



LINGUAGEM DE PROGRAMAÇÃO

AULA 6



Prof. Sandro de Araújo



CONVERSA INICIAL

Esta aula tem como base os livros: *Lógica de programação e estruturas de dados*, *Fundamentos da Programação de Computadores* e *Treinamento em Linguagem C*. Em caso de dúvidas, ou para aprofundamento dos temas, consulte-os na biblioteca virtual.

A aula apresenta a seguinte estrutura de conteúdo:

1. Operações em memória - parte 1
2. Operações em memória - parte 2
3. Arquivos em C
4. Modos de abertura: read (r), write (w) e append (a)
5. Gravação e leitura de arquivos

O objetivo desta aula é apresentar os principais conceitos e aplicações de operações em memória, arquivos em C, modos de leituras de arquivos e gravação e leitura de arquivos para resolver problemas computacionais.

TEMA 1 – OPERAÇÕES EM MEMÓRIA: PARTE 1

As sub-rotinas de memória operam diretamente em áreas de memória, e a linguagem de programação C possui algumas funções para manipulação dessa sub-rotinas. Essas funções pertencem à biblioteca **string.h**. São elas:

- **memset** – usada para preenchimento de memória.
- **memcpy** – faz cópia de memória.
- **memmove** – também faz cópia de memória, só que de uma forma mais segura.
- **memcmp** – faz comparação de memória.

A seguir, veremos as funções **memset** e **memcpy**.

1.1 Função **memset()**

A função **memset()** preenche (**inicializa**) uma quantidade de memória (variável, constante, vetor, estrutura, entre outros) com um determinado valor de **byte**.

A **memset()** tem a seguinte sintaxe:

```
void * memset ( void * nPonteiro , int nValor , size_t nBytes );
```



Na qual:

- **nPonteiro** – refere-se a um ponteiro criado para a região de memória que será preenchida.
- **nValor** – refere-se ao valor usado para preencher a região de memória. O valor será convertido automaticamente para **unsigned char** (8 bits). O **unsigned char** não permite armazenar valores negativos e pode representar números em um intervalo que vai de 0 até 255.
- **nBytes** – refere-se ao número de **bytes** que serão preenchidos. Não é necessariamente o tamanho do *array*.

Retorno:

- Retorna uma cópia do ponteiro **nPonteiro**.
- Retorna **NULL** em caso de erro.

A Figura 1 apresenta um exemplo com a função `memset()`:

Figura 1 – Exemplo de um algoritmo com `memset()`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main() {
6
7      //array de 40 posições
8      char exMemset[40] = "Exemplo memset em C\n";
9
10     //função para "colocar" uma string na saída de dados
11     puts(exMemset);
12
13     //Substitui os seis primeiros caracteres
14     memset(exMemset, '*', 6);
15     puts(exMemset);
16
17     system("pause");
18     return 0;
19 }
```

Fonte: elaborado pelo autor.

A Figura 2 mostra a saída do algoritmo da Figura 1 após a sua execução:



Figura 2 – Saída do algoritmo

```
Exemplo memset em C
*****o memset em C
Pressione qualquer tecla para continuar. . . _
```

Fonte: elaborado pelo auto.

No próximo exemplo, apresentado na Figura 3, veremos outro algoritmo com a função `memset` usada para substituir números inteiros de um vetor com seis posições.

Figura 3 – Exemplo de um algoritmo com `memset()` e vetor

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  void imprime(int *vPont, int num);
5
6  int main() {
7
8      //Vetor com 6 posições com 4 Bytes em cada posição
9      int vPont[6] = {1, 2, 3, 4, 5, 6};
10     imprime(vPont, 6);
11
12     /*Preenche apenas os 12 primeiros Bytes
13     com o valor 0*/
14     memset(vPont, 0, 12);
15     imprime(vPont, 6);
16
17     //Preenche os 24 Bytes com o valor 0
18     memset(vPont, 0, 24);
19     imprime(vPont, 6);
20
21     system("pause");
22     return 0;
23 }
24
25 void imprime(int *vPont, int num){
26     int i;
27     for(i = 0; i < num; i++){
28         printf("%d. ", vPont[i]);
29     }
30     printf("\n\n");
31 }
32

```

Fonte: Elaborado pelo autor.

A Figura 4 mostra a saída do algoritmo da Figura 3 após a sua execução:

Figura 4 – Saída do algoritmo

```

1. 2. 3. 4. 5. 6.
0. 0. 0. 4. 5. 6.
0. 0. 0. 0. 0. 0.

Pressione qualquer tecla para continuar. . .

```

Fonte: Elaborado pelo autor.



Para o próximo algoritmo vamos preencher os espaços com o número 271, só que binário. Esse número é equivalente a: 0 0 0 0 0 0 0 1 0 0 0 0 1 1 1 1.

Conforme a conversão realizada na Tabela 1:

Tabela 1 – Conversão de decimal para binário

512	256	128	64	32	16	8	4	2	1
0	1	0	0	0	0	1	1	1	1

Fonte: Elaborado pelo autor.

Nesse caso a função `memset` vai usar apenas os 8 *bits*, representados em verde na Tabela 2, descartando todo o resto em azul.

Tabela 2 – *Byte* usado pela função `memset()`

0	1	0	0	0	0	1	1	1	1
---	---	---	---	---	---	---	---	---	---

Fonte: Elaborado pelo autor.

Então:

271 = 0 0 0 0 0 0 0 1 0 0 0 0 1 1 1 1

e

15 = 0 0 0 0 1 1 1 1

O `memset()` vai usar só 8 *bits*, que estão representados na Tabela 1. A cor verde, em binário, é equivalente a decimal 15, conforme apresentado no algoritmo da Figura 5:



Figura 5 – Exemplo de um algoritmo com memset() e vetor

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  void imprime(int *vPont, int num);
5
6  int main() {
7
8      //Vetor com 6 posições com 4 Bytes em cada posição
9      int vPont[6] = {1, 2, 3, 4, 5, 6};
10     imprime(vPont, 6);
11
12     memset(vPont, 271, 4);
13     imprime(vPont, 6);
14
15
16     system("pause");
17     return 0;
18 }
19
20 void imprime(int *vPont, int num){
21     int i;
22     for(i = 0; i < num; i++){
23         printf("%d. ", vPont[i]);
24     }
25     printf("\n\n");
26
27 }
```

Fonte: Elaborado pelo autor.

A Figura 6 mostra a saída do algoritmo da Figura 5 após a sua execução:

Figura 6 – Saída do algoritmo

```
1. 2. 3. 4. 5. 6.
3855. 2. 3. 4. 5. 6.
Pressione qualquer tecla para continuar. . . _
```

Fonte: Elaborado pelo autor.



Repare que, na Figura 6, tivemos como saída o número 3855. Isso aconteceu porque o *byte* 0 0 0 0 1 1 1 1 preencheu duas posições.

3855 = 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1

No próximo exemplo, na Figura 7, vamos preencher três posições do vetor. Analise o código e veja o resultado na Figura 8.

Figura 7 – Exemplo de um algoritmo com `memset()` e vetor

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  void imprime(int *vPont, int num);
5
6  int main() {
7
8      //Vetor com 6 posições com 4 Bytes em cada posição
9      int vPont[6] = {1, 2, 3, 4, 5, 6};
10     imprime(vPont, 6);
11
12     memset(vPont, 271, 4);
13     imprime(vPont, 6);
14
15
16     system("pause");
17     return 0;
18 }
19
20 void imprime(int *vPont, int num){
21     int i;
22     for(i = 0; i < num; i++){
23         printf("%d. ", vPont[i]);
24     }
25     printf("\n\n");
26
27 }
```

Fonte: Elaborado pelo autor.

A Figura 8 mostra a saída do algoritmo da Figura 7 após a sua execução:



Figura 8 – Saída do algoritmo

```
1. 2. 3. 4. 5. 6.
3855. 2. 3. 4. 5. 6.
Pressione qualquer tecla para continuar. . . _
```

Fonte: Elaborado pelo autor.

1.2 Função memcpy()

Essa função copia uma quantidade de *bytes* de uma área de memória para outra. Ambas as regiões de memória são tratadas com ***unsigned char***.

Sintaxe da função memcpy():

```
void* memcpy(void* pDestino, void* pOrigem, size_t num);
```

Na qual temos:

- **pDestino** – refere-se ao ponteiro para a região de memória que receberá os dados copiados.
- **pOrigem** – refere-se ao ponteiro para a região de memória de onde os dados serão copiados.
- **num** – refere-se ao número de *bytes* que serão copiados. Não é necessariamente o tamanho de um vetor.

Retorno:

- Retorna uma cópia do ponteiro pDestino;
- Retorna NULL em caso de erro.

A função memcpy() é a função mais rápida entre regiões de memória. Mas o endereço de memória não deve sobrepor-se; caso se sobreponha, então o memcpy() é indefinido – isto é, o memcpy() desconsidera sobreposição. Desse modo, essa função pode copiar do começo ao fim (ou o inverso), em blocos de vários *bytes* etc.

A Figura 9 apresenta um exemplo com a função memcpy():



Figura 9 – Exemplo de um algoritmo com memcpy()

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main ()
5  {
6      const char src[50] = "http://www.uninter.com";
7      char dest[50];
8      int tamanho;
9
10     printf("Antes da funcao memcpy = %s\n\n", dest);
11
12     //Descobrir o tamanho de uma string em C usando strlen();
13     tamanho = strlen(src);
14
15     printf("O tamanho da string %s e': %d\n\n", src, tamanho);
16
17     memcpy(dest, src, tamanho);
18     printf("depois da funcao memcpy = %s\n", dest);
19
20     return 0;
21 }
```

Fonte: Elaborado pelo autor.

A Figura 10 mostra a saída do algoritmo da Figura 9 após a sua execução:

Figura 10 – Saída do algoritmo

```
Antes da funcao memcpy =
O tamanho da string http://www.uninter.com e': 22
depois da funcao memcpy = http://www.uninter.com
-----
Process exited after 0.3292 seconds with return value 0
Pressione qualquer tecla para continuar. . .
```

Fonte: Elaborado pelo autor.

A Figura 11 apresenta um exemplo com a função memcpyt() e struct:



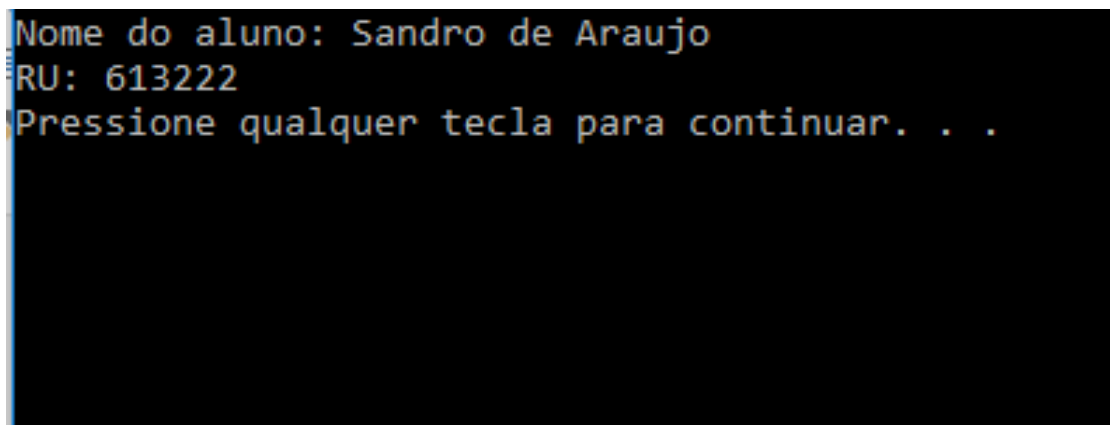
Figura 11 – Exemplo de um algoritmo com memcpy() e struct

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  struct aluno{
5      char nomeAluno[40];
6      int RUAaluno;
7  };
8
9  int main() {
10     struct aluno a1 = {"Sandro de Araujo", 613222};
11     struct aluno a2;
12
13     memcpy(&a2, &a1, sizeof(struct aluno));
14
15     printf("Nome do aluno: %s\nRU: %d\n", a2.nomeAluno, a2.RUAaluno);
16
17     system("pause");
18     return 0;
19 }
```

Fonte: Elaborado pelo autor.

A Figura 12 mostra a saída do algoritmo da Figura 11 após a sua execução:

Figura 12 – Saída do algoritmo



```
Nome do aluno: Sandro de Araujo
RU: 613222
Pressione qualquer tecla para continuar. . .
```

Fonte: Elaborado pelo autor.

TEMA 2 – OPERAÇÕES EM MEMÓRIA: PARTE 2

As funções para manipular a memória estão agrupadas nas categorias definidas na biblioteca **string.h**. Neste momento, veremos como fazer uma cópia



de memória de uma forma mais segura com a função `memmove()` e como fazer comparação de memória com a função `memcmp()`.

2.1 Função `memmove()`

Essa função copia uma quantidade de *bytes* de uma área de memória para outra. Ambas as regiões de memória são tratadas com `unsigned char`.

Sintaxe `memmove()`:

```
Void* memmove(void* pDestino, void* pOrigem, size_t num);
```

Na qual temos:

- **pDestino** – refere-se ao ponteiro para região de memória que receberá os dados copiados.
- **pOrigem** – refere-se ao ponteiro para a região de memória de onde os dados serão copiados.
- **num** – refere-se ao número de *bytes* que serão copiados, não necessariamente o tamanho de um vetor.

Retorno:

- Retorna uma cópia do ponteiro `pDestino`.
- Retorna `NULL` em caso de erro.

O funcionamento da função `memmove()` é igual ao da função `memcpy()`, só que mais lenta. Porém, a função `memmove()` é mais segura no caso da existência de duas regiões sobrepostas na memória, ou seja, duas áreas em comum na memória.

A `memcpy()` usa um vetor auxiliar para fazer a cópia.

Exemplo:

1. `char nome[20];`
2. `memcpy(&nome[0], &nome[4], 10);` //Comportamento inesperado no caso de sobreposição.
3. `Memmove(&nome[0], &nome[4], 10);` //Usa um vetor para tratar o problema de sobreposição.

A Figura 13 apresenta um exemplo com a função `memmove()`:



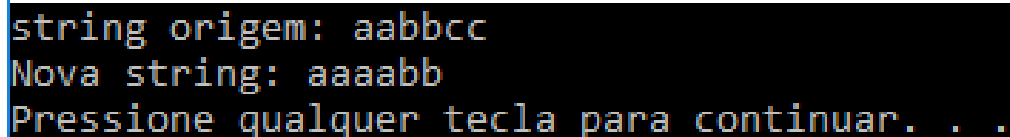
Figura 13 – Exemplo de um algoritmo com memmove()

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  char nom[7] = "aabbcc";
5
6  int main()
7  {
8      printf( "string origem: %s\n", nom );
9      memcpy( nom + 2, nom, 4 );
10     printf( "Nova string: %s\n", nom );
11
12     system("pause");
13     return 0;
14 }
```

Fonte: Elaborado pelo autor.

A Figura 14 mostra a saída do algoritmo da Figura 13 após a sua execução:

Figura 14 – Saída do algoritmo



```
string origem: aabbcc
Nova string: aaaabb
Pressione qualquer tecla para continuar. . .
```

Fonte: Elaborado pelo autor.

2.2 Função memcmp()

A função memcmp() é usada para saber se uma *string* é maior, menor ou igual a outra. Essa função compara as n primeiras posições de duas *strings*, ou seja, de 0 até n-1.

Essas *strings* são, na verdade, números inteiros, e sua representação está na tabela ASCII. Essa comparação é feita caractere por caractere.

Exemplo:



- $a = c$
- $a < c$
- $z > c$

Essa função compara diretamente os caracteres das *strings*. Se, em algum momento da comparação, algum caractere da *string* 1 for menor que o da *string* 2, a função para e retorna -1. Ademais, caso algum caractere da *string* 1 for maior que o da *string* 2, a função para e retorna 1. Caso o processo não retorne nem 1 ou -1, caracteriza *strings* idênticas. Ambas as regiões da memória são tratadas como *unsigned char*, e a comparação é feita na ordem lexicográfica, ou seja, ordem do dicionário ou ordem alfabética.

Sintaxe memcmp():

```
Int memcmp(void* pRegiao1, void* pRegiao2, size_t N);
```

Na qual temos:

- **pRegiao1** – refere-se ao ponteiro para uma região de memória.
- **pRegiao2** – refere-se ao ponteiro para uma região de memória.
- **N** – refere-se ao número de *bytes* que serão comparados, não necessariamente o tamanho de um vetor.

Retorno:

- Se o valor de retorno < 0 , então pRegiao1 menor que pRegiao2.
- Se o valor de retorno $== 0$, então blocos de memória são iguais.
- Se o valor de retorno > 0 , então pRegiao1 maior que pRegiao2.

A Figura 15 apresenta um exemplo com a função memcmp():



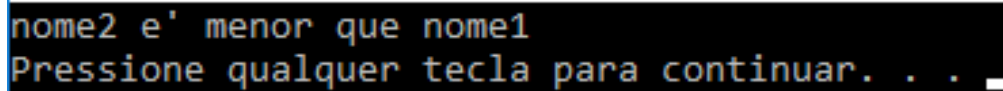
Figura 15 – Exemplo de um algoritmo com memcmp()

```
1  #include <stdio.h>
2  #include<stdlib.h>
3  #include <string.h>
4  int main ()
5  {
6      char nome1[15];
7      char nome2[15];
8      int ret;
9
10     memcpy(nome1, "abcdef", 6);
11     memcpy(nome2, "ABCDEF", 6);
12
13     ret = memcmp(nome1, nome2, 5);
14
15     if(ret > 0)
16     {
17         printf("nome2 e' menor que nome1\n");
18     }
19     else if(ret < 0)
20     {
21         printf("nome1 e' menor que nome2\n");
22     }
23     else
24     {
25         printf("nome1 e' igual a nome2\n");
26     }
27
28     system("pause");
29     return(0);
30 }
```

Fonte: Elaborado pelo autor.

A Figura 16 mostra a saída do algoritmo da Figura 15 após a sua execução:

Figura 16 – Saída do algoritmo



```
nome2 e' menor que nome1
Pressione qualquer tecla para continuar. . . _
```

Fonte: Elaborado pelo autor.

A Figura 17 apresenta um exemplo com a função `memcmp()` e `struct`:

Figura 17 – Exemplo de um algoritmo com `memcmp()` e `struct`

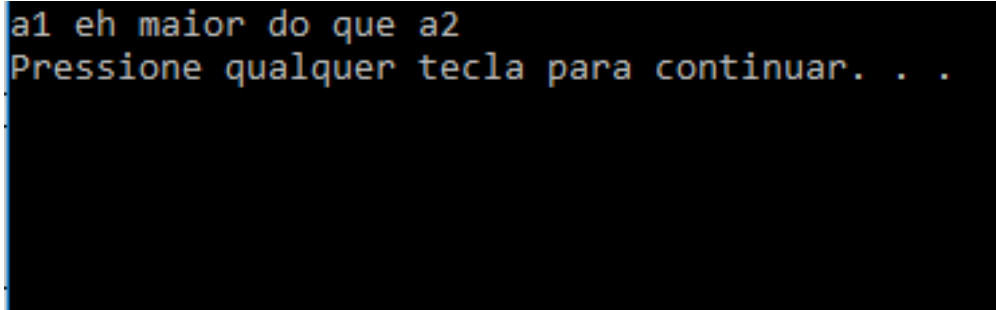
```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  struct aluno{
6      char nomeAluno[40];
7      int RUAaluno;
8  };
9
10 int main() {
11     struct aluno a1 = {"sandro", 65894};
12     struct aluno a2 = {"sandr0", 65894};
13
14     int retorno;
15
16     retorno = memcmp(&a1, &a2, sizeof(struct aluno));
17
18     if(retorno == 0){
19         printf("Campos iguais\n");
20     }
21     else
22     {
23         if(retorno > 0){
24             printf("a1 eh maior do que a2\n");
25         }
26         else{
27             printf("a2 eh maior do que a1\n");
28         }
29
30         system("pause");
31         return 0;
32     }
```

Fonte: Elaborado pelo autor.



A Figura 18 mostra a saída do algoritmo da Figura 14 após a sua execução:

Figura 18 – Saída do algoritmo



```
a1 eh maior do que a2
Pressione qualquer tecla para continuar. . .
```

Fonte: Elaborado pelo autor.

TEMA 3 – ARQUIVOS EM C

Arquivo em linguagem de programação C é uma coleção de *bytes* armazenados em um dispositivo secundário.

Exemplos:

- disco rígido;
- *pen drive*;
- cartão SSD;
- entre outros.

Quais as vantagens de se usar arquivos?

- Armazenamento durável;
- permitem armazenar uma grande quantidade de informação;
- acesso concorrente aos dados.

É importante ressaltar que a extensão do arquivo não define o seu tipo. O que define um arquivo é a maneira como os dados estão organizados por um programa para processar (ler e escrever) esse arquivo.

Para manipular arquivos na linguagem C usa-se a biblioteca ***stdio.h*** e um tipo especial de ponteiro. A função desse ponteiro é apontar a localização de um registro e controlar o fluxo de leitura e escrita dentro de um arquivo. A declaração de um ponteiro para arquivo em C segue a sintaxe a seguir:

```
FILE *nomePonteiro;
```



Lembrando que *FILE* deve ser escrito em letras maiúsculas.

A linguagem C trabalha com apenas dois tipos de arquivos:

- **Arquivos texto** – podem ser editados no bloco de notas.
- **Arquivos binários** – não podem ser editados no bloco de notas.

Um arquivo texto tem seus dados gravados exatamente como seriam impressos na tela. Esses dados são gravados como caracteres de 8 *bits* utilizando a tabela ASCII. Para que isso ocorra, existe uma etapa de **conversão** dos dados. Se um dado inteiro tem 4 *bits*, quando convertido ele terá 8 *bits*, que é o padrão da tabela ASCII para um caractere. Consequentemente, os arquivos são maiores, tendo leitura e escrita mais lentos.

Um arquivo binário tem seus dados gravados exatamente como estão organizados na memória do computador. Não existe etapa de **conversão** dos dados para esse tipo de arquivo, e o conteúdo da memória será copiado diretamente para o arquivo. Consequentemente, os arquivos são menores, tendo leitura e escritas mais rápidas.

Em C, o arquivo é manipulado por meio de um ponteiro especial para o arquivo.

3.1 Abertura e fechamento de arquivos

A manipulação de um arquivo na linguagem C se dá em três etapas:

1. abrir o arquivo;
2. ler e/ou gravar os dados;
3. fechar o arquivo.

Para trabalhar com um arquivo, a primeira operação necessária é abrir tal arquivo. Para se realizar essa operação, usamos a seguinte sintaxe de abertura de arquivo:

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

A função `fopen` recebe como parâmetros o nome do arquivo a ser aberto e o tipo de abertura a ser realizado. Vejamos um exemplo, em que a função `fopen` vai abrir um arquivo `txt` em modo de escrita:

1. `FILE *arq;`



2. `arq = fopen("arquivo.txt", "w");`

Há duas formas para especificar o caminho do arquivo:

1. Caminho absoluto – o endereço completo é escrito. Exemplo:

`f = fopen("C:\\Users\\Casa\\Documents\\uninter.txt", "w");`

Para representar a barra "\" em uma string, usamos duas barras: \\

2. Caminho relativo – relativo ao diretório do programa. Exemplo:

`f = fopen("uninter.txt", "w");`

`f = fopen("../Novo\\uninter2.txt", "w");`

A Figura 19 apresenta um exemplo com a função `fopen()`:

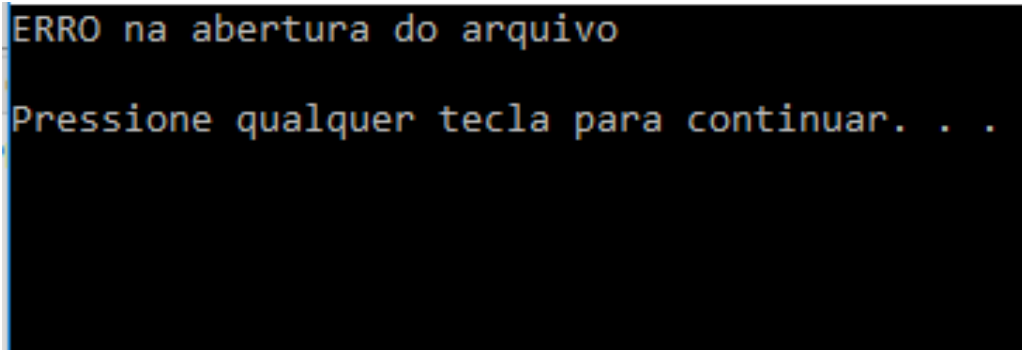
Figura 19 – Exemplo de um algoritmo com `fopen()`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5
6      FILE *arq;
7      arq = fopen("uninter.txt", "r");
8      if((arq=fopen("uninter.txt", "r"))==NULL){
9          printf("ERRO na abertura do arquivo\n\n");
10         system("pause");
11         exit(1);
12     }
13
14
15     system("pause");
16     return 0;
17 }
```

Fonte: Elaborado pelo autor.

A Figura 20 mostra a saída do algoritmo da Figura 19 após a sua execução:

Figura 20 – Saída do algoritmo



```
ERRO na abertura do arquivo
Pressione qualquer tecla para continuar. . .
```

Fonte: Elaborado pelo autor.

Caso o arquivo não exista ou não tenha permissão de acesso, a função `fopen` também irá retornar `NULL` para o ponteiro. Quando esse erro ocorre, o ponteiro irá apontar para `NULL`, sendo essa prática (checar se o ponteiro aponta para `NULL`) muito importante para o tratamento de erros na abertura de arquivos em C.

Para saber o final do arquivo, a linguagem C procura um sinal — uma constante conhecida por EOF —, que sinaliza o fim do arquivo.

Se o *byte* lido pelo algoritmo representa o EOF, a função `fclose()` "fecha" a abertura do arquivo. Ou seja, libera a memória associado ao ponteiro do `FILE*`.

Assim como em ponteiros, quando usamos a função `free()` para liberar memória alocada, fechar os arquivos que não estão mais sendo usados é uma boa prática de programação.

TEMA 4 – MODOS DE ABERTURA: READ (R), WRITE (W) E APPEND (A)

O modo de acesso é uma *string* que contém uma sequência de caracteres que informam se o arquivo será aberto para escrita ou leitura. Depois que abrir o arquivo, podemos executar os tipos de ação previstos pelo modo de acesso. Assim, não será possível ler de um arquivo que foi aberto somente para escrita. Os modos de acesso usados na linguagem C são descritos a seguir.

- **read(r) – Leitura de arquivo**

1. `r` – para ler um arquivo, usamos o modo *read*.

Exemplo: `FILE *arquivo = fopen("uninter.txt", "r");`

O arquivo será aberto unicamente para leitura.



2. **r+** – na adição do sinal "+" no "r", iremos abrir o arquivo tanto para leitura como para escrita; caso ele não exista, o arquivo será criado. Se o arquivo existir, terá o seu conteúdo apagado e substituído pelo novo.

Exemplo: `FILE *arquivo = fopen("uninter.txt", "r+");`

3. **rb** – abre o arquivo em modo binário para leitura.

Exemplo: `FILE *arquivo = fopen("uninter.txt", "rb");`

- **write(w) – Escrita em arquivo**

1. **w** – para abrir um arquivo no modo de escrita, usamos a letra **w**. Esse modo automaticamente cria o arquivo ou substitui seu conteúdo anterior.

Exemplo: `FILE *arquivo = fopen("uninter.txt", "w");`

2. **w+** – para abrir um arquivo tanto para leitura quanto para escrita. Se o arquivo já existir, terá seu conteúdo substituído.

Exemplo: `FILE *arquivo = fopen("uninter.txt", "w+");`

3. **wb** – usado para escrita em arquivos no modo binário.

Exemplo: `FILE *arquivo = fopen("uninter.txt", "wb");`

- **append(a) – escrevendo ao final do arquivo (anexando)**

1. **a** – usamos o modo de abertura "a" para ANEXAR informações ao arquivo. Exemplo: `FILE *arquivo = fopen("uninter.txt", "a");`

2. **a+** – para abrir um arquivo no modo de leitura ou no modo de escrita ao final do arquivo (anexar), usamos o símbolo "+" depois da letra "a".

Exemplo: `FILE *arquivo = fopen("arquivo.txt", "a+");`

3. **ab** – do mesmo modo que a leitura binária "rb" e a escrita binária "wb", podemos anexar informações ao final do arquivo de maneira binária usando o "ab".

Exemplo: `FILE *arquivo = fopen("uninter.txt", "ab");`

Sempre que se terminar de usar um arquivo é preciso fechá-lo. Para realizar essa tarefa, usa-se a função `fclose()` com a seguinte sintaxe:

```
Int fclose(FILE *ponteiro);
```

A função `fclose()` retorna ZERO (0) caso o algoritmo tenha sucesso no fechamento do arquivo.

A Figura 21 apresenta um exemplo com a função `fclose()`:



Figura 21 – Exemplo de um algoritmo com fclose()

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5
6      FILE *arq;
7      arq = fopen("uninter.txt", "r");
8      if((arq=fopen("uninter.txt", "r"))==NULL){
9          printf("ERRO na abertura do arquivo\n\n");
10         system("pause");
11         exit(1);
12     }
13
14     fclose(arq);
15     system("pause");
16     return 0;
17 }
```

Fonte: Elaborado pelo autor.

TEMA 5 – GRAVAÇÃO E LEITURA DE ARQUIVOS

Existem várias funções em C para a operação de gravação e leitura de dados em arquivos. Agora iremos trabalhar com duas funções que gravam e leem um arquivo txt caractere por caractere. A primeira função é o fputc(), que é usada para escrever um caractere de cada vez em um determinado arquivo. A segunda função é o fgetc(), que é usada para obter entrada de um caractere de arquivo por vez, contido na biblioteca stdio.h.

5.1 Função fputc()

Serve para gravar um caractere em um arquivo. Essa função grava o caractere fornecido na posição indicada pelo ponteiro do arquivo e, em seguida, avança o ponteiro do arquivo.

Para realizar essa tarefa, usa-se a função fputc(), que tem a seguinte sintaxe:

```
int fputc(char c, FILE *arquivo);
```

Retorno:

- Se houver erro, a função retorna a constante EOF.
- Se o algoritmo tiver sucesso, retornará o próprio caractere.



A Figura 22 apresenta um exemplo com a função `fputc()`:

Figura 22 – Exemplo de um algoritmo com `fputc()`

```
1  #include <iostream>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int main() {
6
7      char url[] = "uninter.txt";
8      char caractere;
9      FILE *arq;
10     errno_t err;
11     int c;
12
13     err = fopen_s(&arq, url, "w");
14     if (arq == NULL)
15         printf("Erro, nao foi possivel abrir o arquivo\n");
16     else
17         do {
18             printf("digite uma letra:");
19             caractere = getchar();
20             //limpeza buffer teclado
21             while ((c = getchar()) != '\n' && c != EOF) {}
22             fputc(caractere, arq);
23         } while (caractere != '\n');
24
25     fclose(arq);
26     system("pause");
27     return 0;
28 }
```

Fonte: Elaborado pelo autor.

A Figura 23 mostra a execução do algoritmo da Figura 22:

Figura 23 – Execução do algoritmo

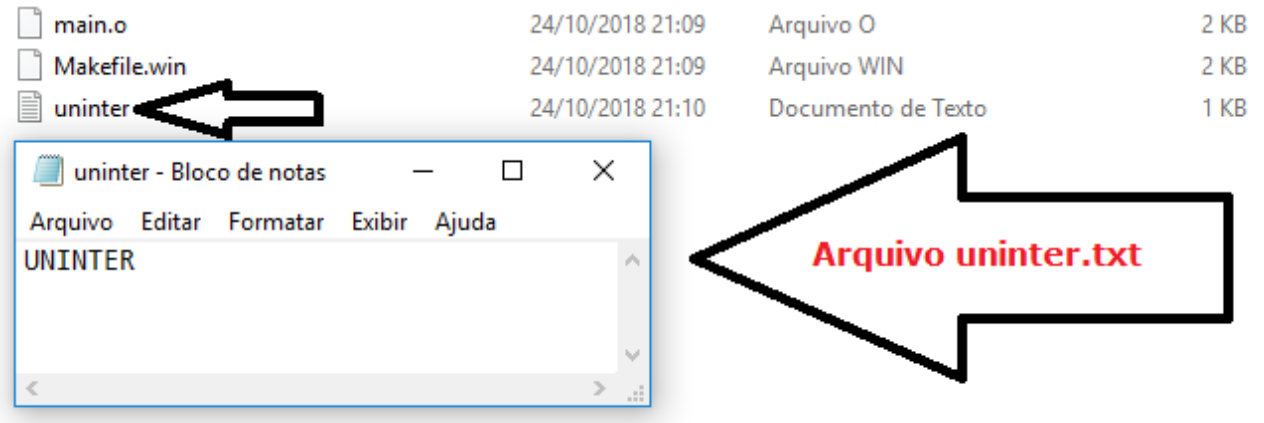
```
Digite uma letra: U
Digite uma letra: N
Digite uma letra: I
Digite uma letra: N
Digite uma letra: T
Digite uma letra: E
Digite uma letra: R
Digite uma letra:
Pressione qualquer tecla para continuar. . .
-----
Process exited after 22.76 seconds with return value 0
Pressione qualquer tecla para continuar. . .
```

Fonte: Elaborado pelo autor.



A Figura 24 mostra a saída do algoritmo, após sua execução na Figura 23:

Figura 24 – Saída do algoritmo



Fonte: Elaborado pelo autor.

5.2 Função `fgetc()`

Essa função lê o caractere presente na posição indicada pelo ponteiro do arquivo e automaticamente já se posiciona no próximo campo, e assim segue lendo até encontrar a constante EOF.

A sintaxe da função `fgetc` é:

```
int fgetc(FILE *arq)
```

E, como comentamos, uma coisa importante que ocorre “por debaixo dos panos” é que, após retornar um caractere, essa função já passa a apontar para o próximo caractere automaticamente, até encontrar -1 (EOF).

Retorno:

- Se houver erro a função, retorna à constante EOF.
- Se o algoritmo tiver sucesso, retornará à leitura do arquivo.

A Figura 25 apresenta um exemplo com a função `fgetc()`:



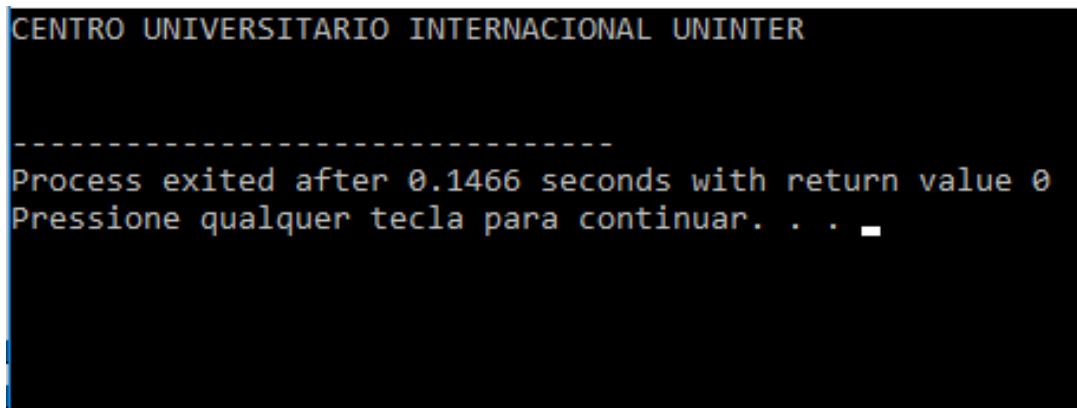
Figura 25 – Exemplo de um algoritmo com fgetc()

```
1  #include <stdio.h>
2
3  int main()
4  {
5      //Arquivo txt que vai ser aberto
6      //O arquivo uninter.txt está no mesmo diretório do projeto
7      char texto[]="uninter.txt";
8      /*variável "letra" do tipo char, que vai
9      receber caractere por caractere do arquivo.*/
10     char letra;
11     FILE *pArq;
12
13     pArq = fopen(texto, "r"); //Abre o arquivo
14     if(pArq == NULL)
15         printf("ERRO, nao foi possivel abrir o arquivo\n");
16     else
17         //Letura do arquivo caractere por caractere
18         while( (letra=fgetc(pArq))!= EOF )
19
20             //Mostrar o caractere na tela.
21             putchar(letra);
22
23     fclose(pArq); //Fecha o arquivo
24
25     return 0;
26 }
```

Fonte: Elaborado pelo autor.

A Figura 26 mostra a execução do algoritmo da Figura 24:

Figura 26 – Saída do algoritmo



```
CENTRO UNIVERSITARIO INTERNACIONAL UNINTER

-----
Process exited after 0.1466 seconds with return value 0
Pressione qualquer tecla para continuar. . . _
```

Fonte: Elaborado pelo autor.



A linguagem de programação C traz uma série de funções usadas para manipular arquivos. Na Tabela 3 estão listadas outras funções relacionadas à manipulação de arquivos.

Tabela 3 – Principais funções de manipulação de arquivos da biblioteca stdio.h

FUNÇÃO	Usada para
fscanf()	Ler um arquivo
fprintf()	Escrever textos (<i>strings</i>) em arquivos
fgets()	Ler uma <i>string</i> num arquivo
fputs()	Inserir uma <i>string</i> no arquivo
fread()	Ler um bloco de dados do arquivo
fwrite()	Escrever um bloco de dados no arquivo
fseek()	Reposicionar o ponteiro
rewind()	Reposicionar o ponteiro para o início do arquivo
ftell()	Retornar a posição do ponteiro.

Fonte: Mizrahi, 2008.

FINALIZANDO

Nesta aula aprendemos os principais conceitos e aplicações de operações em memória, arquivos em C, modos de leituras de arquivos, e gravação e leitura de arquivos. Aproveite a disciplina e bons estudos!



REFERÊNCIAS

ASCENCIO, A. F. G. **Fundamentos da programação de computadores:** algoritmos, Pascal, C/C++ (padrão ANSI), JAVA. 3. ed. São Paulo: Pearson, 2012.

MIZRAHI, V. V. **Treinamento em Linguagem C.** 2. ed. São Paulo: Pearson, 2008.