado, contador;
    resultado = 1;
    if ( n!=0 )
    {
       for(contador=n; contador >= 1; contador--)
       {
          resultado = resultado * contador;
       }
    }
    return resultado;
}
```

Recursividade

PROPOSTA:

Crie um programa em C para calcular a soma dos números inteiros entre 1 e n, onde n é fornecido pelo usuário como entrada.

Crie duas funções soma (uma recursiva e a outra não recursiva) que recebe como parâmetro de entrada o número lido.

NOTAR QUE:

soma_recursiva (n) =
- 1, se n = 1 (solução trivial, caso base)
- n + soma_recursiva (n-1), se n > 1 (solução recursiva)

Tempo de processamento

Medir tempo de processamento

Usando a função clock() da biblioteca <time.h>:

Para calcular o tempo de processamento de um determinado trecho de código, podemos utilizar a função clock() fornecida em <time.h>. Ela retorna o um valor em clock_t, que armazena o número total de tiques do relógio do computador.

Para calcular o número total de segundos decorridos, precisamos dividir o número total de tiques de relógio decorridos por CLOCKS_PER_SEC macro (também presente em <time.h>)

Medir tempo de processamento

Exemplo de uso:

```
#include <stdio.h>
#include <time.h>       // para  clock_t,  clock(), CLOCKS_PER_SEC
#include <unistd.h>     // para sleep()

int main()
{
   double time_spent = 0.0;                    // para armazenar o tempo de execução do código

   clock_t begin = clock();   // armazena o valor atual

   sleep(3);                  // faz algumas coisas aqui, gasta tempo de CPU

   clock_t end = clock();      // armazena o valor atual, após gastar tempo de processamento

   // calcula o tempo decorrido encontrando a diferença (end - begin) e
   // dividindo a diferença por CLOCKS_PER_SEC para converter em segundos

   time_spent += (double)(end - begin) / CLOCKS_PER_SEC;       // subtrai o tempo final do tempo inicial

   printf("The elapsed time is %f seconds", time_spent);              // exibe o tempo consumido
   return 0;
}
```

Medir tempo de processamento

Usando a função **clock()** da biblioteca <time.h>:

Para calcular o tempo de processamento de um determinado trecho de código, podemos utilizar a função clock() fornecida em <time.h>. Ela retorna o um valor em clock_t, que armazena o número total de tiques do relógio do computador.

Atenção: a função clock() **não retorna o tempo real decorrido**, **mas retorna o tempo gasto pelo sistema operacional para executar o processo**. Em outras palavras, o tempo real do relógio de parede pode ser muito maior.

Medir tempo de processamento

Usando a função time() da biblioteca <time.h>:

A função **time()** retorna o número total de segundos decorridos desde a Epoch (00:00:00 UTC, 1º de janeiro de 1970).

Seu uso é semelhante ao função clock(), como mostrado abaixo:

```
#include <stdio.h>
#include <time.h>          // para usar a função  time()
#include <unistd.h>        // para usar a função sleep()
int main()
{
   time_t begin = time(NULL);          // pega o horario atual

   sleep(3);                           // faz algumas coisas aqui

   time_t end = time(NULL);            // pega o horario atual

   // calcula o tempo decorrido encontrando a diferença (end - begin)
   printf("The elapsed time is %d seconds", (end - begin));
   return 0;
}
```

Medir tempo de processamento

Mediremos o tempo de processamento para mostrar como podemos decidir se um algoritmo é mais eficiente que outro, em relação do tempo de processamento.

Também podemos mostrar isso através de quantidade de memória consumida, quantidade de trocas realizadas, quantidade de instruções executadas, entre outras medidas que podem nos auxiliar a decidir por um algoritmo A ao invés de outro algoritmo B, para resolver o mesmo problema computacional.

Manipulação de arquivos

Manipulação de arquivos em C

Arquivos

Os arquivos permitem gravar os dados de um programa de forma permanente em em memória externa.

Vantagens de utilizar arquivos:

- Armazenamento permanente de dados: as informações permanecem disponíveis mesmo que o programa que as gravou tenha sido encerrado, ou seja, podem ser consultadas a qualquer momento.
- Grande quantidade dados pode ser armazenada: A quantidade de dados que pode ser armazenada depende apenas da capacidade disponível da unidade que será utilizada para armazenamento. Normalmente a capacidade da mídia é muito maior do que a capacidade disponível na memória RAM do computador.
- Possibilidade de ocorrer acesso concorrente: Vários programas podem acessar um arquivo de forma concorrente.

Manipulação de arquivos em C

Arquivos

A linguagem C trata os arquivos como uma sequência de bytes. Esta sequência pode ser manipulada de várias formas através, de funções para criar, ler e escrever (gravar) o conteúdo de arquivos independente de quais sejam os dados armazenados. Dois tipos:

Arquivo texto: Armazena caracteres que podem ser mostrados diretamente na tela ou modificados por um editor de texto. Exemplos de arquivos texto: documentos de texto, código fonte C, páginas XHTML.

Arquivo binário é uma sequência de bits que obedece regras do programa que o gerou. Exemplos: Executáveis, documentos do Word, arquivos compactados.

Manipulação de arquivos em C

Arquivos

Em Linguagem C, o arquivo é manipulado através de um ponteiro especial que “aponta” para o arquivo. A função deste ponteiro é “apontar” a localização de um registro.

Sintaxe:

FILE < *ponteiro >

Atenção:

O tipo **FILE** está definido na biblioteca **stdio.h**. (FILE deve ser escrito em letras maiúsculas).

Manipulação de arquivos em C

Arquivos do tipo texto

Operações com arquivos do tipo texto

Abertura de arquivos.

Para trabalhar com um arquivo, a primeira operação necessária é abrir este arquivo.

Sintaxe de abertura de arquivo:

< ponteiro > = fopen(“nome do arquivo”, ”tipo de abertura”);

A função **fopen** recebe como parâmetros o nome do arquivo a ser aberto e o tipo de abertura a ser realizado. Depois de aberto, realizamos as operações necessárias e fechamos o arquivo.

Manipulação de arquivos em C

Arquivos do tipo texto

Operações com arquivos do tipo texto

Fechamento do arquivo

Para fechar o arquivo usamos a função **fclose**.

Sintaxe de fechamento de arquivo

fclose (< ponteiro >);

Lembrando que o ponteiro deve ser a mesma variável ponteiro associada ao comando de abertura de arquivo.

Manipulação de arquivos em C

Arquivos do tipo texto

Tipos de abertura de arquivos (em minúsculo)

r : Permissão de abertura somente para leitura. É necessário que o arquivo já esteja presente no disco.

w : Permissão de abertura para escrita (gravação). Este código cria o arquivo caso ele não exista, e caso o mesmo exista ele recria o arquivo novamente fazendo com que o conteúdo seja perdido. Portanto devemos tomar muito cuidado ao usar esse tipo de abertura.

a : Permissão para abrir um arquivo texto para escrita(gravação), permite acrescentar novos dados ao final do arquivo. Caso não exista, ele será criado.

Manipulação de arquivos em C

Arquivos do tipo texto

Problemas na abertura de arquivos

Na prática, nem sempre é possível abrir um arquivo. Podem ocorrer algumas situações que impedem essa abertura, por exemplo:

- O programa está tentando abrir um arquivo no modo de leitura, mas o arquivo não existe;
- Não existe permissão para ler ou gravar no arquivo;
- O arquivo está bloqueado por estar sendo usado por outro programa.

No caso de um arquivo não poder ser aberto a função fopen retornará o valor NULL.

Manipulação de arquivos em C

Arquivos do tipo texto

Tratamento de erro ao abrir o arquivo

Deverá ser criar um trecho de código a fim de verificar se a abertura ocorreu com sucesso ou não.

Exemplo:

```
if (pont_arq == NULL)  {
   printf("\nERRO! Problema na abertura do arquivo!\n");
}
else  {
   printf("\nO arquivo foi aberto com sucesso!\n");
  }
```

Manipulação de arquivos em C

Arquivos do tipo texto

Funções para leitura de dados de um arquivo

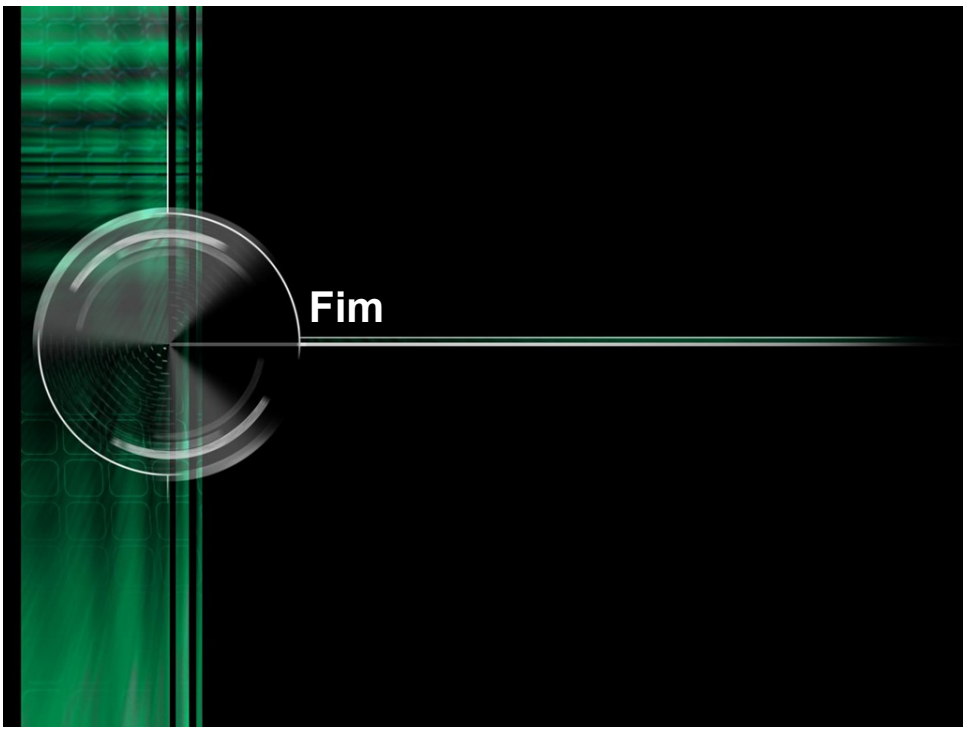
- fscanf()
- fgetc()
- fgets()

Manipulação de arquivos em C

Arquivos do tipo texto

Funções para gravação de dados em um arquivo

- fprintf()
- fputc()
- fputs()



Manipulação de arquivos em C

Tarefa – 01

Implementar em linguagem C uma aplicação que simule uma agenda de contatos, onde deverão ser cadastrados contatos com as seguintes informações: Nome do contato, e-mail do contato, número do telefone do contato (contendo informação de código regional), data de aniversário do contato.

Seu programa deverá exibir através de um menu as seguintes opções: 1-para cadastrar novo contato; 2-para alterar dados de um contato; 3-para excluir os dados de um contato; 4 para exibir os dados de um contato específico; 5-para exibir os dados de todos os contatos; 6-para sair da aplicação.

Os dados dos contatos deverão ser armazenados no arquivo DATA.TXT