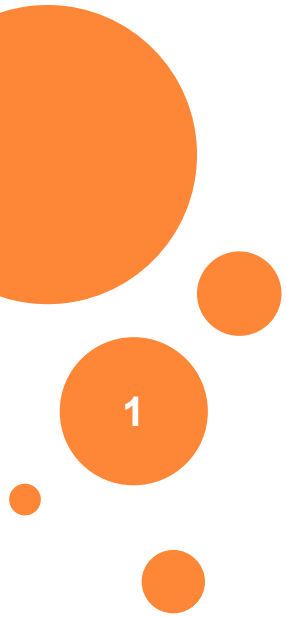


Linguagem C: Arquivos

Prof. Humberto Razente

Sala 1B144

Arquivos

- Por que usar arquivos?
  - Permitem armazenar grande quantidade de informação
  - Persistência dos dados (disco rígido, SSD, pendrive, CD/DVD)
  - Acesso aos dados pode ser não sequencial
  - Acesso concorrente aos dados (mais de um programa pode usar os dados ao mesmo tempo)

Tipos de Arquivos

- Basicamente, a linguagem C trabalha com dois tipos de arquivos: de texto e binários

- Arquivo texto

  - armazena caracteres que podem ser mostrados diretamente na tela ou modificados por um editor de textos simples como o Bloco de Notas.

  - Os dados são gravados como caracteres de 8 bits. Ex.: Um número inteiro de 32 bits com 8 dígitos ocupará 64 bits no arquivo (8 bits por dígito)

Tipos de Arquivos

- Arquivo binário

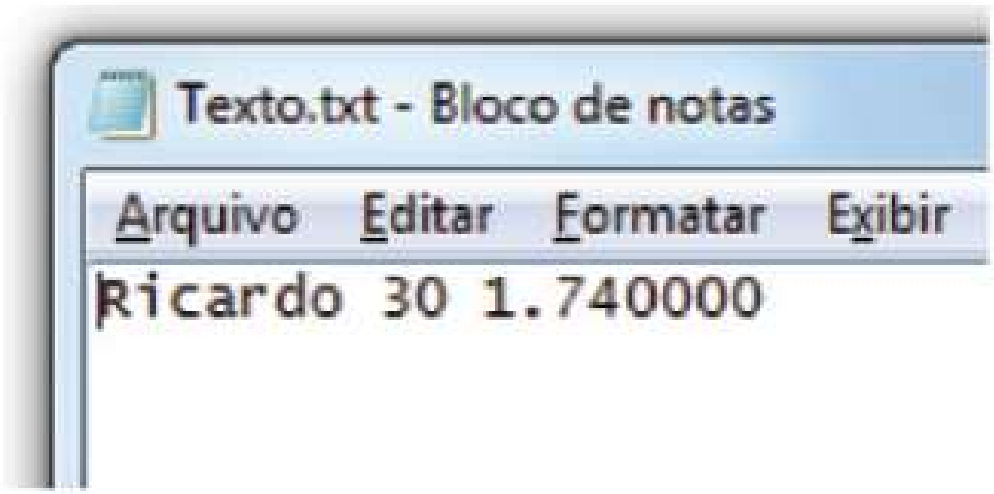
  - armazena uma sequência de bits que está sujeita as convenções dos programas que o gerou
    - Ex: arquivos executáveis, arquivos compactados, arquivos de registros, etc

  - os dados são gravados na forma binária (do mesmo modo que estão na memória)
    - Ex.: um número inteiro de 32 bits com 8 dígitos ocupará 32 bits no arquivo

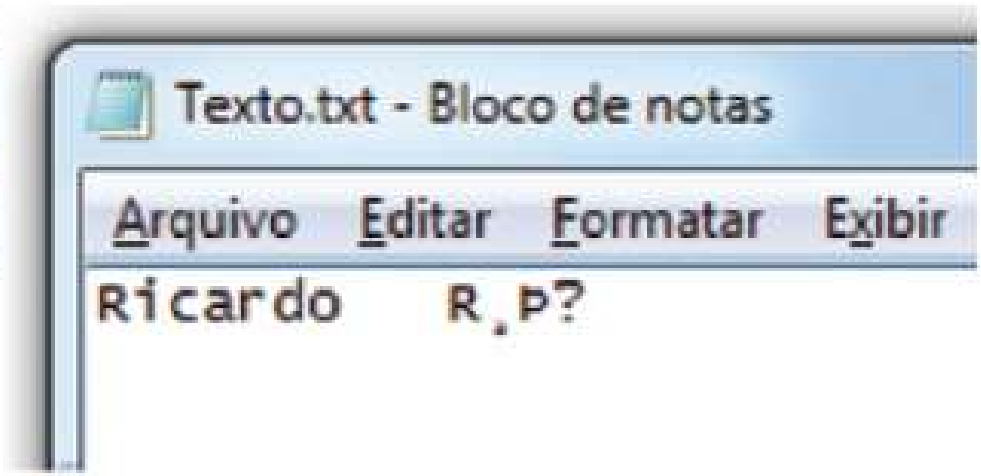
Tipos de Arquivos

- Ex: Os dois trechos de arquivo abaixo possuem os mesmo dados:

```
char nome[20] = "Ricardo";
int i = 30;
float a = 1.74;
```



Arquivo Texto



Arquivo Binário

MANIPULANDO ARQUIVOS EM C

- A linguagem C possui uma série de funções para manipulação de arquivos, cujos protótipos estão reunidos na biblioteca padrão de estrada e saída, **stdio.h**

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

Manipulando arquivos em C

- A linguagem C não possui funções que automaticamente leiam todas as informações de um arquivo
  - Suas funções se limitam a abrir/fechar e ler caracteres/bytes
  - É tarefa do programador criar a função que lerá um arquivo de uma maneira específica

MANIPULANDO ARQUIVOS EM C

- Todas as funções de manipulação de arquivos trabalham com o conceito de "ponteiro de arquivo". Podemos declarar um ponteiro de arquivo da seguinte maneira:

```
FILE *p;
```

- **p** é o ponteiro para arquivos que nos permitirá manipular arquivos no C.

ABRINDO UM ARQUIVO

- Para a abertura de um arquivo, usa-se a função **fopen**

```
FILE *fopen(char *nome_arquivo,char *modo);
```

- O parâmetro **nome_arquivo** determina qual arquivo deverá ser aberto, sendo que o mesmo deve ser válido no sistema operacional que estiver sendo utilizado

ABRINDO UM ARQUIVO

- No parâmetro **nome_arquivo** pode-se trabalhar com caminhos absolutos ou relativos
  - **Caminho absoluto**: descrição de um caminho desde o diretório raiz.
    - **C:\\Projetos\\dados.txt**
  - **Caminho relativo**: descrição de um caminho desde o diretório corrente (onde o programa está salvo)
    - **arq.txt**
    - **..\\dados.txt**

```
FILE *fopen(char *nome_arquivo,char *modo);
```

Abrindo um arquivo

- O modo de abertura determina que tipo de uso será feito do arquivo

- A tabela a seguir mostra os modo válidos de abertura de um arquivo

ABRINDO UM ARQUIVO

Modo	Arquivo	Função
"r"	Texto	Leitura. Arquivo deve existir.
"r+"	Texto	Leitura/Escrita. O arquivo deve existir e pode ser modificado.
"w"	Texto	Escrita. Cria arquivo se não houver. Apaga o anterior se ele existir.
"w+"	Texto	Leitura/Escrita. Cria arquivo se não houver. Apaga o anterior se ele existir.
"a"	Texto	Escrita. Os dados serão adicionados no fim do arquivo ("append"). Operações de reposicionamento (fseek, fsetpos, rewind) são ignoradas.
"a+"	Texto	Leitura/Escrita. Os dados serão adicionados no fim do arquivo ("append"). Operações de reposicionamento (fseek, fsetpos, rewind) afetam as próximas operaçõesde leitura, mas escritas são feitas no final do arquivo.
"rb"	Binário	Mesmo comportamento dos respectivos modos para texto
"r+b"	Binário	
"wb"	Binário	
"w+b"	Binário	
"ab"	Binário	
"a+b"	Binário	

ABRINDO UM ARQUIVO

- Um arquivo binário pode ser aberto para escrita utilizando a seguinte sequência:
  - Quando o resultado da função **fopen** é **NULL**, ou seja, a condição **fp==NULL,** a função não teve sucesso na abertura do arquivo (erro). Se **fp!=NULL** então **fopen** executado com sucesso

```
int main(){
    FILE *fp;
    fp = fopen("exemplo.bin","wb");
    if(fp == NULL)
        printf("Erro na abertura do arquivo.\n");

    fclose(fp);

    return 0;
}
```

Erro ao abrir um arquivo

- Caso o arquivo não tenha sido aberto com sucesso
  - Provavelmente o programa não poderá continuar a executar
  - Nesse caso, utilizamos a função **exit(),** presente na biblioteca **stdlib.h**, para fechar o programa

```
void exit(int codigo_de_retorno);
```

Erro ao abrir um arquivo

- A função **exit()** pode ser chamada de qualquer ponto no programa e faz com que o programa termine e retorne, para o sistema operacional, o **código_de_retorno**

- A convenção mais usada é que um programa retorne

  - **zero** no caso de um término normal

  - um número **diferente de zero**, no caso de ter ocorrido um problema

Erro ao abrir um arquivo

- Exemplo

```
int main(){
    FILE *fp;
    fp = fopen("exemplo.bin","wb");
    if(fp == NULL){
        printf("Erro na abertura do arquivo\n");
        system("pause");
        exit(1);
    }
    fclose(fp);

    return 0;
}
```

Posição do arquivo

- Ao se trabalhar com arquivos, existe uma posição onde estamos dentro do arquivo. É nessa posição onde será lido ou escrito o próximo caractere

  - Quando utilizando o acesso sequencial, raramente é necessário modificar essa posição
    - Isso por que, quando lemos um caractere, a posição no arquivo é automaticamente atualizada

  - Leitura e escrita em arquivos são parecidos com escrever em uma ***máquina de escrever***

FECHANDO UM ARQUIVO

- Sempre que terminamos de usar um arquivo que abrimos, devemos fechá-lo. Para isso usa-se a função **fclose()**

- O ponteiro **fp** passado à função **fclose()** determina o arquivo a ser fechado. A função retorna **zero** no caso de sucesso.

```
int fclose(FILE *fp);
```

FECHANDO UM ARQUIVO

- Por que devemos fechar o arquivo?

  - Ao fechar um arquivo, todo caractere que tenha permanecido no "buffer" é gravado

  - O "buffer“ é uma região de memória que armazena temporariamente os caracteres a serem gravados em disco imediatamente. Apenas quando o "buffer" está cheio é que seu conteúdo é escrito no disco

FECHANDO UM ARQUIVO

- Por que utilizar um “buffer”? Eficiência!
  - Para ler e escrever arquivos no disco temos que posicionar a cabeça de gravação em um ponto específico do disco
  - Se tivéssemos que fazer isso para cada caractere lido/escrito, a leitura/escrita de um arquivo seria uma operação muita lenta
  - Assim a gravação só é realizada quando há um volume razoável de informações a serem gravadas ou quando o arquivo for fechado
- A função **exit()** fecha todos os arquivos que um programa tiver aberto

Escrita/Leitura em Arquivos

- Uma vez aberto um arquivo, podemos ler ou escrever nele

- Para tanto, a linguagem C conta com uma série de funções de leitura/escrita que variam de funcionalidade para atender as diversas aplicações

ESCRITA/LEITURA DE CARACTERES

- A função mais básica de entrada de dados é a função **fputc** (*put character*)

```
int fputc (int ch, FILE *fp);
```

- Cada invocação dessa função grava um único caractere **ch** no arquivo especificado por **fp**

Escrita/Leitura de Caracteres

- Exemplo da função **fputc**

```c
int main(){
    FILE *arq;
    char string[100];
    int i;
    arq = fopen("arquivo.txt","w");
    if(arq == NULL){
        printf("Erro na abertura do arquivo");
        system("pause");
        exit(1);
    }
    printf("Entre com a string a ser gravada no arquivo:");
    gets(string);
    //Grava a string, caractere a caractere
    for(i = 0; i < strlen(string); i++)
        fputc(string[i], arq);
    fclose(arq);

    return 0;
}
```

ESCRITA/LEITURA DE CARACTERES

- A função **fputc** também pode ser utilizada para escrever um caractere na tela.
  - Nesse caso, é necessário mudar a variável que aponta para o local onde será gravado o caractere:
  - Por exemplo, **fputc ('*', stdout)** exibe um * na tela do monitor (dispositivo de saída padrão).

```
int main(){
    fputc ('*',stdout);

    return 0;
}
```

Escrita/Leitura de Caracteres

- Da mesma maneira que gravamos um único caractere no arquivo, também podemos ler um único caractere

```
int fgetc(FILE *fp);
```

Escrita/Leitura de Caracteres

- Cada chamada da função **fgetc** lê um único caractere do arquivo especificado
  - Se **fp** aponta para um arquivo, então **fgetc(fp)** lê o caractere atual no arquivo e se posiciona para ler o próximo caractere do arquivo

```
char c;
c = fgetc(fp);
```

Escrita/Leitura de Caracteres

- Exemplo da função **fgetc**

```
int main(){
    FILE *arq;
    char c;
    arq = fopen("arquivo.txt","r");
    if(arq == NULL){
        printf("Erro na abertura do arquivo");
        system("pause");
        exit(1);
    }
    int i;
    for(i = 0; i < 5; i++){
        c = fgetc(arq);
        printf("%c",c);
    }
    fclose(arq);

    return 0;
}
```

Escrita/Leitura de Caracteres

- Similar ao que acontece com a função **fputc**, a função **fgetc** também pode ser utilizada para a leitura do teclado (dispositivo de entrada padrão):
  - **fgetc(stdin)** lê o próximo caractere digitado no teclado

```
int main(){
    char ch;
    ch = fgetc(stdin);

    printf("%c\n",ch);

    return 0;
}
```

Escrita/Leitura de Caracteres

- O que acontece quando **fgetc** tenta ler o próximo caractere de um arquivo que já acabou?
  - Precisamos que a função retorne algo indicando o arquivo acabou

- Porém, todos os 256 caracteres são "válidos"!

ESCRITA/LEITURA DE CARACTERES

- Para evitar esse tipo de situação, **fgetc** não devolve um **char** mas um **int:**

```
int fgetc(FILE *fp);
```

- O conjunto de valores do **char** está contido dentro do conjunto do **int**
  - Se o arquivo tiver acabado, **fgetc** devolve um valor **int** que não possa ser confundido com um **char**

ESCRITA/LEITURA DE CARACTERES

- Assim, se o arquivo não tiver mais caracteres, **fgetc** devolve o valor **–1**
- Mais exatamente, **fgetc** devolve a constante **EOF** (*end of file*), que está definida na biblioteca **stdio.h**. Em muitas arquitetura o valor de **EOF** é **–1**
  - recomenda-se usar a constante EOF para que o seu programa seja independente da arquitetura

```
char c;
c = fgetc(fp);
if (c == EOF)
   printf ("O arquivo terminou!\n");
```

Escrita/Leitura de Caracteres

- Exemplo de uso do **EOF**

```
int main(){
    FILE *arq;
    char c;
    arq = fopen("arquivo.txt","r");
    if(arq == NULL){
        printf("Erro na abertura do arquivo");
        system("pause");
        exit(1);
    }
    while((c = fgetc(arq)) != EOF)
        printf("%c",c);

    fclose(arq);

    return 0;
}
```

FIM DO ARQUIVO

- Como visto, **EOF** ("End of file") indica o fim de um arquivo

- No entanto, podemos também utilizar a função **feof** para verificar se um arquivo chegou ao fim, cujo protótipo é

```
int feof(FILE *fp);
```

- Cuidado! É muito comum fazer mau uso dessa função!

FIM DO ARQUIVO

- Um mau uso muito comum da função **feof()** é usá-la para terminar um loop
  - Mas por que isso é um mau uso??

```
int main{
    int i, n;
    FILE *F = fopen("teste.txt","r");
    if(arq == NULL) {
        printf("Erro na abertura\n");
        system("pause");
        exit(1);
    }
    while(!feof(F)){
        fscanf(F,"%d",&n);
        printf("%d\n",n);
    }
    fclose(F);

    return 0;
}
```

FIM DO ARQUIVO

- Vamos ver a descrição da função **feof()**
  - A função **feof()** testa o indicador de fim de arquivo para o fluxo apontado por **fp**
  - A função retona um valor inteiro **diferente de zero** se, e somente se, o **indicador de fim de arquivo** está marcado para **fp**

- Ou seja, a função testa o **indicador de fim de arquivo**, não o próprio **arquivo**

```
int feof(FILE *fp);
```

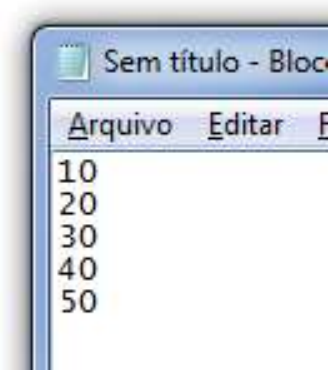
FIM DO ARQUIVO

- Isso significa que outra função é responsável por alterar o indicador para indicar que o EOF foi alcançado
  - A maioria das funções de leitura irá alterar o indicador após ler todos os dados, e então realizar uma nova leitura resultando em nenhum dado, apenas o **EOF**

- Como resolver isso
  - devemos evitar o uso da função **feof()** para testar um loop e usá-la para testar se uma leitura alterou o **indicador de fim de arquivo**

FIM DO ARQUIVO

- Para entender esse problema do mau uso da funções **feof()**, considere que queiramos ler todos os números contidos em um arquivo texto como o mostrado abaixo



FIM DO ARQUIVO

Mau uso da função feof()

```
int main{
    int i, n;
    FILE *F = fopen("teste.txt","r");
    if(arq == NULL) {
        printf("Erro na abertura\n");
        system("pause");
        exit(1);
    }
    while(!feof(F)){
        fscanf(F,"%d",&n);
        printf("%d\n",n);
    }
    fclose(F);

    return 0;
}

Saída: 10 20 30 40 50 50
```

Bom uso da função feof()

```
int main{
    int i, n;
    FILE *F = fopen("teste.txt","r");
    if(arq == NULL) {
        printf("Erro na abertura\n");
        system("pause");
        exit(1);
    }
    while(1){
        fscanf(F,"%d",&n);
        if(feof(F))
            break;
        printf("%d ",n);
    }
    fclose(F);

    return 0;
}

Saída: 10 20 30 40 50
```

ARQUIVOS PRÉ-DEFINIDOS

- Como visto anteriormente, os ponteiros **stdin** e **stdout** podem ser utilizados para acessar os dispositivo de entrada (geralmente o teclado) e saída (geralmente o vídeo) padrão

- Na verdade, no início da execução de um programa, o sistema automaticamente abre alguns arquivos pré-definidos, entre eles **stdin** e **stdout**

Arquivos pré-definidos

- Alguns arquivos pré-definidos
  - **stdin**
    - dispositivo de entrada padrão (geralmente o teclado)
  - **stdout**
    - dispositivo de saída padrão (geralmente o vídeo)
  - **stderr**
    - dispositivo de saída de erro padrão (geralmente o vídeo)
  - **stdaux**
    - dispositivo de saída auxiliar (em muitos sistemas, associado à porta serial)
  - **stdprn**
    - dispositivo de impressão padrão (em muitos sistemas, associado à porta paralela)

Escrita/Leitura de Strings

- Até o momento, apenas caracteres isolados puderam ser escritos em um arquivo

- Porém, existem funções na linguagem C que permitem ler/escrever uma sequência de caracteres, isto é, uma string
  - **fputs()**
  - **fgets()**

ESCRITA/LEITURA DE STRINGS

- Basicamente, para se escrever uma string em um arquivo usamos a função **fputs:**

```
int fputs(char *str, FILE *fp);
```

- Esta função recebe como parâmetro um array de caracteres (string) e um ponteiro para o arquivo no qual queremos escrever

Escrita/Leitura de Strings

- Retorno da função
  - Se o texto for escrito com sucesso um valor inteiro diferente de zero é retornado
  - Se houver erro na escrita, o valor EOF é retornado

- Como a função **fputc**, **fputs** também pode ser utilizada para escrever uma string na tela:

```
int main(){
    char texto[30] = "Hello World\n";
    fputs(texto, stdout);

    return 0;
}
```

Escrita/Leitura de Strings

- Exemplo da função **fputs:**

```
int main(){
    char str[20] = "Hello World!";
    int result;
    FILE *arq;
    arq = fopen("ArqGrav.txt","w");
    if(arq == NULL) {
        printf("Problemas na CRIACAO do arquivo\n");
        system("pause");
        exit(1);
    }
    result = fputs(str,arq);
    if(result == EOF)
        printf("Erro na Gravacao\n");
    fclose(arq);

    return 0;
}
```

ESCRITA/LEITURA DE STRINGS

- Da mesma maneira que gravamos uma cadeia de caracteres no arquivo, a sua leitura também é possível

- Para se ler uma string de um arquivo podemos usar a função **fgets()** cujo protótipo é:

```
char *fgets(char *str, int tamanho,FILE *fp);
```

ESCRITA/LEITURA DE STRINGS

- A função **fgets** recebe 3 parâmetros
  - **str**: aonde a lida será armazenada, **str**
  - **tamanho** :o número máximo de caracteres a serem lidos
  - **fp**: ponteiro que está associado ao arquivo de onde a string será lida

- E retorna
  - NULL em caso de erro ou fim do arquivo
  - O ponteiro para o primeiro caractere recuperado em **str**

```
char *fgets(char *str, int tamanho,FILE *fp);
```

Escrita/Leitura de Strings

- Funcionamento da função **fgets**
  - A função lê a string até que um caractere de nova linha seja lido ou *tamanho-1* caracteres tenham sido lidos
  - Se o caractere de nova linha ('\n') for lido, ele fará parte da string, o que não acontecia com **gets**
  - A string resultante sempre terminará com '\0' (por isto somente *tamanho-1* caracteres, no máximo, serão lidos)
  - Se ocorrer algum erro, a função devolverá um ponteiro nulo em **str**

ESCRITA/LEITURA DE STRINGS

- A função **fgets** é semelhante à função **gets**, porém, com as seguintes vantagens:
  - Pode fazer a leitura a partir de um arquivo de dados e incluir o caractere de nova linha **"\n"** na string
  - Específica o tamanho máximo da string de entrada
    - Isso evita estouro no buffer

Escrita/Leitura de Strings

- Exemplo da função **fgets**

```
int main(){
    char str[20];
    char *result;
    FILE *arq;
    arq = fopen("ArqGrav.txt","r");
    if(arq == NULL) {
        printf("Problemas na ABERTURA do arquivo\n");
        system("pause");
        exit(1);
    }
    result = fgets(str,13,arq);
    if(result == NULL)
        printf("Erro na leitura\n");
    else
        printf("%s",str);
    fclose(arq);

    return 0;
}
```

ESCRITA/LEITURA DE STRINGS

- Vale lembrar que o ponteiro **fp** pode ser substituído por **stdin**, para se fazer a leitura do teclado:

```
int main(){
    char nome[30];
    printf("Digite um nome: ");
    fgets(nome, 30, stdin);
    printf("O nome digitado foi: %s",nome);

    return 0;
}
```

Escrita/Leitura de bloco de dados

- Além da leitura/escrita de caracteres e sequências de caracteres, podemos ler/escrever blocos de dados

- Para tanto, temos duas funções

  - **fwrite()**

  - **fread()**

ESCRITA/LEITURA DE BLOCO DE DADOS

```
unsigned fwrite(void *buffer, int numero_de_bytes,
                int count, FILE *fp);
```

- **fwrite**: escrita de um bloco de dados da memória em um arquivo:
  - **buffer:** ponteiro para a região de memória na qual estão os dados
  - **numero_de_bytes:** tamanho de cada posição de memória a ser escrita
  - **count:** total de unidades de memória que devem ser escritas
  - **fp**: ponteiro associado ao arquivo onde os dados serão escritos

ESCRITA/LEITURA DE BLOCO DE DADOS

- Note que temos dois valores numéricos
  - **`numero_de_bytes`**
  - **`count`**
- Isto significa que o número total de bytes escritos é:
  - **`numero_de_bytes * count`**

- Como retorno, temos o número de unidades efetivamente escritas
  - Este número pode ser menor que **`count`** quando ocorrer algum erro

DADOS

- Exemplo da função **fwrite**

```c
int main(){
    FILE *arq;
    arq = fopen("ArqGrav.txt","wb");

    char str[20] = "Hello World!";
    float x = 5;
    int v[5] = {1,2,3,4,5};
    //grava a string toda no arquivo
    fwrite(str,sizeof(char),strlen(str),arq);
    //grava apenas os 5 primeiros caracteres da string
    fwrite(str,sizeof(char),5,arq);
    //grava o valor de x no arquivo
    fwrite(&x,sizeof(float),1,arq);
    //grava todo o array no arquivo (5 posições)
    fwrite(v,sizeof(int),5,arq);
    //grava apenas as 2 primeiras posições do array
    fwrite(v,sizeof(int),2,arq);
    fclose(arq);

    return 0;
}
```

Escrita/Leitura de bloco de dados

- A função **fread** é responsável pela leitura de um bloco de dados de um arquivo

- Seu protótipo é:

```
unsigned fread(void *buffer, int numero_de_bytes,
               int count, FILE *fp);
```

- A função **fread** funciona de modo análogo ao **fwrite**

- Como na função **fwrite, fread** retorna o número de itens lidos. Este valor será igual a **count** a menos que ocorra algum erro

DADOS

- Exemplo da função **fread**

```
char str1[20],str2[20];
float x;
int i,v1[5],v2[2];
//lê a string toda do arquivo
fread(str1,sizeof(char),12,arq);
str1[12] = '\0';
printf("%s\n",str1);
//lê apenas os 5 primeiros caracteres da string
fread(str2,sizeof(char),5,arq);
str2[5] = '\0';
printf("%s\n",str2);
//lê o valor de x do arquivo
fread(&x,sizeof(float),1,arq);
printf("%f\n",x);
//lê todo o array do arquivo (5 posições)
fread(v1,sizeof(int),5,arq);
for(i = 0; i < 5; i++)
    printf("v1[%d] = %d\n",i,v1[i]);
//lê apenas as 2 primeiras posições do array
fread(v2,sizeof(int),2,arq);
for(i = 0; i < 2; i++)
    printf("v2[%d] = %d\n",i,v2[i]);
```

ESCRITA/LEITURA DE BLOCO DE DADOS

- Quando o arquivo for aberto para dados binários, **fwrite** e **fread** podem manipular qualquer tipo de dado

  - int
  - float
  - double
  - **arrays [ ]**
  - **struct**
  - **arrays de struct [ ]**
  - etc

ESCRITA/LEITURA DE BLOCO DE DADOS

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

// tipos enumerados são conjuntos de identificadores.
// são implicitamente convertidos para números inteiros iniciando em 1
enum curso_t { mecatronica, mecanica, civil, computacao, eletrica, eletronica };

struct aluno { char matricula[10]; char nome[20]; enum curso_t curso; };

typedef struct aluno Aluno;

int main() {
    int x = sizeof(Aluno); printf(" %d \n\n",x);

    Aluno a[5];

    strcpy(a[0].matricula,"2018EMT001");
    strcpy(a[0].nome,"Edson Arantes");
    a[0].curso = mecatronica;

    strcpy(a[1].matricula,"2018CMP002");
    strcpy(a[1].nome,"Joao Doe");
    a[1].curso = computacao;

    FILE *f = fopen("alunos.txt","wb");
    fwrite(&a[0], 1, sizeof(Aluno), f); // escreve aluno "Edson"
    fwrite(&a[1], 1, sizeof(Aluno), f); // escreve aluno "Joao"
    fwrite(&a[0], 2, sizeof(Aluno), f); // escreve alunos "Edson" e "Joao"
    fclose(f);

    return 0;
}
```

Escrita/Leitura por fluxo padrão

- As funções de fluxos padrão permitem ao programador ler e escrever em arquivos da maneira padrão com a qual o já líamos e escrevíamos na tela

- As funções **fprintf** e **fscanf** funcionam de maneiras semelhantes a **printf** e **scanf**, respectivamente

- A diferença é que elas direcionam os dados para arquivos

Escrita/Leitura por fluxo padrão

- Ex: **fprintf**

```
printf("Total = %d",x);//escreve na tela
fprintf(fp,"Total = %d",x);//grava no arquivo fp
```

- Ex: **fscanf**

```
scanf("%d",&x);//lê do teclado
fscanf(fp,"%d",&x);//lê do arquivo fp
```

ESCRITA/LEITURA POR FLUXO PADRÃO

- Atenção

  - Embora **fprintf** e **fscanf** sejam mais fáceis de ler/escrever dados em arquivos, nem sempre elas são as escolhas mais apropriadas

  - Como os dados são escritos em ASCII e formatados como apareceriam em tela, se este for o objetivo, são as funções apropriadas
    - note que há um processamento extra se comparadas com as funções fwrite( ) e fread( )

  - Se a intenção é velocidade ou tamanho do arquivo, deve-se utilizar as funções **fread** e **fwrite**

Escrita/Leitura por fluxo padrão

- Exemplo da funções **fprintf**

```
int main(){
    FILE *arq;
    char nome[20] = "Ricardo";
    int I = 30;
    float a = 1.74;
    int result;
    arq = fopen("ArqGrav.txt","w");
    if(arq == NULL) {
        printf("Problemas na ABERTURA do arquivo\n");
        system("pause");
        exit(1);
    }
    fprintf(arq,"Nome: %s\n",nome);
    fprintf(arq,"Idade: %d\n",i);
    fprintf(arq,"Altura: %f\n",a);
    fclose(arq);

    return 0;
}
```

ESCRITA/LEITURA POR FLUXO PADRÃO

- Exemplo da função **fscanf**

```c
int main(){
    FILE *arq;
    char texto[20], nome[20];
    int i;
    float a;
    int result;
    arq = fopen("ArqGrav.txt","r");
    if(arq == NULL) {
        printf("Problemas na ABERTURA do arquivo\n");
        system("pause");
        exit(1);
    }
    fscanf(arq,"%s%s",texto,nome);
    printf("%s %s\n",texto,nome);
    fscanf(arq,"%s %d",texto,&i);
    printf("%s %d\n",texto,i);
    fscanf(arq,"%s%f",texto,&a);
    printf("%s %f\n",texto,a);
    fclose(arq);

    return 0;
}
```

MOVENDO-SE PELO ARQUIVO

- De modo geral, o acesso a um arquivo é sequencial. Porém, é possível fazer buscas e acessos aleatórios em arquivos

- Para isso, existe a função **fseek**

```
int fseek(FILE *fp,long numbytes,int origem);
```

- Basicamente, esta função move a posição corrente de leitura ou escrita no arquivo em tantos bytes, a partir de um ponto especificado

MOVENDO-SE PELO ARQUIVO

- A função **fseek** recebe 3 parâmetros
  - **fp**: o ponteiro para o arquivo
  - **numbytes**: é o total de bytes a partir de **origem** a ser pulado
  - **origem**: determina a partir de onde os **numbytes** de movimentação serão contados
- A função retorna o valor 0 quando bem sucedida

```
int fseek(FILE *fp,long numbytes,int origem);
```

MOVENDO-SE PELO ARQUIVO

- Os valores possíveis para **origem** são definidos por macros em **stdio.h** e são:

Nome	Valor	Significado
SEEK_SET	0	Início do arquivo
SEEK_CUR	1	Ponto corrente do arquivo
SEEK_END	2	Fim do arquivo

- Portanto, para mover **numbytes** a partir
  - do início do arquivo, **origem** deve ser SEEK_SET
  - da posição atual, **origem** deve ser SEEK_CUR
  - do final do arquivo, **origem** deve ser SEEK_END

- **numbytes** pode ser negativo quando usado com SEEK_CUR e SEEK_END

MOVENDO-SE PELO ARQUIVO

- Exemplo da função **fseek**

```
struct cadastro{ char nome[20], rua[20]; int idade;  };
int main(){
     FILE *f = fopen("arquivo.txt","wb");

     struct cadastro c,cad[4] = {"Ricardo","Rua 1",31,
                                 "Carlos","Rua 2",28,
                                 "Ana","Rua 3",45,
                                 "Bianca","Rua 4",32};
     fwrite(cad,sizeof(struct cadastro),4,f);
     fclose(f);

     f = fopen("arquivo.txt","rb");
     fseek(f,2*sizeof(struct cadastro),SEEK_SET);
     fread(&c,sizeof(struct cadastro),1,f);
     printf("%s\n%s\n%d\n",c.nome,c.rua,c.idade);
     fclose(f);

     return 0;
}
```

MOVENDO-SE PELO ARQUIVO

- Outra opção de movimentação pelo arquivo é simplesmente retornar para o seu início

- Para tanto, usa-se a função **rewind**:

```
void rewind(FILE *fp);
```

FTELL

www.cplusplus.com/reference/cstdio/ftell/

function

ftell

<cstdio>

```
long int ftell ( FILE * stream );
```

Get current position in stream

Returns the current value of the position indicator of the *stream*.

For binary streams, this is the number of bytes from the beginning of the file.

For text streams, the numerical value may not be meaningful but can still be used to restore the position to the same position later using fseek (if there are characters put back using ungetc still pending of being read, the behavior is undefined).

Parameters

`stream`

Pointer to a FILE object that identifies the stream.

Return Value

On success, the current value of the position indicator is returned.
On failure, `-1L` is returned, and errno is set to a system-specific positive value.

FTELL

Example

```c
/* ftell example : getting size of a file */
#include <stdio.h>

int main ()
{
  FILE * pFile;
  long size;

  pFile = fopen ("myfile.txt","rb");
  if (pFile==NULL) perror ("Error opening file");
  else
  {
    fseek (pFile, 0, SEEK_END);
    size=ftell (pFile);
    fclose (pFile);
    printf ("Size of myfile.txt: %ld bytes.\n",size);
  }
  return 0;
}
```

FFLUSH

www.cplusplus.com/reference/cstdio/fflush/

function

fflush

<cstdio>

```
int fflush ( FILE * stream );
```

Flush stream

If the given *stream* was open for writing (or if it was open for updating and the last i/o operation was an output operation) any unwritten data in its output buffer is written to the file.

If *stream* is a null pointer, all such streams are flushed.

In all other cases, the behavior depends on the specific library implementation. In some implementations, flushing a stream open for reading causes its input buffer to be cleared (but this is not portable expected behavior).

The stream remains open after this call.

When a file is closed, either because of a call to fclose or because the program terminates, all the buffers associated with it are automatically flushed.

Parameters

`stream`

Pointer to a FILE object that specifies a buffered stream.

Return Value

A zero value indicates success.
If an error occurs, EOF is returned and the error indicator is set (see ferror).

APAGANDO UM ARQUIVO

- Além de permitir manipular arquivos, a linguagem C também permite apagá-lo do disco. Isso pode ser feito utilizando a função **remove**:

```
int remove(char *nome_do_arquivo);
```

- Diferente das funções vistas até aqui, esta função recebe o **caminho e nome** do arquivo a ser excluído, e não um ponteiro para FILE

- Como retorno temos um valor inteiro, o qual será igual a 0 se o arquivo for excluído com sucesso

Apagando um arquivo

- Exemplo da função **remove**

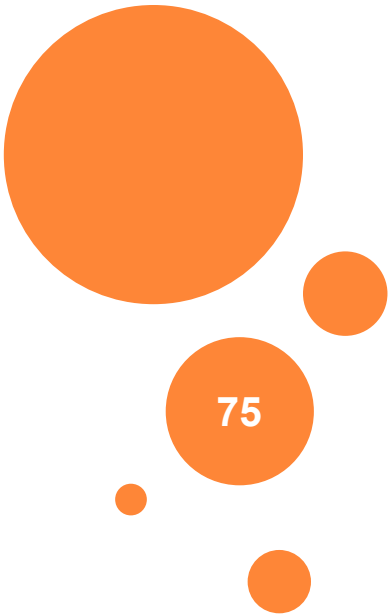
```
int main(){
    int status;
    status = remove("ArqGrav.txt");
    if(status != 0){
        printf("Erro na remocao do arquivo.\n");
        system("pause");
        exit(1);
    }else
        printf("Arquivo removido com sucesso.\n");

    return 0;
}
```

Material Complementar

- Vídeo Aulas
  - Aula 66: Arquivos pt.1 – Introdução
  - Aula 67: Arquivos pt.2 – Arquivos Texto e Binário
  - Aula 68: Arquivos pt.3 – Abrir e Fechar
  - Aula 69: Arquivos pt.4 – fputc
  - Aula 70: Arquivos pt.5 - fgetc
  - Aula 71: Arquivos pt.6 - Trabalhando com Arquivos
  - Aula 72: Arquivos pt.7 – EOF **(contém erros!)**
  - Aula 73: Arquivos pt.8 - fputs
  - Aula 74: Arquivos pt.9 - fgets
  - Aula 75: Arquivos pt.10 - fwrite
  - Aula 76: Arquivos pt.11 - fread
  - Aula 77: Arquivos pt.12 - fprintf
  - Aula 78: Arquivos pt.13 - fscanf
  - Aula 79: Arquivos pt.14 - fseek e rewind
  - Aula 90: Mau uso da função feof()

  - https://programacaodescomplicada.wordpress.com/indice/linguagem-c/



LINGUAGEM C: ARQUIVOS

Contém slides originais gentilmente disponibilizados pelo Prof. André R. Backes (UFU)