Ponteiros, funções, alocação dinâmica de memória, os argumentos argc, argv e env da função main(), I/O de disco, estruturas, *bit-fields*, *unions, enumerations*.

Centro de Tecnologia – Departamento de Eletrônica e Computação

Engenharia de Telecomunicações

O BÁSICO DE LINGUAGEM C PARA PROCESSAMENTO DE SINAL

Prof. Fernando DeCastro

```c
#include <stdio.h>

int main() {
  printf("Hello World\n");
  return 0;
}
```

Ponteiros

A compreensão e o uso correto de ponteiros (*pointers*) é crucial para o desenvolvimento de programas com máxima velocidade de processamento, de modo que estes programas possam ser executados em tempo real. Paralelamente à execução em tempo real, o uso de ponteiros possibilita uma eficiente leitura e escrita em memória e em periféricos.

A versatilidade do uso de ponteiros decorre de:

(1) Uma função cujos argumentos são variáveis ponteiro possibilita que a função modifique o valor dos argumentos com os quais a função foi chamada. Isto não é possível se os argumentos da função não forem variáveis ponteiro.

(2) Ponteiros são extensivamente usados no processo de alocação dinâmica de memória, que é crucial quando é necessário alocar memória para vetores e matrizes de centenas de milhares de elementos, como os vetores e matrizes típicos de processamento de sinal em sistemas de comunicação. Estudaremos alocação dinâmica de memória no Cap II.3.

(3) Ponteiros substituem vetores e matrizes com significativa redução do tempo de acesso ao vetor/matriz, seja o acesso uma leitura da memória que armazena o vetor/matriz, seja o acesso uma escrita na memória que armazena o vetor/matriz.

Um ponteiro é uma variável que armazena um endereço na memória. Este endereço na memória dá acesso a um objeto que pode ser um vetor, uma matriz, uma porta, um *buffer*, uma estrutura (veremos **struct** no Cap II.6), um vetor ou matriz de estruturas, etc... O conteúdo armazenado neste objeto pode ser lido ou escrito usando o endereço armazenado na variável ponteiro. Mais usualmente, o endereço armazenado em uma variável ponteiro é o endereço na memória onde está localizada outra variável, ou o endereço na memória onde inicia uma área de armazenamento do tipo vetor. Mas o endereço armazenado no ponteiro pode se referir também ao endereço de uma porta de I/O, ou também ao endereço de um segmento da RAM de propósito especial, como uma área de armazenamento auxiliar (*buffer*).

Ponteiros são uma das características mais fortes do C, concedendo ao programador um significativo grau de liberdade devido à versatilidade de acesso à memória. No entanto, ponteiros são também algo perigosos se não forem usados corretamente. Por exemplo, ponteiros que, por descuido do programador, armazenam um endereço fora do escopo do programa podem acessar áreas de memória sensíveis, causando desde o travamento do sistema até bugs absolutamente bizarros e difíceis de serem corrigidos (o que, infelizmente, acontece com alguma recorrência durante o desenvolvimento de um programa ...).

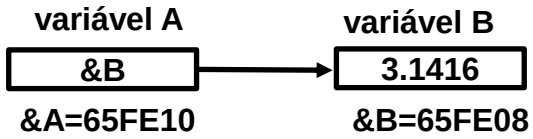
Ponteiros

Consideremos uma variável A que armazena o endereço de uma variável B através da declaração A=&B, e esta variável B armazena o valor 3.1416. Então a variável A é um ponteiro para a variável B e seu valor. O valor armazenado em B pode ser acessado (lido ou escrito) através da declaração *A, que significa "valor armazenado no endereço da variável apontada por A". A declaração C=*A lê o valor de B através do ponteiro A e escreve (atribui) o valor de B à variável C , conforme segue:

```
#include<stdio.h>
void main(void){
double *A; // declara o ponteiro A
double B=3.1416; // declara e inicializa a variavel B
double C; // declara a variavel C

printf("O valor armazenado na variavel B eh B=%lf\n",B); // imprime na tela do console
printf("O endereco na memoria da variavel B eh &B=%p\n",&B); // %p eh o especificador de formato p/ imprimir endereco

A=&B; // O ponteiro A recebe o endereco da variavel B, i.e., A passa a apontar p/ B
C=*A; // o valor *A armazenado na variavel apontada por A eh lido da memoria e eh atribuido (escrito) na variavel C

printf("\nO valor armazenado na variavel apontada por A eh *A=%lf e o valor da variavel C=*A eh C=%lf\n",*A,C);
printf("O endereco armazenado em A e que corresponde ao endereco da variavel apontada por A eh A=%p\n",A);

printf("O endereco na memoria da variavel A eh &A=%p\n",&A);
}
```

Execução na tela do console (endereços estão em hexadecimal):

O valor armazenado na variavel B eh B=3.141600
O endereco na memoria da variavel B eh &B=000000000065FE08



O valor armazenado na variavel apontada por A eh *A=3.141600 e o valor da variavel C=*A eh C=3.141600
O endereco armazenado em A e que corresponde ao endereco da variavel apontada por A eh A=000000000065FE08
O endereco na memoria da variavel A eh &A=000000000065FE10

Ponteiros

Importante manter em mente que as operações com ponteiros devem ser analisadas sempre que possível com base no mapa de memória associado às operações. Por exemplo, consideremos o código abaixo:

```
float mult= 315.2; /* declara e inicializa a variavel mult */
int sum = 32000; /* declara e inicializa a variavel sum */
char letter = 'A', /* equivalente a char letter = 65 */
int *set; /* declara o ponteiro set para variáveis int*/
float *ptr1; /* declara o ponteiro ptr1 para variáveis float */
set=&sum; /* inicializa o ponteiro set com o endereco de sum */
ptr1=&mult; /* inicializa o ponteiro ptr1 com o endereco de mult */
```

Este código gera o mapa de memória abaixo, ou um mapa algo semelhante (os endereços absolutos possivelmente variem com o tipo de sistema operacional e com o tipo de CPU):

Endereço de memória: (hexadecimal)	Valor da variável na memória (decimal exceto indicação contrária):	
FA10	3.152 E 02	→ mult (4 bytes)
FA11		
FA12		
FA13		
FA14	32000	→ sum (2 bytes)
FA15		
FA16	65	→ letter (1 byte) ⇒ Valor ASCII do caracter 'A'
FA17	FA14 (hex)	→ set (2 bytes) ⇒ set aponta para sum
FA18		
FA19	FA10 (hex)	→ ptr1 (2 bytes) ⇒ ptr1 aponta para mult
FA1A		

Ponteiros

Conforme vimos no exemplo do slide 3, se uma variável é um ponteiro que vai armazenar um endereço, então ela deve ser declarada como tal. Uma declaração de uma variável ponteiro consiste em um tipo base, um ***** e o nome da variável. A forma geral para declaração de uma variável ponteiro é:

tipo *nome_da_variavel;

onde o **tipo** é qualquer tipo de dado válido em C (incluindo tipos de dados definidos por **typedef** – a ser visto adiante) e o **nome_da_variável** é o nome do ponteiro, o qual, conforme vimos no exemplo do slide 3, armazena o endereço. O tipo base do ponteiro define qual tipo de variáveis o ponteiro pode apontar. Por exemplo, as declarações abaixo criam um ponteiro **char p** e dois ponteiros **int temp** e **inicio**.

```
char *p;
int *temp, *início;
```

Operadores para ponteiros:

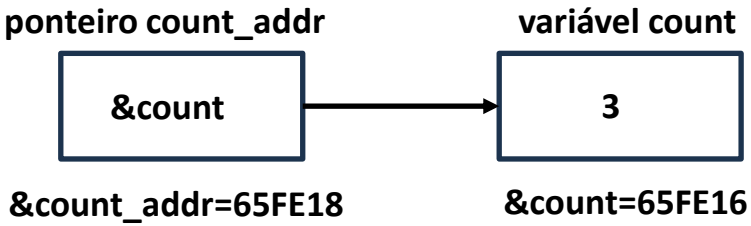
Existem dois operadores para operações com ponteiros: o operador **&** e o operador ***.**

O **&** é um operador unário que retorna o endereço na memória do seu operando (um operador unário requer somente um operando). Por exemplo, considere as declarações

```
int count;
int *count_addr;
count_addr = &count;
```

A declaração **count_addr = &count** armazena no ponteiro **count_addr** o endereço na memória da variável **count**. Esse endereço é a localização da variável **count** na memória do sistema (computador, microprocessador, etc...). Importante notar que a declaração **count_addr = &count** não afeta o valor armazenado em **count**, ela apenas faz com que o ponteiro **count_addr** aponte para a variável **count**.

Vamos supor que a variável **count** armazene o valor 3 e esteja localizada no endereço **65FE16** da memória (todos os endereços estão em hexadecimal). Então, após a atribuição **count_addr = &count**, o ponteiro **count_addr** armazena o endereço **65FE16**, endereço que aponta para a variável **count** conforme mostrado abaixo:

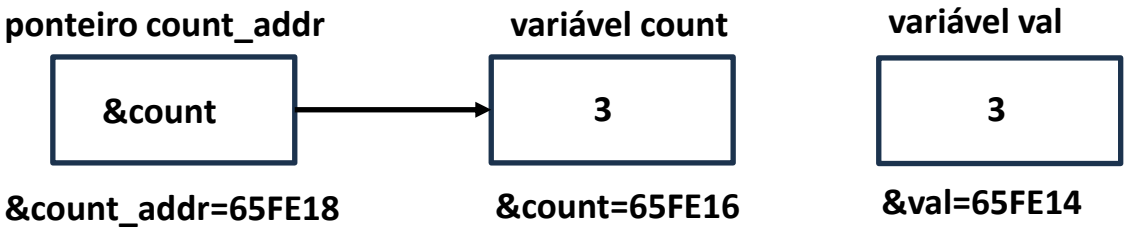


Ponteiros

O operador * é um operador unário que retorna o valor armazenado no endereço da variável apontada pelo ponteiro, sendo o nome do ponteiro declarado após o operador *. Por exemplo , consideremos as declarações

```
int val, count;
int *count_addr;
count_addr = &count;
val = *count_addr;
```

A declaração **val = *count_addr** copia o valor 3 armazenado na variável **count** , cujo endereço é apontado pelo ponteiro **count_addr,** e armazena em **val** o valor copiado de **count**. Especificamente, a declaração **val = *count_addr** é lida como "a variável **val** recebe o valor que está armazenado no endereço apontado pelo ponteiro **count_addr**", conforme mostrado abaixo:



Note que se tivéssemos declarado **val** como um **char** (1 byte) a declaração **val = *count_addr** copiaria apenas 1 byte do valor 3 armazenado na variável inteira **count ,** cujo tamanho é de 2 bytes na memória. Muito cuidado portanto com o tipo base do ponteiro quando se copia valores através do operador * "valor no endereço".

Muito cuidado também para não confundir o operador de multiplicação * com operador "valor no endereço" *. Tanto **&** como * têm precedência maior que todos os outros operadores aritméticos, exceto o menos unário, com o qual eles se igualam.

Ponteiros

O código fonte https://www.fccdecastro.com.br/CursoC&C++/C/VecPtr.c a seguir mostra um exemplo de uso dos operadores **&** e ***** na interação entre vetores e ponteiros.

```c
/* imprime o vetor Vect na tela do console usando ponteiros */
#include <stdio.h>
#define NELEM 4
void main(void)
{
float Vect[NELEM]={0.1, 1.2, 7.7, 8.0};
float Vtmp[NELEM];
unsigned n;
float *Ptr;

Ptr=&Vect[0]; /* Ptr recebe o endereco do primeiro elemento de Vect[]
                 e passa a apontar para Vect[] */

/* Nota 1: Experimente substituir a declaracao Ptr=&Vect[0]; pela declaracao
Ptr=Vect;. Voce vai observar que o programa vai funcionar igualmente. Isto
acontece porque em C o nome de um vetor ou matriz corresponde ao endereco
do primeiro elemento armazenado no vetor/matriz. */

for(n=0;n<NELEM;n++)Vtmp[n]= *(Ptr+n); /* copia os elementos de Vect[] para
                o vetor Vtmp[] atraves do ponteiro Ptr, que aponta para o endereco
                do primeiro elemento de Vect[]. Note que a cada incremento de n, Ptr
                incrementa de 4 bytes o endereco apontado porque Ptr foi declarado como
                um ponteiro float e cada float armazenado na memoria ocupa 32bits (4 bytes) */
```

```
24
25  /* Nota 2: Experimente substituir a declaracao Vtmp[n]= *(Ptr+n); pela declaracao
26  Vtmp[n]= *Ptr++;. Voce vai observar que o programa vai funcionar igualmente. Isto
27  acontece porque o ponteiro Ptr vai incremmentar (++) de 4 bytes o endereço nele
28  armazenado a cada vez que a operacao Ptr++ for executada no loop for. Mas note que
29  ao final do loop for(n=0;n<NELEM;n++) Ptr nao armazena mais o endereco original
30  (porque ele foi sucessivamente incrementado). Isto pode ser um problema se quisermos
31   usar o endereco armazenado em Ptr mais adiante no programa (o que nao eh o caso) */
32
33  for(n=0;n<NELEM;n++)printf("%f ", Vtmp[n] ); /* imprime Vtmp[n] */
34  }
```

Execução na tela do console:

0.100000 1.200000 7.700000 8.000000

Ponteiros

O código fonte https://www.fccdecastro.com.br/CursoC&C++/C/VecPtr1.c a seguir mostra um exemplo de uso alternativo dos operadores **&** e ***** na interação entre vetores e ponteiros.

```c
/* imprime o vetor Vect na tela do console usando ponteiros - V1 */
#include <stdio.h>
#define NELEM 4
void main(void)
{
float Vect[NELEM]={0.1, 1.2, 7.7, 8.0};
float Vtmp[NELEM];

unsigned n;
float *Ptr;

Ptr=Vect; /* Ptr recebe o endereco do primeiro elemento de Vect[]
             e passa a apontar para Vect[] */

for(n=0;n<NELEM;n++)Vtmp[n]= Ptr[n]; /* copia os elementos de Vect[] para
                o vetor Vtmp[] atraves do ponteiro Ptr, que aponta para o endereco
                do primeiro elemento de Vect[]. Note que a cada incremento de n, Ptr
                incrementa de 4 bytes o endereco apontado porque Ptr foi declarado como
                um ponteiro float e cada float armazenado na memoria ocupa 32bits (4 bytes).

                Note tambem que a declaracao
                Ptr[n] equivale a declaracao *(Ptr+n) */

for(n=0;n<NELEM;n++)printf("%f ", Vtmp[n] ); /* imprime Vtmp[n] */
}
```

Execução na tela do console:

0.100000 1.200000 7.700000 8.000000

O tipo base de um ponteiro:

Conforme vimos nos exemplos nos slides 8 a 10 , dado um ponteiro que armazena o endereço do primeiro elemento de um vetor ou matriz, o compilador "sabe" quantos bytes deve ser avançado no endereço nele armazenado cada vez que o ponteiro é incrementado de uma posição porque o compilador "sabe" qual é o tipo base do ponteiro. Portanto o tipo base do ponteiro deve ser o mesmo tipo da variável que o ponteiro aponta através do endereço nele armazenado. Um ponteiro **char** aponta para um vetor **char**, um ponteiro **int** aponta para um vetor **int**, um ponteiro **float** aponta para um vetor **float**, um ponteiro **double** aponta para um vetor **double**, um ponteiro **struct** aponta para um vetor de elementos **struct**.

Mesmo que o ponteiro armazene o endereço inicial de uma única variável, o tipo base deve ser compatível entre variável e ponteiro. Esta situação já foi analisada no penúltimo parágrafo do slide 7, em que declaramos a variável **val** como um **char** (1 byte) e declaramos o ponteiro ***count_addr** como um **int**. E daí fizemos **count_addr = &count** de modo que **count_addr** passou a apontar para o endereço de uma variável **int count**. Nesta situação, a declaração **val = *count_addr** copiaria apenas 1 byte do valor de 2 bytes armazenado na variável inteira **count**. O código abaixo ilustra situação semelhante, com ponteiro int e variável float.

```
/* atribuicao errada de ponteiro */
#include <stdio.h>
void main(void)
{
  float x=3.1416, y; int *Ptr;
  Ptr=&x;  /* ponteiro int Ptr recebe o endereco de float x
(Errado!) */
  y=*Ptr; /* float y recebe o valor armazenado no endereço
apontado por int Ptr (Errado!) */
  printf("y=%f\n",y);
}
```

Execução na tela do console:

y=1078530048.000000

Se declararmos **float *Ptr** ao invés de **int *Ptr** o programa executa corretamente:

y=3.141600

Note o **warning** emitido pelo compilador:

Line	Col	File	Message
		C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Li...	**In function 'main':**
7	6	C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Lingua...	[Warning] assignment to 'int *' from incompatible pointer type 'float *' [-Wincompatible-pointer-types]

Expressões com ponteiros:

Atribuição de Ponteiros:

Como qualquer variável, um ponteiro pode ser usado no lado direito de uma declaração para atribuir seu valor a um outro ponteiro. Por exemplo:

```
/* atribuindo um ponteiro a outro ponteiro */
#include<stdio.h>
void main(void)
{
  int x=888;
  int *p1, *p2;

  p1=&x; // o endereco de x eh atribuido a p1
  p2=p1; // o endereco armazenado em p1 eh atribuido a p2

/* imprime o valor hexadecimal do endereço armazenado em p2 e que eh
 o endereco de x devido as declaracoes p1=&x; e p2=p1; */
printf("O endereco apontado por p2 (que eh o endereco de x) eh %p.\n", p2);

/* imprime o valor de x: */
printf("O valor armazenado no endereco apontado por p2 (que eh valor de x) eh %d.\n", *p2);
}
```

Execução na tela do console:

```
O endereco apontado por p2 (que eh o endereco de x) eh 000000000065FE0C.
O valor armazenado no endereco apontado por p2 (que eh valor de x) eh 888.
```

Aritmética com Ponteiros:

Somente duas operações aritméticas podem ser usadas com ponteiros: adição e subtração. Consideremos a seguinte sequência de declarações:

```
float x;
float *p1;
p1=&x;
```

E vamos supor que o endereço de x na memória seja &x=65FE00**.** Após a declaração de incremento do endereço armazenado em p1 conforme abaixo

```
p1++;
```

o endereço armazenado em p1 será 65FE04 (e não 65FE01) porque p1 é um ponteiro que aponta para o endereço de uma variável **float**, que ocupa 4 bytes na memória.

Cada vez que p1 é incrementado ele apontará p/ o próximo **float**, situado 4 bytes adiante no endereçamento de memória.

O mesmo é válido para decremento. Por exemplo, vamos supor que o endereço de x na memória seja &x=65FE08**.** Após a declaração de decremento do endereço armazenado em p1 conforme abaixo

```
p1--;
```

o endereço armazenado em p1 será 65FE04 porque p1 é um ponteiro que aponta para o endereço de uma variável **float**, que ocupa 4 bytes na memória.

Nota: É instrutivo verificar as operações de incremento e decremento de endereços hexadecimais acima com a calculadora do Windows no modo "Programador – QWORD".

Portanto, cada vez que um ponteiro é incrementado ele apontará para a próxima localização na memória do seu tipo base. E, cada vez que ele é decrementado, o ponteiro apontará para a localização anterior na memória do seu tipo base. No caso de ponteiros **char**, a aritmética é a aritmética normal, porque **char** ocupa 1 byte na memória. Porém, todos os outros ponteiros incrementarão ou decrementarão do tamanho do tipo base para o qual apontam. Por exemplo, o incremento de um ponteiro **char** resulta no incremento de 1 byte no endereço nele armazenado, o incremento de um ponteiro **int** resulta no incremento de 2 bytes no endereço nele armazenado, o incremento de um ponteiro **float** resulta no incremento de 4 bytes no endereço nele armazenado, e assim sucessivamente.

Ponteiros

Mesma regra vale quando se adiciona ou subtrai um valor inteiro a um ponteiro que armazena um endereço:

```
/* deslocando um ponteiro de multiplas posicoes */
#include<stdio.h>
void main(void)
{
float x=3.2, y=4.3, z=5.5, *ptr;
printf("&x=%p *x=%f\n", &x,x);
printf("&y=%p *y= %f\n", &y,y);
printf("&z=%p *z=%f\n", &z,z);

ptr=&z;
printf("\nApos ptr=&z ptr=%p *ptr=%f\n", ptr,*ptr);

ptr+=2; //ptr=ptr+2;
printf("Apos ptr=ptr+2 ptr=%p *ptr=%f\n", ptr,*ptr);

ptr-=1; //ptr=ptr-1;
printf("Apos ptr=ptr-1 ptr=%p *ptr=%f\n", ptr,*ptr);
}
```

Execução na tela do console:

```
&x=000000000065FE14 *x=3.200000
&y=000000000065FE10 *y= 4.300000
&z=000000000065FE0C *z=5.500000

Apos ptr=&z ptr=000000000065FE0C *ptr=5.500000
Apos ptr=ptr+2 ptr=000000000065FE14 *ptr=3.200000
Apos ptr=ptr-1 ptr=000000000065FE10 *ptr=4.300000
```

Portanto, dado um ponteiro **ptr** que armazena um endereço **addr**, ao adicionarmos um numero inteiro **n** ao ponteiro **ptr** o ponteiro passará a armazenar o novo endereço **addr+n*bytes_tipo_base**, onde **bytes_tipo_base** é o número de bytes ocupados pelo tipo base do ponteiro (**char 1byte, int 2bytes, float 4bytes, double 8bytes,** e assim por diante).

Comparaçao entre Ponteiros:

É possível comparar dois ponteiros em uma expressão relacional. Por exemplo, dados dois ponteiros p e q, a seguinte declaração é perfeitamente válida:

```
if(p<q) printf("p aponta para um endereço de memória mais baixo que q\n");
```

Geralmente, comparações entre ponteiros devem ser usadas somente quando dois ou mais ponteiros estão apontando para um objeto em comum.

Ponteiros , vetores e matrizes:

Conforme vimos nos exemplos nos slides 8 a 10, há uma relação entre ponteiros e vetores. Consequentemente, também há uma relação entre ponteiros e matrizes, dado que uma matriz é armazenada na memória na forma de um vetor, conforme vimos nos exemplos dos slides 131 a 134 do Cap I.3. Consideremos o código abaixo:

```
#include <stdio.h>
void main(void)
{
char str[]="ABCDEFGHIJK", ch,*ptr;
ptr = str; // equivale a ptr=&str[0]
ch= str[4]; // ch recebe o valor de str[4]

printf("ch=%u=%c\n",ch,ch);
printf("ptr[4]=%u=%c\n",ptr[4],ptr[4]);
printf("str[4]=%u=%c\n",str[4],str[4]);
}
```

Cuja execução na tela do console resulta:

```
ch=69=E
ptr[4]=69=E
str[4]=69=E
```

Após a declaração **ptr = str** o ponteiro **ptr** armazena o endereço do primeiro elemento do vetor **str[]**. Isto acontece porque em C **o nome de um vetor (ou matriz) sem a especificação de índice é o endereço do primeiro elemento do vetor (ou matriz)**.

A seguir, após a declaração **ch= str[4]** , o valor do quinto elemento no vetor **str[]** é atribuído à variável **ch**, de modo que **ch** armazena o valor 69 que corresponde ao caractere ASCII 'E'

Dado que fizemos **ptr = str** , então a declaração **ch= str[4]** pode alternativamente ser substituída por **ch=*(ptr+4)**, que equivale à declaração **ch=ptr[4]** (experimente alterar o código acima com estas alternativas e verifique que o resultado é o mesmo).

Ponteiros

Qual é a “moral da estória” no exemplo do slide anterior? Vimos neste exemplo que o quinto elemento do vetor **str[]** é acessado e atribuído a uma variável **ch** através da declaração **ch= str[4]** . Paralelamente, apontamos o ponteiro **ptr** para o 1° elemento do vetor **str[]** através da declaração **ptr = str** , e, fazendo assim, o ponteiro **ptr** passou a representar o vetor **str[]**, de modo que a declaração **ch= ptr[4]** substituí a declaração **ch= str[4]** .

Porque usar a declaração **ch= ptr[4]** se a mesma resulta no mesmo resultado de **ch= str[4]**? É preferível usar **ch= ptr[4]** porque ela é **executada mais rapidamente** do que **ch= str[4]**. É claro que neste exemplo a diferença entre o tempo de execução da indexação de ponteiro e o tempo de execução da indexação de vetor é desprezível. Mas para vetores de centenas de milhares de elementos a diferença no tempo de execução é nitidamente perceptível entre os dois métodos de indexação.

Apenas em uma situação a rapidez da indexação de vetor (ou matriz) se aproxima da rapidez da indexação de ponteiro: Quando o vetor (ou matriz) não for acessado (seja leitura ou escrita no vetor/matriz) através de índices ordenadamente ascendentes ou descendentes. Mas daí, fica a pergunta: Em quais situações em Engenharia isto acontece? Certamente em em não muitas situações ...

Ponteiros

Obtendo o endereço de um elemento de um vetor (ou matriz):

É possível atribuir o endereço de um elemento específico de um vetor (ou matriz) aplicando o operador **&** para o elemento de um vetor (ou matriz) indexado. Por exemplo, a declaração abaixo armazena no ponteiro **p** o endereço do terceiro elemento do vetor **x** :

p= &x[2];

Essa declaração é útil para, por exemplo, encontrar uma *substring* em uma frase de texto. Por exemplo, o código abaixo lê uma frase de texto do teclado e armazena em uma *string*. Daí localiza o primeiro caractere de espaço na *string* e imprime na tela do console a *string* que resta após o espaço encontrado:

```
#include <stdio.h>
void main(void)
{
  char s[80],*p;
  int i;
  printf("Digite uma linha de texto: ");
  gets(s);

  /* encontra o primeiro espaço ou o fim da string */
  for (i=0; s[i] && s[i]!=' '; i++);
  p=&s[i];

  printf("A partir do primeiro espaco (inclusive) no texto digitado resta o seguinte texto:\n");
  printf(p);
}
```

Execução na tela do console:

```
Digite uma linha de texto: Alfa Beta Gama Delta
A partir do primeiro espaco (inclusive) no texto digitado resta o seguinte texto:
 Beta Gama Delta
```

Ponteiros

Comprovando que uma matriz é armazenada na memória na forma de um vetor:

Nos exemplos dos slides 131 a 134 do Cap I.3 foi mostrada a maneira como uma matriz é armazenada na memória na forma de um vetor.

Naquela etapa de nosso curso ainda não havíamos visto o conceito de ponteiro, então a referida maneira foi mostrada sem comprovação.

O código fonte https://www.fccdecastro.com.br/CursoC&C++/C/MatxyVecPtr.c no próximo slide comprova numericamente através do **cast** de **int Mat[Ny][Nx]** para **int *V** (na linha 16) o que foi apenas mostrado no exemplo dos slides 131 e 132 do Cap I.3.

Note que o **cast** instrui para que a matriz **Mat[Ny][Nx]** seja acessada na forma de um vetor **V[n],** **sem haver qualquer cópia dos elementos de Mat[][] para V[], i.e, V apenas aponta para os elementos de Mat[][]**.

Note ainda que o **cast** na linha 16

V= (int *)Mat; /* cast de int Mat[Ny][Nx] para int *V; */

é equivalente a

V= (int *)&Mat[0][0]; /* cast de int Mat[Ny][Nx] para int *V; */

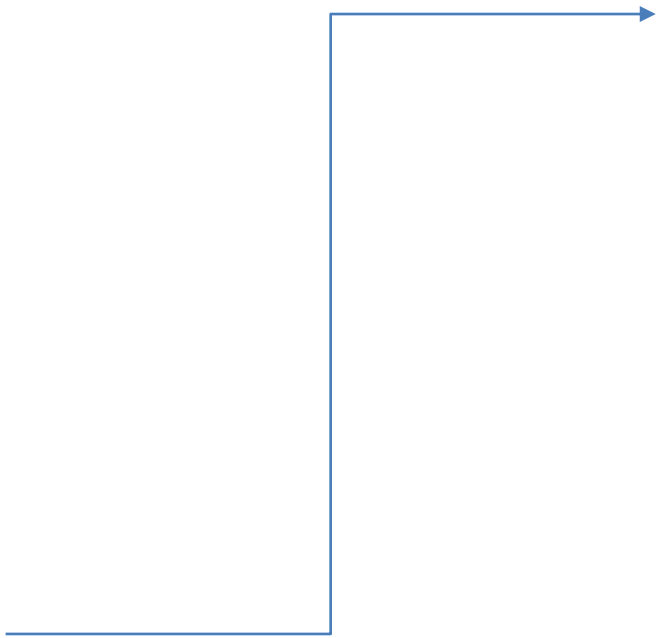
Ponteiros

```c
/* Armazena os indices n= 0 a Nx*Ny-1 de V[n] em uma matriz 2D Mat[Ny][Nx] atraves
 da atribuicao Mat[y][x]=n, onde n= (Nx*y)+x. Dai faz um cast de Mat[Ny][Nx] para
 o ponteiro *V  e imprime na tela do console os elementos de V[n] com  referencia
 aos indices y e x de Mat[y][x]. E finalmente imprime a totalidade de V[n]. */
#include<stdio.h>
#define Ny 5
#define Nx 4
void main(void)
{
int Mat[Ny][Nx]; /* declara a matriz int Mat[][] de Ny elementos no eixo y e Nx
                    elementos no eixo x */
int *V; // declara o ponteiro V
int y,x; // indices das coordenadas y e z dos elementos da matriz Mat[y][x]
int n; // indice n do vetor v{n];

V= (int *)Mat; /* cast de int Mat[Ny][Nx] para int *V; */

    for(y=0;y<Ny; y++){
        for(x=0;x<Nx;x++){
            n=(Nx*y)+x; /* indice n do vetor V[n] */
            Mat[y][x]=n; /* Mat[y][x] recebe o indice do vetor V[n] */
            printf("Mat[%d][%d]= V[%d]\n",y,x,V[n]); /* imprime V[n] com referencia
                    aos indices y e x de Mat[y][x] */
        }
    }
printf("\n");
for(n=0;n<Ny*Nx;n++)printf("V[%d]= %d\n",n,V[n]); /* imprime a totalidade de V[n]  */
}
```

Ponteiros

Execução na tela do console:

Mat[0][0]= V[0]
Mat[0][1]= V[1]
Mat[0][2]= V[2]
Mat[0][3]= V[3]
Mat[1][0]= V[4]
Mat[1][1]= V[5]
Mat[1][2]= V[6]
Mat[1][3]= V[7]
Mat[2][0]= V[8]
Mat[2][1]= V[9]
Mat[2][2]= V[10]
Mat[2][3]= V[11]
Mat[3][0]= V[12]
Mat[3][1]= V[13]
Mat[3][2]= V[14]
Mat[3][3]= V[15]
Mat[4][0]= V[16]
Mat[4][1]= V[17]
Mat[4][2]= V[18]
Mat[4][3]= V[19]



V[0]= 0
V[1]= 1
V[2]= 2
V[3]= 3
V[4]= 4
V[5]= 5
V[6]= 6
V[7]= 7
V[8]= 8
V[9]= 9
V[10]= 10
V[11]= 11
V[12]= 12
V[13]= 13
V[14]= 14
V[15]= 15
V[16]= 16
V[17]= 17
V[18]= 18
V[19]= 19

Ponteiros

A seguir, a mesma comprovação para o caso de uma matriz 3D.

O código fonte https://www.fccdecastro.com.br/CursoC&C++/C/MatxyzVecPtr.c no próximo slide comprova numericamente através do **cast** de **int Mat[Nz][Ny][Nx]** para **int *V** (na linha 17) o que foi apenas mostrado no exemplo dos slides 133 e 134 do Cap I.3.

Note que o **cast** instrui para que a matriz **Mat[Nz][Ny][Nx]** seja acessada na forma de um vetor **V[n], sem haver qualquer cópia dos elementos de Mat[][][] para V[], i.e, V apenas aponta para os elementos de Mat[][][]**.

Note ainda que o **cast** na linha 17

V= (int *)Mat; /* cast de int Mat[Nz][Ny][Nx] para int *V; */

é equivalente a

V= (int *)&Mat[0][0][0]; /* cast de int Mat[Nz][Ny][Nx] para int *V; */

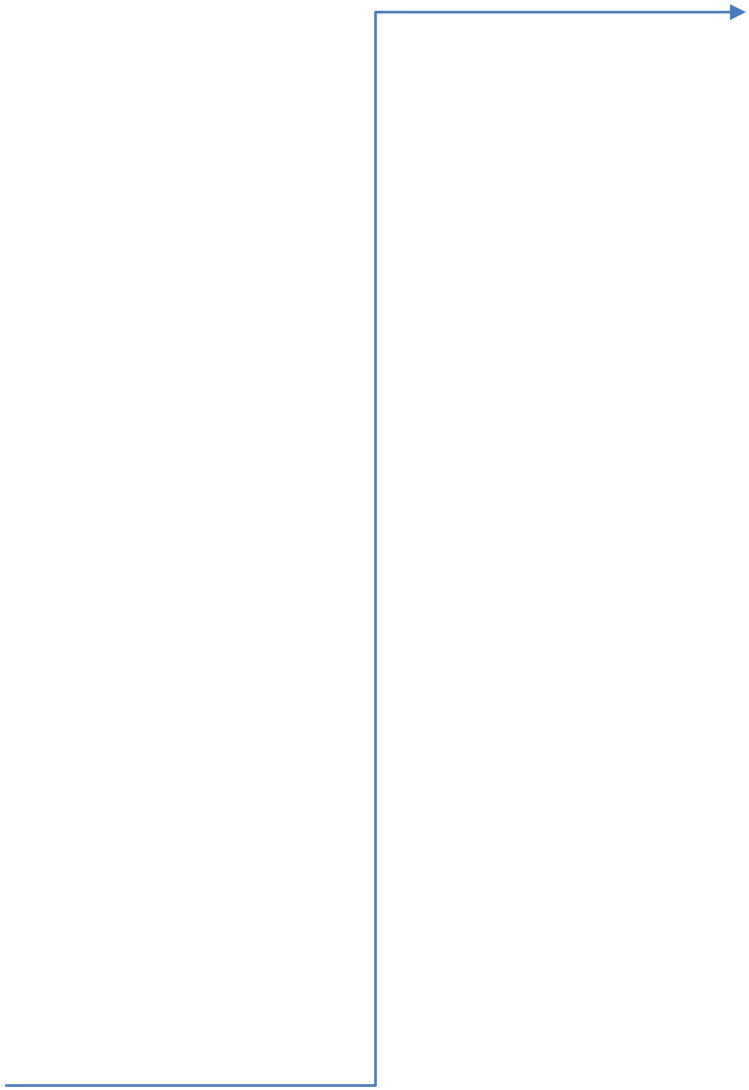
Ponteiros

```c
/* Armazena os indices n= 0 a Nz*Nx*Ny-1 de V[n] em uma matriz 3D Mat[Nz][Ny][Nx] atraves
 da atribuicao Mat[z][y][x]=n, onde n= (Ny*Nx*z)+(Nx*y)+x. Dai faz um cast de  Mat[Nz][Ny][Nx]
 para o ponteiro *V  e imprime na tela do console os elementos de V[n] com  referencia aos
 indices y e x de Mat[y][x]. E finalmente imprime a  totalidade de V[n]. */
#include<stdio.h>
#define Nz 2
#define Ny 3
#define Nx 4
void main(void)
{
int Mat[Nz][Ny][Nx]; /* declara a matriz 3D Mat de Nz elementos em z, Ny elementos
                        em y e Nx elementos em x */
int z,y,x; // indices das coordenadas z, y e x dos elementos da matriz Mat[z][y][x]
int *V; // declara o ponteiro V
int n; // indice n do vetor V[n];

V= (int *)Mat; /* cast de int Mat[Nz][Ny][Nx] p/ int *V. Equivale a V= (int*)&Mat[0][0][0]. */

for(z=0;z<Nz; z++){
 for(y=0;y<Ny; y++){
   for(x=0;x<Nx;x++){
     n=(Ny*Nx*z)+(Nx*y)+x; /* indice n do vetor V[n] */
     Mat[z][y][x]=n; /* Mat[z][y][x] recebe o indice do vetor V[n] */
     printf("Mat[%d][%d][%d]= V[%d]\n",z,y,x,V[n]); /* imprime V[n] c/ referencia aos
                                                       indices z,y e x de Mat[z][y][x] */
    }
  }
}
printf("\n");
for(n=0;n<Nz*Ny*Nx;n++)printf("V[%d]= %d\n",n,V[n]); /* imprime a totalidade de V[n]  */
}
```

Ponteiros

Execução na tela do console:

Mat[0][0][0]= V[0]
Mat[0][0][1]= V[1]
Mat[0][0][2]= V[2]
Mat[0][0][3]= V[3]
Mat[0][1][0]= V[4]
Mat[0][1][1]= V[5]
Mat[0][1][2]= V[6]
Mat[0][1][3]= V[7]
Mat[0][2][0]= V[8]
Mat[0][2][1]= V[9]
Mat[0][2][2]= V[10]
Mat[0][2][3]= V[11]
Mat[1][0][0]= V[12]
Mat[1][0][1]= V[13]
Mat[1][0][2]= V[14]
Mat[1][0][3]= V[15]
Mat[1][1][0]= V[16]
Mat[1][1][1]= V[17]
Mat[1][1][2]= V[18]
Mat[1][1][3]= V[19]
Mat[1][2][0]= V[20]
Mat[1][2][1]= V[21]
Mat[1][2][2]= V[22]
Mat[1][2][3]= V[23]



V[0]= 0
V[1]= 1
V[2]= 2
V[3]= 3
V[4]= 4
V[5]= 5
V[6]= 6
V[7]= 7
V[8]= 8
V[9]= 9
V[10]= 10
V[11]= 11
V[12]= 12
V[13]= 13
V[14]= 14
V[15]= 15
V[16]= 16
V[17]= 17
V[18]= 18
V[19]= 19
V[20]= 20
V[21]= 21
V[22]= 22
V[23]= 23

Ponteiros para ponteiros :

Uma variável que é um ponteiro para um ponteiro deve ser declarada como tal. **Isso é feito colocando-se um asterisco adicional na frente do seu nome**. Por exemplo, a declaração abaixo diz ao compilador que novobalanço é um ponteiro para um ponteiro do tipo float:

```
float **novobalanço;
```

É importante entender que **novobalanço** não é um ponteiro para um número de ponto flutuante, mas, sim, um ponteiro para um ponteiro float.

Para acessar o valor desejado apontado indiretamente por um ponteiro para um ponteiro, **o operador asterisco deve ser aplicado duas vezes**, conforme mostrado no código fonte a seguir:

```
#include <stdio.h>
void main(void)
{
float x=3.4, *p, **q;

printf("O valor armazenado na variavel x eh x=%f\n",x); // imprime na tela do console
printf("O endereco na memoria da variavel x eh &x=%p\n",&x); // %p eh o especificador de formato p/ imprimir
endereco

p=&x; // O ponteiro p recebe o endereco da variavel x, i.e., p passa a apontar p/ x
printf("O endereco armazenado no ponteiro p eh p=%p\n",p);
printf("O valor armazenado na variavel apontada por p eh *p=%f\n",*p);
printf("O endereco na memoria do ponteiro p eh &p=%p\n",&p);

q=&p; // O ponteiro q recebe o endereco do ponteiro p, i.e., q passa a apontar p/ p
printf("O endereco armazenado no ponteiro q eh q=%p\n",q);
printf("O valor armazenado na variavel indiretamente apontada por q eh **q=%f\n",**q);
printf("O endereco na memoria do ponteiro q eh &q=%p\n",&q);
}
```

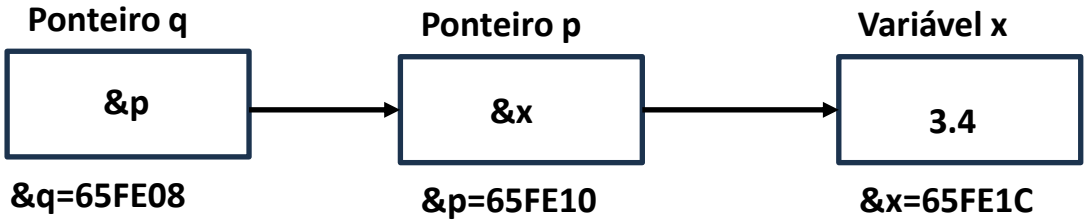
Ponteiros para ponteiros :

Uma variável que é um ponteiro para um ponteiro deve ser declarada como tal. **Isso é feito colocando-se um asterisco adicional na frente do seu nome**. Por exemplo, a declaração abaixo diz ao compilador que novobalanço é um ponteiro para um ponteiro do tipo float:

float **novobalanço;

É importante entender que **novobalanço** não é um ponteiro para um número de ponto flutuante, mas, sim, um ponteiro para um ponteiro float.

Para acessar o valor apontado indiretamente por um ponteiro para um ponteiro, **o operador asterisco deve ser aplicado duas vezes**, conforme mostrado no código fonte no slide que segue. O diagrama de memória resultante do código fonte do slide que segue é mostrado abaixo.



Descrição do diagrama de memória:

O valor armazenado na variável x é x=3.4.
O endereço na memória da variável x é &x=65FE1C.
O endereço armazenado no ponteiro p é p=65FE1C.
O valor armazenado na variável apontada por p é *p=3.4.
O endereço na memória do ponteiro p é &p=65FE10.
O endereço armazenado no ponteiro q é q=65FE10.
O valor armazenado na variável indiretamente apontada por q é **q=3.4.
O endereço na memória do ponteiro q é &q=65FE08

Ponteiros

```
#include <stdio.h>
void main(void)
{
float x=3.4, *p, **q;

printf("O valor armazenado na variavel x eh x=%f\n",x); // imprime na tela do console
printf("O endereco na memoria da variavel x eh &x=%p\n",&x); // %p eh o especificador de formato p/ imprimir endereco

p=&x; // O ponteiro p recebe o endereco da variavel x, i.e., p passa a apontar p/ x
printf("O endereco armazenado no ponteiro p eh p=%p\n",p);
printf("O valor armazenado na variavel apontada por p eh *p=%f\n",*p);
printf("O endereco na memoria do ponteiro p eh &p=%p\n",&p);

q=&p; // O ponteiro q recebe o endereco do ponteiro p, i.e., q passa a apontar p/ p
printf("O endereco armazenado no ponteiro q eh q=%p\n",q);
printf("O valor armazenado na variavel indiretamente apontada por q eh **q=%f\n",**q);
printf("O endereco na memoria do ponteiro q eh &q=%p\n",&q);
}
```

Execução na tela do console:

```
O valor armazenado na variavel x eh x=3.400000
O endereco na memoria da variavel x eh &x=000000000065FE1C
O endereco armazenado no ponteiro p eh p=000000000065FE1C
O valor armazenado na variavel apontada por p eh *p=3.400000
O endereco na memoria do ponteiro p eh &p=000000000065FE10
O endereco armazenado no ponteiro q eh q=000000000065FE10
O valor armazenado na variavel indiretamente apontada por q eh **q=3.400000
O endereco na memoria do ponteiro q eh &q=000000000065FE08
```

Inicialização de ponteiros:

Depois que um ponteiro é declarado é necessário apontar o ponteiro para algum endereço válido. Se isto não for feito, o ponteiro apontará para um endereço desconhecido. Se tentarmos escrever um valor no endereço desconhecido, muito provavelmente o programa travará ou, dependendo da complexidade do programa, poderá se manifestar um comportamento bizarro sem qualquer aparente explicação lógica. Por convenção, um ponteiro que está apontando para nenhum endereço válido deve ter o endereço **NULL** nele armazenado. **NULL** é uma macro definida em **stddef.h** . Entretanto, somente o fato de um ponteiro apontar para o endereço **NULL** não torna seu uso seguro.

O fonte abaixo mostra como inicializar um ponteiro **char** com o endereço de uma *string*:

```
 /* inicializa ponteiro c/ uma string e imprime normal e inversa*/
#include <stdio.h>
#include <string.h>
#include <stddef.h>
void main(void)
{
  int i;
  char *p="Hello World!\n"; /* inicializa o ponteiro p com o endereço na memoria
                      do caractere 'H' da string  "Hello World!\n" */
  printf(p); // imprime normal
  for(i=strlen(p)-1; i>=0; i--) printf("%c",p[i]); // imprime inverso
}
```

Execução na tela do console:(notar que na *string* inversa o '\n' é impresso primeiro)

```
Hello World!

!dlroW olleH
```

Bugs decorrentes do mau uso de ponteiros:

Um bug decorrente do mau uso de um ponteiro não é trivial de ser solucionado. O ponteiro por si só não é um problema – o problema é que, cada vez que realizarmos uma operação errada usando o ponteiro ele pode estar "lendo de" ou "escrevendo em" algum lugar desconhecido da memória.

Na leitura de memória o que ocorre é o ponteiro ler informação inválida, o que usualmente leva o programa a imediatamente resultar em valores errados. Na escrita em memória o ponteiro escreve ou em endereços que armazenam o código binário que executa o programa ou em endereços que armazenam dados. Isto pode não causar bugs na execução imediata do programa, podendo o bug se manifestar mais tarde, em outra instância posterior do desenvolvimento do programa que envolva o endereço inválido armazenado no ponteiro. E isto complica muito a "caça ao bug" porque o programador é levado a crer que o bug é resultante do código recém escrito, quando, na realidade, o bug é decorrente de um ponteiro declarado no código que ele escreveu dois ou três dias atrás ...

Um exemplo clássico de erro com ponteiros é o ponteiro não-inicializado, conforme o código que segue:

```
/* Este programa estah errado. Nao o execute.*/
void main(void)
{
  int x, *p;
  x =10; //  valor 10 eh atribuido a variavel x
  *p = x; // atribui o valor de x a *p, i.e., atribui o valor de x ao endereco apontado pelo ponteiro p
}
```

O código acima atribui o valor 10 a alguma localização desconhecida na memória porque o ponteiro **p** nunca foi inicializado com um endereço. Mesmo tipo de problema surge quando o ponteiro é inicializado com um endereço, mas, por descuido do programador, o endereço de inicialização é inválido.

Ponteiros

Consideremos agora um outro erro clássico de uso de ponteiros, conforme mostrado no código fonte abaixo:

```
1  /* Este programa estah incorreto. Nao o execute. */
2  #include <stdio.h>
3  void main(void)
4  {
5    int x, *p;
6    x = 10;
7    p = x;
8    printf("%d", *p);
9  }
```

Note o ***warning*** emitido pelo compilador:
Line 7 Col 5 File:ErrInitPtr.c [Warning] assignment to 'int *' from 'int' makes pointer from integer without a cast [-Wint-conversion]

A função printf() não imprimirá na tela do console o valor 10 de x, mas sim Imprimirá algum valor desconhecido por causa da atribuição errada na linha 7 do código fonte acima:

p = x;

Esta declaração **p = x** faz com que o 10 armazenado em **x** seja o endereço armazenado em **p**, e, portanto, seja o endereço apontado pelo ponteiro **p**. Só que um endereço tão pequeno quanto 10 não tem qualquer validade no endereçamento usual da memória alocada por um programa, que normalmente usa endereços de 6 dígitos hexadecimais... Quando **printf("%d", *p)** é executado, o valor impresso na tela do console será sabe-se lá qual é o valor armazenado neste endereço 10 ... Para que o programa funcione, a linha 7 deve ser corrigida para:

p = &x;

que, agora sim, corretamente faz o ponteiro p armazenar o endereço na memória correspondente à variável x.

Funções

Já havíamos discutido em linhas gerais o conceito de funções, conforme vimos nos slides 52 a 57 do Cap I.1. Até o Cap I.1 não tínhamos conhecimento sobre o uso de ponteiros. Passamos então agora a revisitar funções, mas sem a limitação de evitar o uso de ponteiros que fomos obrigados a atender na explanação nos slides 52 a 57 do Cap I.1.

A forma geral de uma função:

A forma geral de uma função é:

```
tipo_de_dado_retorno  nome_da_função (declaração de argumentos)
{
        corpo da função
}
```

O **tipo_de_dado_retorno** especifica o tipo de dado de retorno que a função retornará quando no **corpo da função** houver uma declaração **return**.

O **tipo_de_dado_retorno** pode ser qualquer tipo de dado válido, incluindo tipos de dados definidos com **typedef** (veremos a declaração **typedef** adiante). Se nenhum tipo de dado de retorno é especificado, então, por definição, a função retorna um resultado **int**. Se a função não retorna valor, então o tipo de retorno a ser especificado é o tipo **void**.

A lista de **declaração de argumentos** é uma lista de tipos de variáveis e respectivos nomes, separados por vírgula, e que recebem os valores dos argumentos que são passados à função quando a função é chamada no âmbito do escopo da função chamadora. Uma função que não tiver argumentos em sua lista de argumentos é uma função de argumento do tipo **void** (vazio). Mesmo que a função tenha um argumento **void**, ainda assim os parênteses **()** são requeridos na chamada da função.

Funções

O código fonte https://www.fccdecastro.com.br/CursoC&C++/C/Dist2Ptos.c a seguir determina a distância euclidiana no espaço $\mathbb{R}^3$ dada por $D = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$ entre dois pontos $P1$ e $P2$ cujas coordenadas no espaço $\mathbb{R}^3$ são $P1 = [x_1 \quad y_1 \quad z_1]$ e $P2 = [x_2 \quad y_2 \quad z_2]$.

```c
/*****************************************************************************
* Este programa calcula a distancia entre dois pontos P1 e P2 do espaco R^3
*****************************************************************************/
/*****************************************************************************
* HEADERS:
*****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
/*****************************************************************************
* MACROS:
*****************************************************************************/
static double sqrarg;
#define SQR(a) ((sqrarg=(a)) == 0.0 ? 0.0 : sqrarg*sqrarg)
/*****************************************************************************
* FUNCTION PROTOTYPES:
*****************************************************************************/
double Dist2Points(double *U, double *V);
```

Funções

```
19  /*****************************************************************************
20  * main():
21  *****************************************************************************/
22  void main(void){
23
24  double P1[]={3.0, 4.0, 5.0}; // coordenadas do ponto P1 no espaco R^3
25  double P2[]={6.0, 7.0, 8.0}; // coordenadas do ponto P2 no espaco R^3
26
27  printf("Distancia entre P1 e P2 eh = %lf", Dist2Points(P1,P2) );
28  }
29  /*****************************************************************************
30  * FUNC: double Dist2Points(double *U, double *V)
31  *
32  * DESC: Return the distance between two points U and V in R^N space
33  *****************************************************************************/
34  double Dist2Points(double *U, double *V){
35  #define N 3 // R^N space dimension N
36  unsigned register i;
37  double acc=0.0;
38
39  for(i=0;i<N;i++)acc+=SQR(U[i]-V[i]);
40
41  return sqrt(acc);
42  }
```

Lembre: P1=&P1[0] e P2=&P2[0].

Note que os argumentos da função são ponteiros.

Execução na tela do console:

Distancia entre P1 e P2 eh = 5.196152

Retornando valores através do argumento–ponteiro de uma função:

O código fonte https://www.fccdecastro.com.br/CursoC&C++/C/MaxOfVect.c a seguir determina o máximo valor armazenado em um vetor float e retorna o índice do vetor em que ocorre o valor máximo. **O retorno do índice é efetuado através do argumento–ponteiro da função que determina o valor máximo.**

```
1  /*****************************************************************************
2  * Este programa determina o máximo elemento em um vetor float e o indice
3  * em que o maximo ocorre
4  *****************************************************************************/
5  /*****************************************************************************
6  * HEADERS:
7  *****************************************************************************/
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <math.h>
11 /*****************************************************************************
12 * PROGRAM DEFINITIONS:
13 *****************************************************************************/
14 #define TAMANHO_VETOR 4
15 /*****************************************************************************
16 * FUNCTION PROTOTYPES:
17 *****************************************************************************/
18 float MaxVetIndice(float *Vet, unsigned TamVet, unsigned *IndiceDoMax);
```

```
19  /*****************************************************************************
20  * main():
21  *****************************************************************************/
22  void main(void){
23  unsigned IndiceDoMaximo;
24  float ValorDoMaximo;
25
26  float Vetor[]={-3.0, 14.0, 25.0, 7,0}; // vetor float
27
28  ValorDoMaximo=MaxVetIndice(Vetor, TAMANHO_VETOR, &IndiceDoMaximo);
29
30  printf("Valor maximo %f no indice %u.\n", ValorDoMaximo, IndiceDoMaximo);
31  }
32
33  /*****************************************************************************
34  * FUNC:  float MaxVetIndice(float *Vet, unsigned TamVet, unsigned *IndiceDoMax)
35  *
36  * DESC: Retorna o valor maximo no vetor Vet e o indice do maximo via ponteiro IndiceDoMax.
37  *****************************************************************************/
38  float MaxVetIndice(float *Vet, unsigned TamVet, unsigned *IndiceDoMax)
39  {
40  unsigned register i;
41
42  float MaxVal=-3.4E38; /* menor valor float possivel */
43
44  for(i=0;i<TamVet;i++)if(Vet[i]>MaxVal){
45      MaxVal=Vet[i];
46      *IndiceDoMax=i;
47      }
48  return MaxVal;
49  }
```

Execução na tela do console: Valor maximo 25.000000 no indice 2.

Funções

Passando uma matriz float 2D como argumento de uma função através de um ponteiro:

O código fonte https://www.fccdecastro.com.br/CursoC&C++/C/FunMat.c a seguir passa uma matriz float 2D como argumento de uma função através de um ponteiro para a matriz 2D.

```c
/*****************************************************************************
* ESte programa passa uma matriz float 2D como argumento de uma funcao através
* de um ponteiro para a matriz 2D.
*****************************************************************************/
/*****************************************************************************
* HEADERS:
*****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
/*****************************************************************************
* PROGRAM DEFINITIONS:
*****************************************************************************/
#define NLIN 4
#define NCOL 4
/*****************************************************************************
* DATA TYPES:
*****************************************************************************/
typedef float Mat2D[NLIN][NCOL]; /* cria um novo tipo de dado denominado "Mat2D"
que representa uma matriz float de tamanho [NLIN][NCOL] */
/*****************************************************************************
* MACROS:
*****************************************************************************/
static double sqrarg;
#define SQR(a) ((sqrarg=(a)) == 0.0 ? 0.0 : sqrarg*sqrarg)
```

Funções

```
26  /*****************************************************************************
27  * FUNCTION PROTOTYPES:
28  *****************************************************************************/
29  void MatPrint(Mat2D *Mat, unsigned NLinhas, unsigned NColunas);
30  double MatDiagSqr(Mat2D *Mat, unsigned NLinhas, unsigned NColunas);
31  /*****************************************************************************
32  * main():
33  *****************************************************************************/
34  void main(void){
35
36
37  float Mat[NLIN][NCOL]={0.0, 0.1, 0.2, 0.3,
38                         1.0, 1.1, 1.2, 1.3,
39                         2.0, 2.1, 2.2, 2.3,
40                         3.0, 3.1, 3.2, 3.3,
41                         };
42  double Energia;
43  unsigned linha,coluna;
44
45  Mat2D *p;// declara o ponteiro p que aponta p/ o tipo de dado Mat2D
46
47  p=&Mat; // p recebe o endereço do primeiro elemento de float Mat[NLIN][NCOL]
48
49  printf("\nImprimindo Mat dentro do escopo da main() via ponteiro p=&Mat:\n");
50  /* Imprime na tela do console os elementos de Mat atraves do ponteiro p */
51      for(linha=0; linha<NLIN; linha++){
52          for(coluna=0;coluna<NCOL;coluna++){
53                  printf("%f\t",(*p)[linha][coluna]);
54          }
55          printf("\n");
56      }
```

```
57
58  printf("\nImprimindo Mat via funcao MatPrint() com argumento ponteiro p=&Mat:\n");
59  MatPrint(p,NLIN, NCOL);
60
61  printf("\nImprimindo Mat via funcao MatPrint() com argumento definido por cast de float Mat p/ (Mat2D *)Mat:\n");
62  MatPrint((Mat2D *)Mat,NLIN, NCOL);
63
64  /* eleva ao quadrado os elementos da maior diagonal e retorna a soma destes elementos: */
65  Energia=MatDiagSqr((Mat2D *)Mat, NLIN, NCOL);
66
67  printf("\nImprimindo Mat apos funcao MatDiagSqr(), cuja acao eh elevar os elementos da diagonal ao quadrado:\n");
68  MatPrint((Mat2D *)Mat,NLIN, NCOL);
69
70  printf("Energia da diagonal=%lf\n",Energia);
71
72  }
```

Note que MatDiagSqr() recebe o endereço do 1° elemento de Mat[][] como argumento. **Portanto todas as operações efetuadas no argumento dentro do escopo de MatDiagSqr() estarão simultaneamente sendo feitas na própria matriz Mat[][] dentro do escopo da main()**. Desta maneira, não há necessidade da função MatDiagSqr(), após efetuar operações em Mat[][], retornar Mat[][] p/ a main().

```
73
74  /*****************************************************************************
75  * FUNC: void MatPrint(Mat2D *Mat, unsigned NLinhas, unsigned NColunas)
76  *
77  * DESC: Imprime a matriz Mat na tela do console.
78  *****************************************************************************/
79  void MatPrint(Mat2D *Mat, unsigned NLinhas, unsigned NColunas)
80  {
81  unsigned register i,j;
82  double temp;
83
84  for(i=0; i<NLinhas; i++){
85          for(j=0;j<NColunas;j++){
86                  printf("%f\t",temp=(*Mat)[i][j]);
87          }
88          printf("\n");
89      }
90
91  }
```

Funções

```
92
93  /*****************************************************************************
94  * FUNC: double MatDiagSqr(Mat2D *Mat, unsigned NLinhas, unsigned NColunas)
95  *
96  * DESC: Eleva ao quadrado os elementos da maior diagonal da matriz Mat e retorna
97  *       a soma dos seus valores.
98  *****************************************************************************/
99  double MatDiagSqr(Mat2D *Mat, unsigned NLinhas, unsigned NColunas)
100 {
101 unsigned register k;
102 unsigned MaxIndice;
103 double Energia=0.0;
104
105 // Determina o maximo indice da maior diagonal:
106 if(NLinhas<=NColunas){
107     MaxIndice=NLinhas;
108     }else{
109     MaxIndice=NColunas;
110     }
111
112 // Eleva ao quadrado os elementos da maior diagonal:
113 for(k=0;k<MaxIndice;k++){
114     (*Mat)[k][k]=SQR((*Mat)[k][k]);
115     }
116
117 // Soma os elementos ao quadrado na maior diagonal:
118 for(k=0;k<MaxIndice;k++){
119         Energia+=(*Mat)[k][k];
120         }
121
122 return Energia;
123 }
```

Execução na tela do console:

```
Imprimindo Mat dentro do escopo da main() via ponteiro p=&Mat:
0.000000        0.100000        0.200000        0.300000
1.000000        1.100000        1.200000        1.300000
2.000000        2.100000        2.200000        2.300000
3.000000        3.100000        3.200000        3.300000

Imprimindo Mat via funcao MatPrint() com argumento ponteiro p=&Mat:
0.000000        0.100000        0.200000        0.300000
1.000000        1.100000        1.200000        1.300000
2.000000        2.100000        2.200000        2.300000
3.000000        3.100000        3.200000        3.300000

Imprimindo Mat via funcao MatPrint() com argumento definido por cast de float Mat p/ (Mat2D *)Mat:
0.000000        0.100000        0.200000        0.300000
1.000000        1.100000        1.200000        1.300000
2.000000        2.100000        2.200000        2.300000
3.000000        3.100000        3.200000        3.300000

Imprimindo Mat apos funcao MatDiagSqr(), cuja acao eh elevar os elementos da diagonal ao quadrado:
0.000000        0.100000        0.200000        0.300000
1.000000        1.210000        1.200000        1.300000
2.000000        2.100000        4.840000        2.300000
3.000000        3.100000        3.200000        10.889999
Energia da diagonal=16.940000
```

Funções

A organização funcional da memória – a pilha (*stack*) e a chamada de uma função:

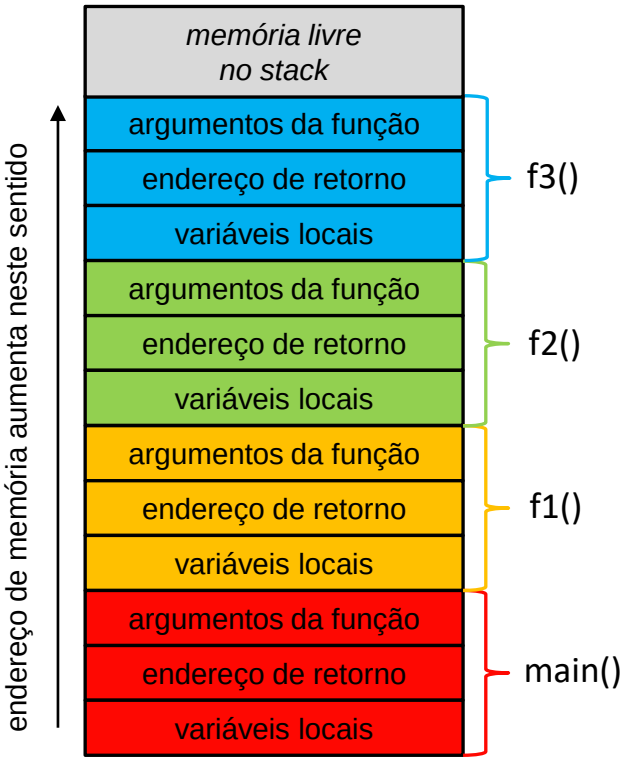
Toda a vez que uma função é chamada três tipos de dados são gravados em um segmento especial da memória:

(1) Os **argumentos** da função,
(2) as **variáveis locais** (= variáveis no âmbito do escopo da função) da função,
(3) o **endereço de retorno** ao qual o resultado da função chamada deve ser retornado.

Estes dados (1),(2) e (3) são armazenados em um segmento da memória denominado ***stack*** (pilha), conforme mostrado de maneira simplificada na figura abaixo. A designação "pilha" decorre do fato de que, à medida que uma função chama outra ao longo da execução de um programa, seus respectivos **argumentos**, **variáveis locais** e **endereço de retorno** são empilhados no ***stack***, semelhante à uma pilha de pratos empilhados.

À medida que as funções terminam de ser executadas, seus **argumentos** e **variáveis locais** são desempilhadas e o valor de retorno é entregue no **endereço de retorno**, que também é desempilhado do ***stack*** após o término da função chamada, quando então a função chamadora passa agora a ser executada. Esta maneira de empilhar e desempilhar dados é denominada **LIFO** (*Last In First Out*) – ver https://en.wikipedia.org/wiki/Stack-based_memory_allocation .

À medida que as funções são chamadas uma pela outra, a **memória livre** no *stack* diminui (ver figura) porque mais e mais argumentos, variáveis locais e endereços de retorno são empilhados no *stack*, podendo exceder a memória reservada para o *stack*, situação que é denominada ***stack overflow***. O programador deve ficar atento no sentido de evitar este tipo de situação, dado que a ocorrência do *stack overflow* travará a execução do programa. Hackers deliberadamente causam o *stack overflow* sobrescrevendo o endereços de retorno de uma função armazenada no *stack* com um endereço espúrio e assim desviando a execução do programa para uma função escrita pelo hacker e que dará ao hacker o controle da máquina por ele invadida (ver https://en.wikipedia.org/wiki/Stack_buffer_overflow).

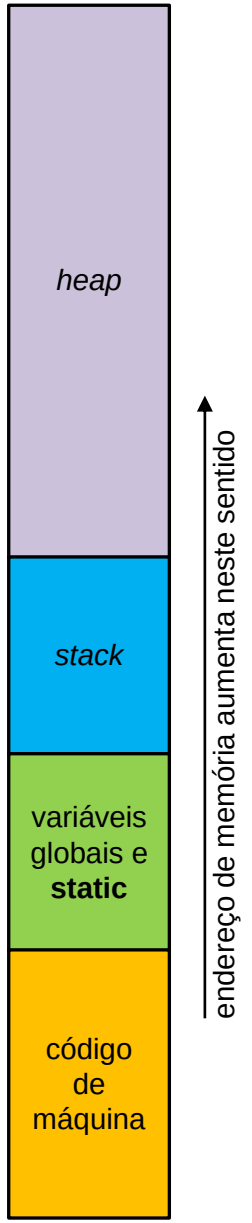


stack p/ a situação em que main() chama f1(), f1() chama f2() e f2() chama f3().

Funções

A organização funcional da memória – o *heap* como área de armazenamento dinâmico:

O *stack* discutido no slide anterior não é a única área de armazenamento de dados na memória. Quando se trabalha com vetores, matrizes ou estruturas que armazenam grande quantidade de dados, em que a memória precisa ser alocada dinamicamente através das funções **malloc()** ou **calloc()** (veremos alocação dinâmica no Cap II.3), não é aconselhável que os dados sejam armazenados no *stack*, mas sim na área de memória denominada ***heap***. A figura ao lado mostra a organização funcional da memória durante a execução de um programa e a localização funcional do *heap* na memória. O termo "organização funcional da memória" aqui significa que os endereços de memória são apenas relativos e apenas associados à funcionalidade da área de armazenamento, não correspondendo a endereços físicos.



O *heap* é mais flexível que o *stack* quanto à forma de alocar e liberar memória como também permite armazenar dados em uma área de memória bem maior que o *stack*. Conforme vimos no slide anterior, o *stack* é um LIFO, e, portanto, permite apenas alocação e liberação de memória ordenadamente no topo do *stack*. Já o *heap* é uma árvore binária (ver https://en.wikipedia.org/wiki/Heap_(data_structure) . Isto permite às funções de alocação dinâmica de memória **malloc()** e **calloc()** alocarem memória em qualquer lugar do *heap*, bem como permite à função de liberação de memória alocada **free()** liberar memória em qualquer lugar do *heap*. Isto significa que é possível ter "buracos" no meio do *heap* – memória não alocada cercada por memória alocada. Notar que o processo de alocação de memória via árvore binária é bem mais lento que o processo via LIFO. Portanto devemos evitar alocar memória com **malloc()** e liberar memória com **free()** muito mais de que algumas poucas vezes na execução de um programa, não só para evitar aumentar o tempo de execução como também para evitar fragmentação da memória.

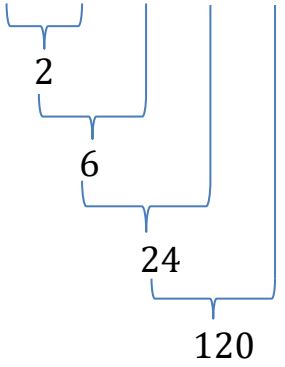
Quando **malloc()** e/ou **calloc()** são chamadas, elas alocam memória no *heap* localizando e retornando o primeiro bloco de memória grande o suficiente para satisfazer a solicitação. A memória alocada por **malloc()** e/ou **calloc()** é liberada por **free()**, e quando **free()** libera dois blocos de memória adjacentes, os blocos são mesclados para formar um único bloco. Isso permite **malloc()** e/ou **calloc()** atender às demandas futuras por grandes blocos de memória.

As operações de leitura e escrita (R/W) na memória alocada no *heap* são um pouco mais lentas que a operações R/W no *stack*. Isto acontece porque o *stack* é um LIFO enquanto que o *heap* é uma árvore binária.

Funções recursivas:

Uma função que chama a si própria de modo recorrente é denominada de função recursiva. A cada nova chamada recursiva a si própria as novas variáveis locais e os argumentos são alocados e gravados no *stack* e o código da função é executado com esses novos valores a partir do início do código da função. Uma nova chamada recursiva não faz uma nova cópia da função – somente os argumentos e as variáveis locais são novos. Quando cada chamada recursiva encerra e o valor de retorno é entregue no endereço de retorno, as antigas variáveis locais e argumentos são removidos do *stack* e a execução do programa recomeça no ponto de chamada da função (dado pelo endereço de retorno) dentro do escopo da função chamadora. Dado que os argumentos e as variáveis locais são armazenados no *stack* e que a cada nova chamada recursiva é criado uma cópia destes argumentos e variáveis no *stack*, então quando ocorrem muitas chamadas recursivas pode eventualmente acontecer que toda a memória reservada para o *stack* seja ocupada e excedida. Esta situação caracteriza o denominado *stack overflow*, já discutido no slide 41.

O código fonte https://www.fccdecastro.com.br/CursoC&C++/C/Fatorial.c no próximo slide exemplifica o uso da recursividade de uma função aplicada no âmbito do cálculo do fatorial de um número inteiro (ver https://pt.wikipedia.org/wiki/Fatorial) . Por exemplo, o fatorial de 5 é dado por $5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$.



Funções

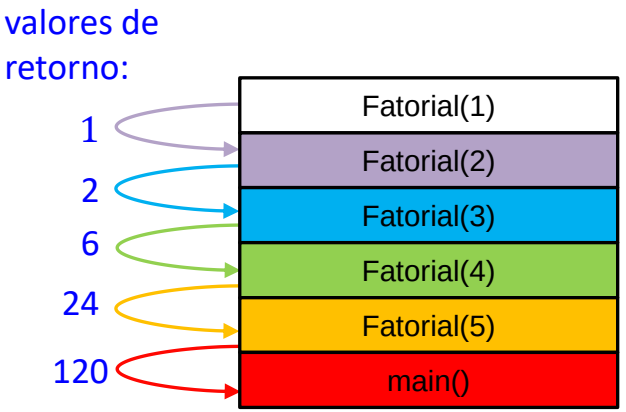
```c
/*****************************************************************************
* ESte programa determina recursivamente o fatorial de um numero inteiro
*****************************************************************************/
/*****************************************************************************
* HEADERS:
*****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
/*****************************************************************************
* FUNCTION PROTOTYPES:
*****************************************************************************/
unsigned long int Fatorial(unsigned long int n);
/*****************************************************************************
* main():
*****************************************************************************/
void main(void)
{
unsigned long int n,fat;

printf("Digite um numero: ");
scanf("%ld",&n);
fat=Fatorial(n);
printf("O fatorial de %ld eh %ld.\n", n,fat);
}
```

Funções

```
25  /*************************************************************************
26  * FUNC: unsigned long int Fatorial(unsigned long int n)
27  *
28  * DESC: Calcula o fatorial do argumento n.
29  *************************************************************************/
30  unsigned long int Fatorial(unsigned long int n)
31  {
32  unsigned long int resposta;
33  if(n==1||n==0)return(1);
34  resposta = Fatorial(n-1)*n;
35  printf("%ld\n",resposta);  // print resultado parcial
36  return(resposta);
37  }
```

Execução na tela do console:

```
Digite um numero: 5
2
6
24
120
O fatorial de 5 eh 120.
```



stack p/ o cálculo de $5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$

Alocação dinâmica de memória

Conforme discutimos no slide 42, quando um programa usa vetores, matrizes ou estruturas que armazenam grande quantidade de dados, a memória deve ser alocada dinamicamente no *heap* através das funções **malloc()** ou **calloc().**

Quando **malloc()** e/ou **calloc()** são chamadas, elas alocam memória no *heap* localizando e retornando o endereço do início do primeiro bloco de memória grande o suficiente para satisfazer o tamanho da solicitação de alocação de memória. A memória alocada por **malloc()** e/ou **calloc()** é liberada por **free()**, como também pode ser redimensionada através da função **realloc()**.

Uma vez alocada memória c/ **malloc()/calloc()**, não é aconselhável liberar memória c/ **free()** e chamar novamente **malloc()/calloc()** (ou redimensionar a memória com **realloc()**) mais de que algumas poucas vezes, não só p/ evitar evitar fragmentação (ver https://pt.wikipedia.org/wiki/Aloca%C3%A7%C3%A3o_din%C3%A2mica_de_mem%C3%B3ria_em_C , https://en.wikipedia.org/wiki/C_dynamic_memory_allocation e https://en.wikipedia.org/wiki/Fragmentation_(computing)) como também p/ evitar aumentar o tempo de execução.

Existem, portanto, quatro funções básicas para gerenciamento dinâmico da memória do *heap*:

malloc() aloca um especificado tamanho de memória em bytes no *heap*.
calloc() aloca um especificado tamanho de memória em bytes no *heap*, e armazena o valor zero em cada byte alocado.
free() libera a memória alocada por **malloc() , calloc()** ou **realloc()** p/ ser reutilizada.
realloc() encolhe ou expande o bloco de memória alocado por **malloc()** ou c**alloc()**, e se necessário move-o para caber no espaço disponível no *heap*.

As declarações destas funções de gerenciamento de memória do *heap* estão disponíveis na seção "2.13.3 Memory Functions" na página 115 de https://www.fccdecastro.com.br/CursoC&C++/CLibraryReferenceGuide - Huss.pdf . Os protótipos destas funções estão no *header* stdlid.h, e, especificamente, são conforme segue:

void *malloc (unsigned long NumElements*NumBytes);

malloc() retorna um endereço do tipo **void**, significando que o tipo de endereço pode ser alterado através de um *cast* e ser armazenado em um ponteiro de qualquer tipo, de modo ao ponteiro apontar para o que se desejar: número, vetor, matriz, estrutura, etc... **NumBytes** é calculado através do operador **sizeof()**, operador que retorna o número de bytes ocupado na memória pelo tipo de dado do *cast*. Por exemplo, a declaração **p=(double *)malloc(4*sizeof(double))** aloca 32 bytes no *heap* no endereço apontado pelo ponteiro **p** do tipo double. Isto acontece porque **sizeof()** retorna o valor **8** (cada **double** ocupa **8 bytes)** e porque **NumElements=4** é o número de elementos de tamanho N**umBytes=8** que são alocados no *heap*. Se não houver memória suficiente, **malloc()** retorna um ponteiro **NULL**.

Alocação dinâmica de memória

void *calloc(unsigned NumElements, unsigned long NumBytes);

Assim como **malloc(), calloc()** retorna um endereço do tipo **void**, significando que o tipo de endereço pode ser alterado através de um *cast* e ser armazenado em um ponteiro de qualquer tipo, de modo ao ponteiro apontar para o que se desejar: número, vetor, matriz, estrutura, etc... A diferença é que **calloc()** armazena o valor zero em cada byte de memória alocado no *heap*, enquanto **malloc()** não altera o valor previamente armazenado em cada byte alocado. **NumElements** é o número de elementos de tamanho **NumBytes** que são alocados no *heap*. **NumBytes** é calculado pelo operador **sizeof()**, operador que retorna o número de bytes ocupado na memória pelo tipo de dado do *cast* conforme discutido *no slide* anterior.

void free(void *p);

free() libera memória previamente alocada por **malloc()** e/ou **calloc()**, memória que é apontada pelo ponteiro **p**. Por exemplo, a declaração **p= (float *)calloc(5, sizeof(float))** aloca 20 bytes no *heap* e o endereço do primeiro elemento do segmento de 20 bytes é armazenado no ponteiro **p** do tipo **float**. Para liberar estes 20 bytes alocados no *heap* basta declarar **free(p)**. **Nota: NUNCA** usar **free()** p/ liberar um bloco de memória apontada por um ponteiro que armazene um endereço que não tenha sido gerado por **malloc()** ou **calloc()**, ou a memória usada no bloco será perdida.

void *realloc(void *p, unsigned long Tamanho);

realloc() altera o tamanho da memória no *heap* apontada pelo ponteiro **p** para o tamanho especificado por **Tamanho**.

Por exemplo, a declaração **p= (float *)malloc(5*sizeof(float))** aloca 20 bytes no *heap* e o endereço do primeiro elemento do segmento de 20 bytes é armazenado no ponteiro **p** do tipo **float**.

Se, por exemplo, quisermos alterar o tamanho da memória alocada de modo ao *heap* armazenar 50 elementos **float** ao invés dos 5 elementos previamente alocados basta declararmos **p=(float *)realloc((float *)p, 50*sizeof(float))** ou simplesmente **p=realloc(p, 50*sizeof(float))**, onde **p** é um ponteiro previamente declarado como do tipo **float**.

Se não houver memória suficiente no *heap*, **realloc()** retorna um ponteiro **NULL** e libera (executa **free()**) o bloco original.

Alocação dinâmica de memória

O código fonte https://www.fccdecastro.com.br/CursoC&C++/C/VectAlloc.c a seguir utiliza alocação dinâmica em uma aplicação em que a memória para o domínio e a imagem de uma função quadrática é dinamicamente alocada no *heap* através de **malloc()**:

```c
/*****************************************************************************
* Calcula NPtos de y(x)=A*x^2+B*x+C no dominio xMinimo <=x <= xMaximo
* e imprime x & y(x) na tela do console. Alocacao dinamica de memoria eh usada
* para armazenar na memoria os valores do dominio x e da imagem y(x).
*****************************************************************************/
/*****************************************************************************
* HEADERS:
*****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
/*****************************************************************************
* MACROS:
*****************************************************************************/
static double sqrarg;
#define SQR(a) ((sqrarg=(a)) == 0.0 ? 0.0 : sqrarg*sqrarg)
/*****************************************************************************
* FUNCTION PROTOTYPES:
*****************************************************************************/
void Quadratic(double A,double B,double C,double xMin,double xMax,unsigned NPtos,double *x,double *y);
void PrintUV(double *U, double *V, unsigned NumElems);
```

Alocação dinâmica de memória

```
22  /*****************************************************************************
23  * main():
24  *****************************************************************************/
25  void main(void){
26
27  double A=1.0, B=-10.0, C= 20; // y(x)= A*x^2+B*x+C
28  double xMinimo=0.0, xMaximo = 10.0; // intervalo do dominio x
29  unsigned NumPontos= 10; // numero de pontos do dominio
30  double *x,*y;
31
32  /* aloca memoria para o dominio x */
33  x= (double *)malloc(NumPontos*(sizeof(double)));
34  if(!x){puts("Nao consigo alocar memoria para o dominio x!");exit(1);}
35
36  /* aloca memoria para a imagem y */
37  y= (double *)malloc(NumPontos*(sizeof(double)));
38  if(!y){puts("Nao consigo alocar memoria para a imagem y!");exit(1);}
39
40  /* Calcula NumPontos de y(x)=A*x^2+B*x+C no dominio xMinimo <= x <= xMaximo */
41  Quadratic(A,B,C,xMinimo,xMaximo,NumPontos,x,y);
42
43  /* Imprime vetores x[] e y[] de NumPontos elementos (=componentes) */
44  printf("Dominio:\tImagem:\n");
45  PrintUV(x,y,NumPontos);
46  }
```

Alocação dinâmica de memória

```
47  /*****************************************************************************
48  * FUNC: void Quadratic(double A,double B,double C,double xMin,double xMax,unsigned NPtos,double *x,doub
49  *
50  * DESC: Calcula NPtos de y(x)=A*x^2+B*x+C no dominio xMin <= x <= xMax.
51  *****************************************************************************/
52  void Quadratic(double A,double B,double C,double xMin,double xMax,unsigned NPtos,double *x,double *y)
53  {
54  double Delta_x;
55  unsigned register n;
56
57  Delta_x=(xMax-xMin)/(NPtos-1); // intervalo entre dois pontos
58
59  for(n=0;n<NPtos;n++){
60      x[n]=xMin+n*Delta_x;
61      y[n]=A*SQR(x[n])+B*x[n]+C;
62  }
63  }
64  /*****************************************************************************
65  * FUNC: void PrintUV(double *U, double *V, unsigned NumElems)
66  *
67  * DESC: Imprime vetores U[] e V[] de NumElems elementos (=componentes)
68  *****************************************************************************/
69  void PrintUV(double *U, double *V, unsigned NumElems)
70  {
71  unsigned register n;
72  for(n=0;n<NumElems;n++)printf("%lf\t%lf\n",U[n],V[n]);
73  }
```

Execução na tela do console:

```
Dominio:        Imagem:
0.000000        20.000000
1.111111        10.123457
2.222222        2.716049
3.333333        -2.222222
4.444444        -4.691358
5.555556        -4.691358
6.666667        -2.222222
7.777778        2.716049
8.888889        10.123457
10.000000       20.000000
```

Alocação dinâmica de memória

O código fonte https://www.fccdecastro.com.br/CursoC&C++/C/Mat2DAlloc.c a seguir aloca e libera dinamicamente memória para uma matriz bidimensional Mat[][] e executa operações com os elementos da matriz:

```c
/*****************************************************************************
* Aloca memoria dinamicamente para uma matriz 2D Mat, atribui valores aos
* elementos de Mat e imprime Mat na tela do console. Dai eleva ao quadrado
* os elementos de Mat e imprime Mat. E finalmente libera a memoria alocada p/
* Mat e, a seguir, imprime o que sobrou de Mat apos a liberacao de memoria.
*****************************************************************************/
/*****************************************************************************
* HEADERS:
*****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
/*****************************************************************************
* MACROS:
*****************************************************************************/
static double sqrarg;
#define SQR(a) ((sqrarg=(a)) == 0.0 ? 0.0 : sqrarg*sqrarg)
/*****************************************************************************
* PROGRAM DEFINITIONS:
*****************************************************************************/
#define NLINS 3 // numero de linhas da matriz Mat
#define NCOLS 4 // numero de colunas da matriz Mat
/*****************************************************************************
* FUNCTION PROTOTYPES:
*****************************************************************************/
float **FloatMatCAlloc(unsigned NLinhas,unsigned NColunas);
void FloatMatFree(float **Mat);
void FloatMatPrint(float **Mat,unsigned NLinhas,unsigned NColunas);
```

Alocação dinâmica de memória

```
/*****************************************************************************
* main():
*****************************************************************************/
void main(void){
float **Mat;
unsigned register Linha, Coluna;

/* aloca memoria p/ a matriz Mat */
Mat=FloatMatCAlloc(NLINS,NCOLS);

/* atribui valores aos elementos da matriz Mat */
for(Linha=0;Linha<NLINS;Linha++){
    for(Coluna=0;Coluna<NCOLS;Coluna++){
    Mat[Linha][Coluna]=(NCOLS*Linha)+Coluna; // ver slides 19,20 e 21 do Cap II.1
    }
}
/* imprime Mat */
FloatMatPrint(Mat,NLINS,NCOLS);

/* eleva ao quadrado os elementos da matriz Mat */
for(Linha=0;Linha<NLINS;Linha++){
    for(Coluna=0;Coluna<NCOLS;Coluna++){
    Mat[Linha][Coluna]=SQR(Mat[Linha][Coluna]);
    }
}
/* imprime Mat com os elementos elevados ao quadrado */
printf("\n");
FloatMatPrint(Mat,NLINS,NCOLS);

/* libera memoria alocada p/ Mat */
FloatMatFree(Mat);
```

Alocação dinâmica de memória

```
60
61  /* tenta imprimir o que sobrou de Mat apos liberar memoria */
62  printf("\n");
63  FloatMatPrint(Mat,NLINS,NCOLS);
64  }
```

Alocação dinâmica de memória

```
65  /*************************************************************************
66  * FUNC: float **FloatMatCAlloc(unsigned NLinhas,unsigned NColunas)
67  *
68  * DESC: Aloca memoria para a matriz float Mat[NLinhas][NColunas].
69  *************************************************************************/
70  float **FloatMatCAlloc(unsigned NLinhas,unsigned NColunas)
71  {
72  float  *V; // vetor V[]
73  float  **Mat; // matriz Mat[][]
74  register unsigned i;
75
76  /* aloca memoria para todos os elementos da matriz Mat[NLinhas][NColunas]
77  na forma de um vetor V[NLinhasxNColunas] - ver slides 19,20 e 21 do Cap II.1 */
78  V = (float *)calloc((unsigned long)(NLinhas*NColunas),sizeof(float));
79  if(!V){puts("Nao consigo alocar memoria para o vetor V[]!");exit(1);}
80
81  /* aloca memoria para o vetor de ponteiros Mat[NLinhas], vetor que armazenarah
82  os endereços do primeiro elemento de cada linha da matriz Mat[NLinhas][NColunas] */
83  Mat=(float  **)calloc((unsigned long)NLinhas,sizeof(float *));
84  if(!Mat){puts("Nao consigo alocar memoria para o vetor Mat[]!");exit(1);}
85
86  /* atribui a cada elemento do vetor de ponteiros Mat[NLinhas] o endereço do primeiro
87  elemento de cada linha da matriz Mat[NLinhas][NColunas] */
88  for(i=0;i<NLinhas;i++){
89  Mat[i]=(float *)&V[i*NColunas];
90  }
91
92  /* retorna o endereço do primeiro elemento do vetor Mat[NLinhas] e que correponde
93  ao endereco do primeiro elemento da primeira linha da matriz Mat[NLinhas][NColunas] */
94  return Mat;
95  }
```

Alocação dinâmica de memória

```
96  /*****************************************************************************
97  * FUNC: void FloatMatFree(float **Mat)
98  *
99  * DESC: libera memoria alocada p/ a matrix float Mat[][].
100 *****************************************************************************/
101 void FloatMatFree(float **Mat)
102 {
103 free(Mat[0]); /* libera vetor float Mat[0]=&V[0]=V alocado atraves de
104 V = (float *)calloc((unsigned long)(NLinhas*NColunas),sizeof(float)) em
105  FloatMatCAlloc() */
106 free(Mat); /* libera vetor de ponteiros Mat alocado atraves de
107 Mat=(float  **)calloc((unsigned long)NLinhas,sizeof(float *)) em
108 em FloatMatCAlloc() */
109 }
110 /*****************************************************************************
111 * FUNC: void FloatMatPrint(float **Mat,unsigned NLinhas,unsigned NColunas)
112 *
113 * DESC: Imprime a matriz float Mat[NLinhas][NColunas] na tela do console.
114 *****************************************************************************/
115 void FloatMatPrint(float **Mat,unsigned NLinhas,unsigned NColunas)
116 {
117 register unsigned i,j;
118
119 for(i=0;i<NLinhas;i++){
120     for(j=0;j<NColunas;j++){
121     printf("%8.6g\t",Mat[i][j]);
122     }
123 printf("\n");
124 }
125 }
```

Alocação dinâmica de memória

Execução na tela do console:

```
     0          1          2          3
     4          5          6          7
     8          9         10         11

     0          1          4          9
    16         25         36         49
    64         81        100        121

1.84594e-038          0 1.84977e-038          0
    16         25         36         49
    64         81        100        121
```

Valores armazenados nos elementos de Mat.

Valores armazenados nos elementos de Mat após elevar os mesmos ao quadrado.

Note os valores espúrios após a liberação de memória.

Os argumentos int argc, char *argv[] e char *env[] da função main()

Dado que configuramos o DevC++ para gerar executáveis do tipo “Console Application” (ver slides 9 e 22 do Cap I.1) é crucial que possamos passar informações através da linha de comando de um programa quando ele é executado na janela do console. O método consiste em passar estas informações para a função **main()** por meio dos **argumentos de linha de comando**. Um argumento de linha de comando é um argumento da função **main()** e ele é especificado a seguir do nome do programa digitado na linha de comando da tela do console.

Existem três argumentos para a função **main()**. Os dois primeiros, **int argc** e **char *argv[]** , são usados para receber argumentos da linha de comando. O terceiro, **char *env[]**, é usado para ler os parâmetros do ambiente do sistema operacional que estão ativos quando o programa começa a execução. Estes são os únicos argumentos que **main()** pode ter.

O argumento **argc** armazena o número de argumentos digitados na linha de comando. O argumento **argc** sempre armazenará no mínimo o valor um, já que o próprio nome do programa é considerado como o primeiro argumento. O argumento ***argv[]** é um vetor de ponteiros em que cada elemento aponta para a respectiva *string* digitada na linha de comando**. Importante notar que todos os argumentos de linha de comando são *strings*** - qualquer número digitado como parâmetro é interpretado como uma *string* e necessita ser convertido para seu correspondente valor numérico através das funções **atof()** , **atoi()** ou **atol()** – ver ver seção "2.13.2 String Functions" na página 112 de https://www.fccdecastro.com.br/CursoC&C++/CLibraryReferenceGuide - Huss.pdf . Por exemplo:

```
/* cmdline.c imprime a linha de comando na tela do console */
#include<stdio.h>
void main(int argc, char *argv[]){
unsigned i;
for(i=0;i<argc;i++) puts(argv[i]);
}
```

Linha de comando

Execução na tela do console:

```
C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\ExplosNoPost>cmdline alfa 0.5 1.4 7
cmdline  <-- argv[0] – nome do programa
alfa     <-- argv[1]
0.5      <-- argv[2]
1.4      <-- argv[3]
7        <-- argv[4]
```

Os argumentos int argc, char *argv[] e char *env[] da função main()

```
/* envcmdline.c imprime as variaveis de ambiente do sistema operacional */
#include<stdio.h>
void main(int argc, char *argv[], char *env[]){
unsigned i;
for(i=0;env[i];i++) puts(env[i]);
}
```

Execução na tela do console:

```
C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\ExplosNoPost>envcmdline
ALLUSERSPROFILE=C:\ProgramData
APPDATA=C:\Users\fccde\AppData\Roaming
CommonProgramFiles=C:\Program Files\Common Files
CommonProgramFiles(x86)=C:\Program Files (x86)\Common Files
CommonProgramW6432=C:\Program Files\Common Files
COMPUTERNAME=SHAMAN
ComSpec=C:\WINDOWS\system32\cmd.exe
DriverData=C:\Windows\System32\Drivers\DriverData
EFC_19320=1
FPS_BROWSER_APP_PROFILE_STRING=Internet Explorer
FPS_BROWSER_USER_PROFILE_STRING=Default
HOMEDRIVE=C:
HOMEPATH=\Users\fccde
IGCCSVC_DB=AQBKLNCMnd8BF
LOCALAPPDATA=C:\Users\fccde\AppData\Local
LOGONSERVER=\\ SHAMAN
NUMBER_OF_PROCESSORS=16
OneDrive=C:\Users\fccde\OneDrive
OS=Windows_NT
Path=C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\WINDOWS\System32\WindowsPowerShell\
.
.
.
```

Os argumentos int argc, char *argv[] e char *env[] da função main()

O código fonte https://www.fccdecastro.com.br/CursoC&C++/C/argcv.c a seguir mostra na tela do console os argumentos digitados na linha de comando e converte p/ double os argumentos numéricos:

```
1  /* mostra os argumentos digitados e converte p/ double os argumentos numericos */
2  #include <stdio.h>
3  #include<stdlib.h>
4  #include <string.h>
5  #include <ctype.h>
6  void main(int argc,char *argv[])
7  {
8  int i,j;
9  char flag='v';/*'v' indica que a string eh um numero */
10
11 printf("argc=%d\n",argc);
12 for(i=0;i<argc;i++){
13 printf("argv[%d]=%s\n",i,argv[i]); /* imprime a i-esima string-argumento
14                                        como uma string */
15 }
16
17 printf("\n Convertendo para double os argumentos numericos:\n\n");
```

Os argumentos int argc, char *argv[] e char *env[] da função main()

```
18
19  for(i=0;i<argc;i++){
20      flag='v'; // assume inicialmente que a i-esima string-argumento eh um numero
21
22  /* verifica se os caracteres de cada i-esima string-argumento nao sao os numeros 0 a 9,
23  nem sao o '.' do ponto-flutuante, nem sao o expoente 'e' da notacao cientifica e nem sao
24  o sinal '-' */
25      for(j=0;j<strlen(argv[i]);j++) {
26          if((!isdigit(argv[i][j]))&&(argv[i][j]!='.')&&(tolower(argv[i][j])!='e')&&(argv[i][j]!='-')){
27          flag='f'; // sinaliza que os carateres da i-esima string nao caracterizam um numero
28          break;
29          }
30      }
31      if(flag=='v'){// 'v' sinaliza que os carateres da i-esima string caracterizam um numero
32      double x;  /* x soh existe neste bloco */
33      x=atof(argv[i]); // converte a i-esima string p/ double
34      printf("argv[%d]=%lf\n",i,x); /* imprime argumentos da i-esima s como double */
35      }
36  }
37  }
```

Os argumentos int argc, char *argv[] e char *env[] da função main()

Execução na tela do console:

C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para proc de sinais\C\Explos>argcv alfa beta -0.1E-2 gama 0.2

Linha de comando

argc=6
argv[0]=argcv
argv[1]=alfa
argv[2]=beta
argv[3]=-0.1E-2
argv[4]=gama
argv[5]=0.2

Convertendo para double os argumentos numericos:

argv[3]=-0.001000
argv[5]=0.200000

I/O de disco

Um ***stream*** é uma entidade lógica que representa um arquivo de disco ou um dispositivo de I/O, como, por exemplo, uma unidade de fita magnética, uma porta serial, etc... (ver https://en.wikipedia.org/wiki/Input/output) . O *stream* serve como interface no âmbito do processo de leitura e/ou escrita de dados neste arquivo de disco ou neste dispositivo de I/O. Todas as funções de I/O da biblioteca padrão do C operam em um *stream* de dados.

Embora um *stream* lide com diferentes meios/fontes de dados, como, por exemplo, um *socket* para estabelecer comunicação com outra máquina através de uma rede de dados, neste capítulo de nosso estudo os *streams* estarão associados a meios/fontes de dados armazenados em arquivos gravados em disco. Os *streams* de disco podem ser divididos em *streams* de texto e *streams* de dados binários (ver seção "2.12.2 Streams and Files" na página 93 de https://www.fccdecastro.com.br/CursoC&C++/CLibraryReferenceGuide - Huss.pdf).

Neste contexto, um *stream* é uma interface lógica que auxilia o programador no I/O de disco, evitando que o programador tenha que se envolver com a maneira em que os dados são gerenciados a nível de hardware. Em C um *stream* de disco é basicamente um *buffer* de armazenamento intermediário na memória que é acessado através de um ponteiro de arquivo ***p** do tipo **FILE**, que é definido no *header* **stdio.h**. O *buffer* apontado pelo ponteiro de arquivo ***p** do tipo **FILE** é a área intermediária para transferência de dados entre o arquivo no disco e o programa (ver "type FILE" na página 232 de https://www.fccdecastro.com.br/CursoC&C++/TheStandardCLibrary%20-%20Plaucer.pdf). Mais especificamente, o tipo de dados **FILE** não é apenas um *buffer*, mas sim é um tipo de dado definido por uma declaração **typedef** em **stdio.h** que consiste em uma estrutura com vários membros, cada membro desempenhando uma função no processo de I/O de disco, sendo um dos membros da estrutura **FILE** o referido *buffer* (veremos estruturas adiante):

```
struct _iobuf {
   char *_ptr;
   int _cnt;
   char *_base;
   int _flag;
   int _file;
   int _charbuf;
   int _bufsiz;
   char *_tmpfname;
 };
 typedef struct _iobuf FILE;
```

I/O de disco

Para efetuar operações de I/O de dados em um arquivo de disco, o programa deve preliminarmente declarar um ponteiro de arquivo ***p** do tipo **FILE** para o arquivo de disco para o qual se deseja efetuar operações de I/O de dados. Isto deve ser feito para cada arquivo de disco utilizado no programa e cada ponteiro de arquivo do tipo **FILE** declarado deve ter um nome distinto dos demais ponteiros **FILE** declarados para os demais arquivos. Uma declaração típica para um ponteiro de arquivo do tipo **FILE** é

FILE *p;

onde **p** é o nome que escolhemos para o ponteiro de arquivo (podia ser qualquer outro nome).

Funções mais comuns para I/O de disco

Função	Operação
fopen()	**Abre um *stream***
fclose()	**Fecha um *stream***
putc()	**Escreve um caractere em um *stream***
getc()	**Lê um caractere de um *stream***
fseek()	**Procura em um *stream* um byte especificado**
fprintf()	**É para um *stream* aquilo que printf() é para o console**
fscanf()	**É para um *stream* aquilo que scanf() é para o console**
feof()	**Retorna verdadeiro se o fim do arquivo é encontrado**
ferror()	**Retorna verdadeiro se ocorreu um erro**
fread()	**Lê um bloco de dados de um *stream***
fwrite()	**Escreve um bloco de dados em um *stream***
rewind()	**Reposiciona no começo do arquivo o ponteiro de posição do arquivo**
remove()	**Apaga um arquivo**

Ver seção "2.12.3 File Functions“ na página 94 de https://www.fccdecastro.com.br/CursoC&C++/CLibraryReferenceGuide%20-%20Huss.pdf

Abrindo um Arquivo:

A função **fopen()** serve a dois propósitos: (1) cria um *stream* para uso em I/O de disco e conecta o nome de um arquivo de disco com o *stream* e (2) retorna o ponteiro de arquivo do tipo **FILE** associado ao arquivo de disco. A funçao **fopen()** apresenta este protótipo:

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

onde **modo** é uma *string* que identifica se o arquivo é para dados de texto ou para dados binários e se a operação de abertura de arquivo é para leitura ou escrita no arquivo, conforme tabela no próximo slide.

O **nome_de_arquivo** é uma *string* que armazena um nome de arquivo válido para o sistema operacional, podendo ser incluído a especificação de caminho (**PATH**) no sistema de pastas do disco.

fopen() retorna um ponteiro de arquivo do tipo **FILE** **que não deve ter o valor alterado pelo programa**. Se um erro ocorre na abertura de um arquivo, **fopen()** retorna um ponteiro **NULL**.

Sempre devemos testar se o valor de retorno de **fopen()** é **NULL**, e, em caso positivo, o programa deve avisar que o arquivo não pode ser aberto ou executar o tratamento da situação de impossibilidade de abertura de arquivo.

Se queremos abrir um arquivo para escrita de texto e/ou números com o nome “notas.txt” podemos, por exemplo, usar o seguinte trecho de código:

```
FILE  *out;
if((out=fopen(“notas.txt","wt"))==NULL){
  puts(“Nao posso abrir o arquivo!");
  exit(1);
}
```

onde a macro **NULL** é definida no *header* **stdio.h**. O teste do valor de retorno **NULL** detecta qualquer erro na abertura do arquivo, como um arquivo protegido contra escrita ou disco cheio, evitando assim escrever no arquivo nestas situações. A função **exit()** termina imediatamente o programa, não importando de onde **exit()** é chamada, e retorna o valor dentro do argumento **()** para o sistema operacional. **exit(1)** ou qualquer valor diferente de zero dentro de **()** significa que ocorreu um erro, enquanto que **exit(0)** significa que não houve erro. O protótipo de **exit()** é definido no *header* stdlib.h.

I/O de disco

Conforme mostra a tabela abaixo, arquivos podem ser abertos no modo texto ou no modo binário. Em um arquivo texto os dados são armazenados na forma de um conjunto de caracteres ASCII, e são organizados em linhas, cada linha terminando com um caractere de nova linha ('\n'). Em um arquivo binário os dados são armazenados como uma cópia byte a byte da memória que armazena os dados a serem gravados no arquivo binário. Em geral um arquivo binário é usado para armazenar uma imagem (cópia byte a byte) de dados numéricos armazenados no *heap*, como vetores e matrizes.

---

strings válidas para modo.

---

modo	**ação**
"r"	**Abre um arquivo para leitura**
"w"	**Cria um arqivo para escrita**
"a"	**Acrescenta dados para um arquivo existente**
"rb"	**Abre um arquivo binário para leitura**
"wb"	**Cria um arquivo binário para escrita**
"ab"	**Acrescenta dados a um arquivo binário existente**
"r+"	**Abre um arquivo para leitura/escrita**
"w+"	**Cria um arquivo para leitura/escrita**
"a+"	**Acrescenta dados ou cria um arquivo para leitura/escrita**
"r+b"	**Abre um arquivo binário para leitura/escrita**
"w+b"	**Cria um arquivo binário para leitura/escrita**
"a+b"	**Acrescenta ou cria um arquivo binário para leitura/escrita**
"rt"	**Abre um arquivo texto para leitura**
"wt"	**Cria um arquivo texto para escrita**
"at"	**Acrescenta dados a um arquivo texto**
"r+t"	**Abre um arquivo-texto para leitura/escrita**
"w+t"	**Cria um arquivo texto para leitura/escrita**
"a+t"	**Acrescenta dados ou cria um arquivo texto para leitura/escrita**

---

ver seção "2.12.3.7 fopen“ nas páginas 95 e 96 de https://www.fccdecastro.com.br/CursoC&C++/CLibraryReferenceGuide%20-%20Huss.pdf

Nota: Dependendo do compilador, alguns modos da tabela acima não são implementados.

I/O de disco

Ao chamar **fopen()** para abrir um arquivo no modo de escrita em arquivo, qualquer arquivo preexistente com o mesmo nome será apagado e o novo arquivo será iniciado e substituirá o arquivo preexistente. Se não existir arquivo preexistente com o mesmo nome, então será criado um.

Se quisermos acrescentar dados ao final de um arquivo preexistente, devemos chamar **fopen()** no modo **"a".**

Para abrir um arquivo para operações de leitura é necessário que o arquivo já exista. Se ele não existir, **fopen()** retorna **NULL**.

Se um arquivo é aberto com **fopen()** para escrita/leitura, as operações feitas não apagarão o arquivo se ele for preexistente. No entanto, se o arquivo não existir, o arquivo será criado.

Escrevendo um caractere com putc():

A função **putc()** é usada para escrever caracteres em um *stream* que foi previamente aberto para escrita por **fopen()**. A função **putc()** apresenta o seguinte protótipo:

int putc(int ch, FILE *fp);

onde **fp** é um ponteiro de arquivo do tipo **FILE** retornado por **fopen()** e **ch** é o caractere a ser escrito. Por razoes históricas, **ch** é declarado como **int** (2 bytes = 16 bits), mas somente o byte de ordem mais baixa é escrito no arquivo.

Se uma operação de escrita com **putc()** terminar em sucesso, **putc()** retorna o caractere escrito. Em caso de falha, um **EOF** (*End Of File*) é retornado. **EOF** é definido em **stdio.h** e significa fim do arquivo.

Lendo um caractere com getc():

A função **getc()** é usada para ler caracteres de um *stream* que foi previamente aberto para leitura por **fopen()**. A função **getc()** apresenta o seguinte protótipo:

int getc(FILE *fp):

onde **fp** é um ponteiro de arquivo do tipo **FILE** retornado por **fopen()**.

Por razoes históricas, **getc()** retorna um **int** (2 bytes = 16 bits), porém o byte de mais alta ordem é zero.

getc() retorna **EOF** quando o fim do arquivo tiver sido encontrado ou um erro tiver ocorrido. Portanto, para ler um arquivo-texto até o **EOF** podemos usar o seguinte trecho de código:

```
ch=getc(fp);
while(ch!=EOF) {
ch=getc(fp);
}
```

onde **fp** é um ponteiro de arquivo do tipo **FILE** retornado por **fopen()** e **ch** armazena o caractere lido do arquivo.

Para demais funções de I/O de caracteres ver seção "2.12.5 Character I/O Functions" na página 106 de https://www.fccdecastro.com.br/CursoC&C++/CLibraryReferenceGuide%20-%20Huss.pdf .

Usando a função feof() p/ detectar o EOF:

Quando um arquivo é aberto para leitura binária pode acontecer que **getc()** leia um valor inteiro correspondente ao **EOF** sem que o arquivo tenha sido efetivamente lido até o final. Quando isto acontece a declaração **while(ch!=EOF)** no código acima detecta erroneamente (como um falso alarme) um **EOF** ainda que o fim do arquivo físico não tenha sido encontrado. Para resolver esse problema, devemos usar a função **feof()**, que determina sem falso alarme o final de arquivo na leitura de um arquivo binário. **feof()** apresenta o seguinte protótipo, que é definido no *header* **stdio.h**:

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

feof() retorna um valor diferente de zero (= verdadeiro) se o final do arquivo tiver sido encontrado, caso contrário retorna zero (= falso). Neste contexto, o trecho de código abaixo lê um arquivo binário até que o final do arquivo seja encontrado, sem o problema do falso alarme:

```
while(!feop(fp))ch=getc(fp);
```

onde **fp** é um ponteiro de arquivo do tipo **FILE** retornado por **fopen().**

feof() pode ser aplicada para detectar o **EOF** tanto na leitura de arquivos textos como na leitura de arquivos binários.

Usando a função fclose() p/ fechar um arquivo:

A função **fclose()** é usada para fechar um arquivo associado a um *stream* que foi aberto por **fopen()**. Quando **fclose()** é chamada através de **fclose(fp)** ela escreve quaisquer dados restantes no *buffer* de armazenamento intermediário que é membro da estrutura **FILE** (ver slide 63) apontada pelo ponteiro **fp** . A seguir **fclose()** faz um fechamento formal do arquivo em nível de sistema operacional. Uma falha no fechamento de um arquivo leva a todo tipo de problemas, incluindo a perda de dados, arquivos destruídos e a possibilidade de erros intermitentes na execução do programa. Uma chamada à **fclose()** também libera memória de todos os dados associados aos demais membros da estrutura **FILE**.

Assim que os dados forem lidos e/ou escritos em um arquivo, e desde que não haja necessidade de posterior leitura e/ou escrita no arquivo, **é aconselhável que o *stream* seja imediatamente fechado com fclose()**. Evitar deixar *streams* abertos sem necessidade: Nunca se sabe, por exemplo, o efeito de um ponteiro não inicializado quando ele escrever na memória ...

fclose() apresenta o seguinte protótipo, que é definido no *header* **stdio.h**:

int fclose(FILE*fp);

onde **fp** é um ponteiro de arquivo do tipo **FILE** retornado por **fopen()**.

fclose() retorna zero quando a operação de fechamento teve sucesso e retorna um valor diferente de zero caso tenha ocorrido algum erro.

Usando a função ferror() p/ determinar se uma operação com arquivo resultou em erro:

A funçao **ferror()** deve ser chamada imediatamente após qualquer função que efetue uma operação com arquivo. **ferror()** apresenta o seguinte protótipo, que é definido no *header* **stdio.h**:

int ferror(FILE*fp);

onde **fp** é um ponteiro de arquivo do tipo **FILE** retornado por **fopen()**.

ferror() retorna um valor diferente de zero caso tenha ocorrido algum erro durante a operação com arquivo executada imediatamente antes da chamada de **ferror()** e retorna zero quando a operação teve sucesso.

Usando a função rewind() p/ colocar o indicador de posição no início do arquivo:

A função **rewind()** recoloca o indicador de posição do arquivo no início do arquivo associado ao ponteiro de arquivo especificado como seu argumento. **rewind()** apresenta o seguinte protótipo, que é definido no *header* **stdio.h**:

void rewind(FILE*fp);

onde **fp** é um ponteiro de arquivo do tipo **FILE** retornado por **fopen()**.

I/O de disco

Usando a função fwrite() para escrever um bloco de memória em arquivo de disco :

fwrite () escreve em disco um bloco de dados armazenado na memória. Apresenta o seguinte protótipo, definido no *header* **stdlio.h**:

unsigned fwrite(void *Ptr, unsigned NumBytes, unsigned NumElements, FILE *fp);

onde **Ptr** é um ponteiro **void** que armazena um endereço de memória correspondente ao início do segmento da memória onde se encontram armazenados os dados a serem escritos no arquivo de disco. Um ponteiro **void** significa que o tipo de endereço pode ser alterado através de um *cast* e ser armazenado em um ponteiro para qualquer tipo de dado, de modo ao ponteiro apontar para o tipo de dado que se desejar: **char**, **int**, **float**, **double**, vetor, matriz, **struct** (estrutura), etc...

fp é um ponteiro de arquivo do tipo **FILE** para um *stream* previamente aberto por **fopen()**.

NumBytes é o número de bytes de cada elemento do tipo de dado a ser escrito no arquivo apontado por **fp**. Usualmente o operador **sizeof()** determina o valor de **NumBytes** do tipo de dado.

NumElements é o número de elementos do tipo de dado a ser escrito no arquivo apontado por **fp**.

fwrite() retorna o número de elementos escritos no arquivo de disco, valor de retorno que resultará igual a **NumElements** exceto na ocorrência de um erro de escrita. Portanto, o valor de retorno pode ser usado para fazer o *check* do sucesso da operação de escrita.

Quando o arquivo for aberto para escrita de dados binários, **fwrite()** pode escrever em disco qualquer tipo de informação armazenada na memória. Isto é possível porque **fwrite()** executa uma copia exata byte a byte da memoria e grava em disco uma “fotografia” da memoria – uma imagem exata do que está armazenado na memória.

Usando a função fread() para ler dados gravados em arquivo de disco e armazenar os dados lidos em bloco de memória:

fread () lê dados de um arquivo de disco e armazena os dados na memória. Apresenta o seguinte protótipo, definido no *header* **stdlio.h**:

unsigned fread(void *Ptr, unsigned NumBytes, unsigned NumElements, FILE *fp);

onde **Ptr** é um ponteiro **void** que armazena um endereço de memória correspondente ao início do segmento da memória onde serão armazenados os dados a serem lidos do arquivo de disco. Um ponteiro **void** significa que o tipo de endereço pode ser alterado através de um *cast* e ser armazenado em um ponteiro para qualquer tipo de dado, de modo ao ponteiro apontar para o tipo de dado que se desejar: **char**, **int**, **float**, **double**, vetor, matriz, **struct** (estrutura), etc...

fp é um ponteiro de arquivo do tipo **FILE** para um *stream* previamente aberto por **fopen()**.

NumBytes é o número de bytes de cada elemento do tipo de dado a ser lido do arquivo apontado por **fp**. Usualmente o operador **sizeof()** determina o valor de **NumBytes** do tipo de dado.

NumElements é o número de elementos do tipo de dado a ser lido do arquivo apontado por **fp**.

fread() retorna o número de elementos lidos do arquivo de disco, valor de retorno que resultará igual a **NumElements** exceto na ocorrência de um erro de leitura ou caso o **EOF** seja encontrado. Portanto, o valor de retorno pode ser usado para fazer o *check* do sucesso da operação de leitura.

Quando o arquivo for aberto para leitura de dados binários, **fread()** pode ler do disco qualquer tipo de informação e armazenar na memória uma cópia "imagem" da informação lida do disco. Isto é possível porque **fread()** executa uma copia exata byte a byte dos dados gravados em disco e armazena na memória uma "fotografia" dos dados em disco – uma imagem exata do que está armazenado no disco.

I/O de disco

O código fonte https://www.fccdecastro.com.br/CursoC&C++/C/Mat2DBinWR.c a seguir demonstra o uso de **fwrite()** e **fread ()** para gravar-em e ler-de um arquivo binário (mat.dat) a matriz **float Mat[NLINS][NCOLS]** de **NLINS** linhas e **NCOLS** colunas.

```c
/*****************************************************************************
* Aloca memoria dinamicamente para uma matriz Mat[NLINS][NCOLS], atribui valores
* aos elementos de Mat de acordo com Mat[Linha][Coluna]=(NCOLS*Linha)+Coluna
* e imprime Mat na tela do console. Grava em disco a matriz Mat original em
* um arquivo binário Mat.dat. le Mat.dat do disco e armazena na memoria do heap
* em uma matriz MatR. Imprime MatR na tela do console para efeito de comparacao
* e verificacao de conformidade com a matriz original Mat.
*****************************************************************************/
/*****************************************************************************
* HEADERS:
*****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
/*****************************************************************************
* MACROS:
*****************************************************************************/
static double sqrarg;
#define SQR(a) ((sqrarg=(a)) == 0.0 ? 0.0 : sqrarg*sqrarg)
/*****************************************************************************
* PROGRAM DEFINITIONS:
*****************************************************************************/
#define NLINS 8 // numero de linhas da matriz Mat
#define NCOLS 7 // numero de colunas da matriz Mat
#define BUFSIZE 0xFFFF /* file buffer size: 0xFFFF = 64Kb */
```

I/O de disco

```
26  /*************************************************************************
27  * FUNCTION PROTOTYPES:
28  *************************************************************************/
29  float **FloatMatBRead(float **Mat, char *FileName, char *buffer, unsigned *NLinhas,unsigned *NColunas);
30  void FloatMatBWrite(float **Mat, char *FileName, char *buffer,unsigned NLinhas,unsigned NColunas);
31  float **FloatMatCAlloc(unsigned NLinhas,unsigned NColunas);
32  void FloatMatFree(float **Mat);
33  void FloatMatPrint(float **Mat,unsigned NLinhas,unsigned NColunas);
34  /*************************************************************************
35  * main():
36  *************************************************************************/
37  void main(void){
38  float **Mat, **MatR; // matrizes 2D
39  unsigned register Linha, Coluna; // inexadores de linas e colunas
40  char *buffer; // buffer p/ I/O de disco
41  unsigned Nlinhas_R,NColunas_R; // numero de linhas e colunas da matriz lida de disco
42
43  /* Aloca memoria para o buffer p/ I/O de disco */
44  buffer=(char *)malloc(BUFSIZE*sizeof(char));
45  if(!buffer){puts("Nao consigo alocar memoria para o buffer de disco!");exit(1);}
46
47  /* aloca memoria p/ a matriz original Mat */
48  Mat=FloatMatCAlloc(NLINS,NCOLS);
49
50  /* atribui valores aos elementos da matriz Mat */
51  for(Linha=0;Linha<NLINS;Linha++){
52      for(Coluna=0;Coluna<NCOLS;Coluna++){
53      Mat[Linha][Coluna]=(NCOLS*Linha)+Coluna; // ver slides 19,20 e 21 do Cap II.1
54      }
55  }
```

I/O de disco

```
56  /* imprime matriz Mat original na tela do console */
57  puts("Matriz original:");
58  FloatMatPrint(Mat,NLINS,NCOLS);
59
60  /* grava NLINS, NCOLS e a matriz Mat no arquivo binario Mat.dat*/
61  FloatMatBWrite(Mat,"Mat.dat", buffer,NLINS, NCOLS);
62
63  /* Le o arquivo binario Mat.dat, armazena o numero de linhas em Nlinhas_R,
64   armazena o numero de colunas em NColunas_R e armazena a matriz gravada em
65   disco na memoria alocada no heap p/ a matriz MatR */
66  MatR=FloatMatBRead(MatR, "Mat.dat", buffer, &Nlinhas_R,&NColunas_R);
67
68  /* imprime na tela do console a matriz MatR lida do disco  */
69  puts("Matriz lida do disco:");
70  FloatMatPrint(MatR,Nlinhas_R,NColunas_R);
71
72  }
```

I/O de disco

```
73
74  /*************************************************************************
75  * FUNC: float **FloatMatBRead(float **Mat, char *FileName, char *buffer, unsigned *NLinhas,unsigned *NC
76  *
77  * DESC: Le o arquivo binario FileName e armazena na matriz float Mat[NLinhas][NColunas].
78  *************************************************************************/
79  float **FloatMatBRead(float **Mat, char *FileName, char *buffer, unsigned *NLinhas,unsigned *NColunas)
80  {
81  FILE  *inp; // ponteiro de arquivo
82  int CheckR; // armazena o numero de elementos lidos do disco, conforme abaixo
83
84  /* abre arquivo p/ leitura binaria */
85  if((inp=fopen(FileName,"rb"))==NULL){
86    printf("Nao posso abrir o arquivo %s!\n",FileName);exit(1);}
87
88  /* estabelece buffer de I/O de tamanho BUFSIZE para maximizar a velocidade de leitura */
89  if (setvbuf(inp,buffer,_IOFBF, BUFSIZE) != 0){
90   printf("Nao consigo estabelecer o buffer p/ o arquivo %s!\n",FileName);exit(1);}
91
92  /* le o numero de Linhas NLinhas da matriz gravada em disco */
93  CheckR=fread((unsigned *)NLinhas, sizeof(unsigned), 1, inp);
94  if(CheckR!=1){printf("Nao consigo ler NLinhas do arquivo %s!\n",FileName);exit(1);}
95
96  /* le o numero de colunas NColunas da matriz gravada em disco */
97  CheckR=fread((unsigned *)NColunas, sizeof(unsigned), 1, inp);
98  if(CheckR!=1){printf("Nao consigo ler NColunas do arquivo %s!\n",FileName);exit(1);}
99
100 /* imprime NLinhas e NColunas natela do console */
101 printf("\nNLinhas lidas=%u\tNColunas lidas=%u\n",*NLinhas,*NColunas);
102
```

I/O de disco

```
103  /* aloca memoria p/ a matriz Mat que serah lida do disco */
104  Mat=FloatMatCAlloc(*NLinhas,*NColunas);
105
106  /* le de disco a matriz gravada e armazena na memoria alocada no heap
107  p/ a matriz Mat atraves de FloatMatCAlloc() na declaracao anterior */
108  CheckR=fread((float *)Mat[0], sizeof(float), (*NLinhas)*(*NColunas), inp);
109  if(CheckR!=((*NLinhas)*(*NColunas))){printf("Nao consigo ler Mat do arquivo %s!\n",FileName);exit(1);}
110
111  fclose(inp); // fecha o arquivo
112
113  return Mat; // retorna o endereço do 1º elemento de Mat[][] - eh o mesmo que return &Mat[0]
114
115  }
```

I/O de disco

```
116
117 /*************************************************************************
118 * FUNC: void FloatMatBWrite(float **Mat, char *FileName, char *buffer,unsigned NLinhas,unsigned NC
119 *
120 * DESC: Grava no arquivo binario FileName a matriz float Mat[NLinhas][NColunas].
121 *************************************************************************/
122 void FloatMatBWrite(float **Mat, char *FileName, char *buffer,unsigned NLinhas,unsigned NColunas)
123 {
124 FILE  *out; // ponteiro de arquivo
125 int Check; // armazena o numero de elementos escritos em disco, conforme abaixo
126
127 /* abre arquivo p/ escrita binaria */
128 if((out=fopen(FileName,"wb"))==NULL){
129   printf("Nao posso abrir o arquivo %s!\n",FileName);exit(1);}
130
131 /* estabelece buffer de I/O de tamanho BUFSIZE para maximizar a velocidade de escrita */
132 if (setvbuf(out,buffer,_IOFBF, BUFSIZE) != 0){
133  printf("Nao consigo estabelecer o buffer p/ o arquivo %s!\n",FileName);exit(1);}
134
135 /* escreve no arquivo o numero de Linhas NLinhas da matriz que serah gravada em disco */
136 Check=fwrite((unsigned *)&NLinhas, sizeof(unsigned), 1, out);
137 if(Check!=1){printf("Nao consigo gravar NLinhas no arquivo %s!\n",FileName);exit(1);}
138
139 /* escreve no arquivo o numero de colunas NColunas da matriz que serah gravada em disco */
140 Check=fwrite((unsigned *)&NColunas, sizeof(unsigned), 1, out);
141 if(Check!=1){printf("Nao consigo gravar NColunas no arquivo %s!\n",FileName);exit(1);}
142
143 /* escreve no arquivo e grava a matriz Mat no arquivo de disco */
144 Check=fwrite((float *)Mat[0], sizeof(float), NLinhas*NColunas, out);
145 if(Check!=NLinhas*NColunas){printf("Nao consigo gravar Mat no arquivo %s!\n",FileName);exit(1);}
146
147 fclose(out); // fecha o arquivo
148
149 }
150
```

I/O de disco

```
151 /*****************************************************************************
152 * FUNC: float **FloatMatCAlloc(unsigned NLinhas,unsigned NColunas)
153 *
154 * DESC: Aloca memoria para a matriz float Mat[NLinhas][NColunas].
155 *****************************************************************************/
156 float **FloatMatCAlloc(unsigned NLinhas,unsigned NColunas)
157 {
158 float  *V; // vetor V[]
159 float  **Mat; // matriz Mat[][]
160 register unsigned i;
161
162 /* aloca memoria para todos os elementos da matriz Mat[NLinhas][NColunas]
163 na forma de um vetor V[NLinhasxNColunas] - ver slides 19,20 e 21 do Cap II.1 */
164 V = (float *)calloc((unsigned long)(NLinhas*NColunas),sizeof(float));
165 if(!V){puts("Nao consigo alocar memoria para o vetor V[]!");exit(1);}
166
167 /* aloca memoria para o vetor de ponteiros Mat[NLinhas], vetor que armazenarah
168 os endereços do primeiro elemento de cada linha da matriz Mat[NLinhas][NColunas] */
169 Mat=(float  **)calloc((unsigned long)NLinhas,sizeof(float *));
170 if(!Mat){puts("Nao consigo alocar memoria para o vetor Mat[]!");exit(1);}
171
172 /* atribui a cada elemento do vetor de ponteiros Mat[NLinhas] o endereço do primeiro
173 elemento de cada linha da matriz Mat[NLinhas][NColunas] */
174 for(i=0;i<NLinhas;i++){
175 Mat[i]=(float *)&V[i*NColunas];
176 }
177
178 /* retorna o endereço do primeiro elemento do vetor Mat[NLinhas] e que correponde
179 ao endereco do primeiro elemento da primeira linha da matriz Mat[NLinhas][NColunas] */
180 return Mat;
181 }
```

I/O de disco

```
182 /*****************************************************************************
183 * FUNC: void FloatMatFree(float **Mat)
184 *
185 * DESC: libera memoria alocada p/ a matrix float Mat[][].
186 *****************************************************************************/
187 void FloatMatFree(float **Mat)
188 {
189 free(Mat[0]); /* libera vetor float Mat[0]=&V[0]=V alocado atraves de
190 V = (float *)calloc((unsigned long)(NLinhas*NColunas),sizeof(float)) em
191  FloatMatCAlloc() */
192 free(Mat); /* libera vetor de ponteiros Mat alocado atraves de
193 Mat=(float  **)calloc((unsigned long)NLinhas,sizeof(float *)) em
194 em FloatMatCAlloc() */
195 }
196 /*****************************************************************************
197 * FUNC: void FloatMatPrint(float **Mat,unsigned NLinhas,unsigned NColunas)
198 *
199 * DESC: Imprime a matriz float Mat[NLinhas][NColunas] na tela do console.
200 *****************************************************************************/
201 void FloatMatPrint(float **Mat,unsigned NLinhas,unsigned NColunas)
202 {
203 register unsigned i,j;
204
205 for(i=0;i<NLinhas;i++){
206     for(j=0;j<NColunas;j++){
207     printf("%8.6g\t",Mat[i][j]);
208     }
209 printf("\n");
210 }
211 }
```

I/O de disco

Execução na tela do console:

```
C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\Exemplos>prgm
Matriz original:
       0              1              2              3              4              5              6
       7              8              9             10             11             12             13
      14             15             16             17             18             19             20
      21             22             23             24             25             26             27
      28             29             30             31             32             33             34
      35             36             37             38             39             40             41
      42             43             44             45             46             47             48
      49             50             51             52             53             54             55

NLinhas lidas=8 NColunas lidas=7
Matriz lida do disco:
       0              1              2              3              4              5              6
       7              8              9             10             11             12             13
      14             15             16             17             18             19             20
      21             22             23             24             25             26             27
      28             29             30             31             32             33             34
      35             36             37             38             39             40             41
      42             43             44             45             46             47             48
      49             50             51             52             53             54             55
```

Usando fprintf() para escrever dados formatados em arquivo de disco e fscanf() para ler dados formatados de arquivo de disco:

fprintf() se comporta como **printf()**, a diferença sendo que **printf()** envia os dados para a tela do console (*stream* **stdout** – ver https://en.wikipedia.org/wiki/Standard_streams) enquanto **fprintf()** envia os dados para um *stream* aberto por **fopen()**.

fscanf() se comporta como **scanf()**, a diferença sendo que **scanf()** lê dados do teclado (*stream* **stdin** – ver https://en.wikipedia.org/wiki/Standard_streams) enquanto **fscanf()** lê os dados de um *stream* aberto por **fopen()**.

Os protótipos, basicamente os mesmos protótipos de **printf()** e **scanf()** a menos do argumento **FILE *fp** , são definidos no *header* stdio.h, são:

int fprintf(FILE *fp, char *string_de_controle, lista_de_argumentos);

int fscanf(FILE *fp, char *string_de_controle, lista_de_argumentos);

onde **fp** é um ponteiro de arquivo do tipo **FILE** para um *stream* previamente aberto por **fopen()**. No caso de **fprintf()** o *stream* deve ser aberto para ecsrita e no caso de **fscanf()** o *stream* deve ser aberto para leitura.

Os **especificadores de formato** e **modificadores** para **fprintf()** e **fscanf()** são os mesmos de **printf()** e **scanf()** e são mostrados nas tabelas dos slides que seguem. A descrição da sintaxe de uso e detalhes de formatação para as funções **fprintf()** e **fscanf()** estão disponíveis na seção "2.12.4 Formatted I/O Functions" na página 100 e na seção "1.1.2 Escape sequences" na página 17 de https://www.fccdecastro.com.br/CursoC&C++/CLibraryReferenceGuide - Huss.pdf .

fprintf() retorna o número de bytes (=número de caracteres) escritos com sucesso no *stream* e retorna um número negativo em caso de erro de escrita.

fscanf() retorna o número de itens da **lista_de_argumentos** lidos com sucesso do *stream* e retorna -1 quando lê o **EOF**.

Portanto, o valor de retorno de **fscanf()** pode ser usado para fazer o *check* do sucesso da operação de leitura comparando com o número de variáveis na lista de argumentos. Por exemplo, para a declaração **teste=fscanf(fp,"%f%f",&x1, &x2)** o sucesso da operação de leitura é indicado por um valor 2 armazenado na variável **teste** após a operação de leitura. Se **teste=-1** significa que **fscanf()** leu o **EOF** no final do arquivo.

I/O de disco

A **string_de_controle** para **fprintf()** consiste em dois tipos de itens. O primeiro tipo é o conjunto de caracteres que se deseja que sejam escritos no arquivo. O segundo tipo é o conjunto de especificadores de formato, que definem a maneira como os argumentos são escritos. Um especificador de formato começa com um sinal de porcentagem (%) e é seguido pelo código de formato. Deve haver o mesmo número de argumentos quantos forem os especificadores de formato, necessitando haver uma correspondência de tipo entre especificador de formato e argumento na ordem de ocorrência respectiva de cada um na **string_de_controle** e na **lista_de_argumentos.** Por exemplo **fprintf(fp,"Olá %c %d %s",'c',10,"todos!")** grava no arquivo apontado por fp o seguinte conjunto de caracteres "**Olá c 10 todos!**".

Especificadores de formato para **fprintf()**:	
Especificador:	Formato resultante nos dados escritos no arquivo:
%c	Um único caractere
%d	Decimal inteiro
%i	Decimal inteiro
%e	Notaçao científica ('e' minúsculo: 6.22e17)
%E	Notaçao científica ('E' maiúsculo: 6.22E17)
%f	Decimal ponto flutuante
%g	Usa %e ou %f - o que ocupar menos espaço na tela
%G	Igual a %g só que usa 'E' maiúsculo no caso de acionar %e
%o	Octal inteiro
%s	String de caracteres
%u	Decimal inteiro sem sinal
%x	Hexadecimal inteiro (usando letras minúsculas)
%X	Hexadecimal inteiro (usando letras maiúsculas)
%%	Imprime o sinal %
%p	O endereço hexadecimal para o qual aponta o argumento ponteiro
%n	O argumento associado é um ponteiro para um inteiro onde é armazenado o número de caracteres escritos na chamada de **fprintf()**

Os especificadores de formato podem ter **modificadores** que especificam o tamanho do campo, o número de casas decimais e um indicador de justificação à direita.

Por exemplo, um número inteiro colocado entre o sinal % e o especificador de formato atua como um especificador de largura mínima do campo. Este especificador preenche a saída com brancos ou zeros para assegurar que o campo esteja com pelo menos um determinado comprimento mínimo. Se a *string* ou número é maior que o comprimento mínimo, ela será escrita por completo, mesmo se o comprimento mínimo for ultrapassado. O preenchimento padrão é feito com espaços. Se desejarmos preencher com zeros, é necessário colocar um 0 antes do especificador de comprimento do campo. Por exemplo, **%05d** preenche um número de menos de cinco dígitos com zeros, de modo que o tamanho total seja cinco.

Para especificar o número de casas decimais escritas para um número em ponto flutuante, coloca-se um ponto decimal depois do especificador de tamanho de campo, seguido pelo número de casas decimais desejadas. Por exemplo, **%10.4f** exibe um número de, no mínimo, dez caracteres de comprimento com quatro casas decimais após o ponto. Quando isto é aplicado a *strings* ou inteiros, o número seguinte ao ponto especifica o comprimento máximo do campo. Por exemplo, **%5.7s** exibe uma *string* de no mínimo cinco caracteres e no máximo sete. Se a *string* a ser escrita exceder sete caracteres, ela será truncada.

Por definição, toda saída é justificada à direita: se a largura de campo é maior que o dado impresso, o dado será colocado na extremidade direita do campo. Pode-se forçar a informação a ser justificada à esquerda colocando-se um sinal de menos (-) diretamente depois do %. Por exemplo, %-10.2f justificará a esquerda um número em ponto flutuante, com duas casas decimais, em um campo de 10 caracteres.

Existem dois modificadores dos especificadores de formato que permitem **fprintf()** gravar inteiros **short** e **long**. Esses modificadores podem ser aplicados aos especificadores de tipo **d**,**i**,**o**,**u**,**x**.

O modificador **l** indica que um dado tipo **long** será escrito no arquivo:

%ld long int

%lu unsigned long int

O modificador **h** indica que um dado tipo **short int** será escrito no arquivo:

%hu unsigned short int

%hd short int

%ho unsigned short int octal

O modificador **l** também pode alterar os especificadores **e**,**f**,**g** para indicar que um **double** será escrito:

%lf double em ponto flutuante

%Lf long double em ponto flutuante

%LE long double em notaçao científica c/ 'E' maiúsculo

I/O de disco

```
/* Exemplos de especificadores de formato e de campo para fprintf() */
#include<stdio.h>
void main(void)
{
fprintf(stdout,"%-7.2f\n",123.234);
fprintf(stdout,"%7.2f\n",123.234);
fprintf(stdout,"%-5.2f\n",123.234);
fprintf(stdout,"%5.2f\n",3.324);
fprintf(stdout,"%10s\n","Alo");
fprintf(stdout,"%-10s\n","Alo");
fprintf(stdout,"%5.7s\n","123456789");
}
```

Execução na tela do console:

```
123.23
 123.23
123.23
 3.32
       Alo
Alo
1234567
```

Nota: **stdout** é um ponteiro do tipo FILE automaticamente aberto na inicialização do programa e que aponta para a porta da tela do console, i.e., o console é "visto" pela fprintf() como um arquivo de disco. Ver https://en.wikipedia.org/wiki/Standard_streams .

I/O de disco

A **string_de_controle** para **fscanf()** consiste de três conjunto de caracteres:

- especificadores de formato
- caracteres de espaço em branco
- caracteres de espaço não branco

Os especificadores de formato de entrada são precedidos pelo sinal % e dizem a **fscanf()** qual tipo de dado deve ser lido do arquivo. A lista de argumentos contém uma lista de ponteiros para as variáveis-argumentos correspondentes em tipo, número e ordem de ocorrência aos especificadores de formato na *string* de controle.

Especificadores de formato de fscanf():	
Especificador	Dado a ser lido:
%c	Um único caractere
%d	Decimal inteiro
%i	Decimal inteiro
%e ou %E	Notaçao científica
%f	Decimal ponto flutuante
%g ou %G	Lê %e ou %f
%o	Octal inteiro
%s	String de caracteres
%u	Decimal inteiro sem sinal
%x	Hexadecimal inteiro
%p	Lê o endereço hexadecimal para o qual irá apontar o argumento ponteiro
%n	O argumento associado é um ponteiro para um inteiro onde é colocado o número de caracteres lidos até então..
%le ou %lg	Idêntico ao caso sem o modificador l só que lê double em vez de float
%l seguido de d,i,o,u,x	Idêntico ao caso sem o modificador l só que lê long
%h seguido de d,i,o,u x	Idêntico ao caso sem o modificador h só que lê short

Caracteres de espaço em branco:

Um caratere de espaço em branco na *string* de controle faz com que **fscanf()** salte um ou mais caracteres em branco no *buffer* de entrada. Um caractere de espaço em branco pode ser:

- um espaço ‘ ‘
- uma tabulação ‘\t’
- um *newline* ‘\n’

Especificamente, um caractere de espaço em branco na *string* de controle fará com que **fscanf()** leia, mas não armazene, qualquer número de espaços em branco (incluindo zero, isto é, nenhum) de caracteres de espaço em branco até o primeiro caractere que não seja um espaço em branco. Por exemplo, **"%d,%d"** faz com que fscanf() leia um inteiro, e então leia e descarte uma vírgula e, finalmente, leia um outro inteiro. Se o caractere especificado não é encontrado **fscanf()** termina prematuramente e não lê o segundo inteiro.

Um * colocado depois do % e antes do especificador de formato lerá um dado do tipo especificado, mas suprimirá sua atribuição. Assim,

fscanf(fp,"%d%*c%d",&x,&y);

quando lê a entrada 10!20, armazenará o valor 10 em x descartando o caractere '!’ e armazenará em y o valor 20.

Os especificadores de formato podem especificar um modificador de comprimento máximo do campo. Esse modificador é um número inteiro colocado entre o sinal % e o código de comando de formato, que limita o número de caracteres lidos para qualquer campo. Por exemplo, para ler não mais do que 20 caracteres que serão armazenados em **str**, escrevemos:

fscanf(fp,"%20s",str);

Se o *stream* de entrada contiver mais do que 20 caracteres, **fscanf()** termina a leitura e só atribui à **str** os 20 primeiros caracteres deixando os restantes no *stream*. Uma chamada subsequente a **fscanf()** começará a leitura do *atream* de entrada onde ela terminou. Por exemplo, se o arquivo de disco armazena o texto abaixo

ABCDEFGHIJKLMNOPQRSTUVWXYZ

em resposta à chamada de **fscanf()** para este exemplo, somente os 20 primeiros caracteres, até 'T', serão armazenados em **str**. Na próxima chamada à **fscanf()** será atribuído a **str** a *string* "UVWXYZ“.

I/O de disco

O código fonte https://www.fccdecastro.com.br/CursoC&C++/C/Mat2DTxtWR.c a seguir demonstra o uso de **fprintf()** e **fscanf ()** para gravar-em e ler-de um arquivo texto (mat.txt) a matriz **float Mat[NLINS][NCOLS]** de **NLINS** linhas e **NCOLS** colunas.

```
1  /*************************************************************************
2  * Aloca memoria dinamicamente para uma matriz Mat[NLINS][NCOLS], atribui valores
3  * aos elementos de Mat de acordo com Mat[Linha][Coluna]=(NCOLS*Linha)+Coluna
4  * e imprime Mat na tela do console. Grava em disco a matriz Mat original em
5  * um arquivo texto ASCII Mat.txt. le Mat.txt do disco e armazena na memoria do
6  * heap em uma matriz MatR. Imprime MatR na tela do console para efeito de
7  * comparacao e verificacao de conformidade com a matriz original Mat.
8  *************************************************************************/
9  /*************************************************************************
10 * HEADERS:
11 *************************************************************************/
12 #include <stdio.h>
13 #include <stdlib.h>
14 #include <math.h>
15 #include <string.h>
16 /*************************************************************************
17 * MACROS:
18 *************************************************************************/
19 static double sqrarg;
20 #define SQR(a) ((sqrarg=(a)) == 0.0 ? 0.0 : sqrarg*sqrarg)
```

I/O de disco

```
21  /*****************************************************************
22  * PROGRAM DEFINITIONS:
23  *****************************************************************/
24  #define NLINS 8 // numero de linhas da matriz Mat
25  #define NCOLS 7 // numero de colunas da matriz Mat
26  #define BUFSIZE 0xFFFF /* file buffer size: 0xFFFF = 64Kb */
27  #define MAXFLOAT 3.4E38 /* 32 bits max float */
28  /*****************************************************************
29  * FUNCTION PROTOTYPES:
30  *****************************************************************/
31  float **FloatMatCAlloc(unsigned NLinhas,unsigned NColunas);
32  void FloatMatFree(float **Mat);
33  void FloatMatPrint(float **Mat,unsigned NLinhas,unsigned NColunas);
34  void FloatMatWrite(float **Mat,char *FileName,char *buffer,unsigned NLinhas,unsigned N
35  float  **FloatMatReadCheck(float **Mat,char *FileName,char *buffer,unsigned *NLinhas,u
36  unsigned long GetFileNumLines(FILE *Fp);
37  long GetFileNumSamples(FILE *Fp);
```

I/O de disco

```
38  /*************************************************************************
39  * main():
40  *************************************************************************/
41  void main(void){
42  float **Mat, **MatR; // matrizes 2D
43  unsigned register Linha, Coluna; // inexadores de linas e colunas
44  char *buffer; // buffer p/ I/O de disco
45  unsigned Nlinhas_R,NColunas_R; // numero de linhas e colunas da matriz lida de disco
46
47  /* Aloca memoria para o buffer p/ I/O de disco */
48  buffer=(char *)malloc(BUFSIZE*sizeof(char));
49  if(!buffer){puts("Nao consigo alocar memoria para o buffer de disco!");exit(1);}
50
51  /* aloca memoria p/ a matriz original Mat */
52  Mat=FloatMatCAlloc(NLINS,NCOLS);
53
54  /* atribui valores aos elementos da matriz Mat */
55  for(Linha=0;Linha<NLINS;Linha++){
56      for(Coluna=0;Coluna<NCOLS;Coluna++){
57      Mat[Linha][Coluna]=(NCOLS*Linha)+Coluna; // ver slides 19,20 e 21 do Cap II.1
58      }
59  }
60  /* imprime matriz Mat original na tela do console */
61  puts("Matriz original:");
62  FloatMatPrint(Mat,NLINS,NCOLS);
63
64  /* grava Mat[NLINS][NCOLS] original no arquivo texto Mat.txt */
65  FloatMatWrite(Mat,"Mat.txt",buffer,NLINS,NCOLS);
```

I/O de disco

```
66
67  /* Le o arquivo texto Mat.txt, armazena o numero de linhas em Nlinhas_R,
68     armazena o numero de colunas em NColunas_R e armazena a matriz gravada em
69     disco na memoria alocada no heap p/ a matriz MatR */
70  MatR=FloatMatReadCheck(MatR,"Mat.txt",buffer,&Nlinhas_R,&NColunas_R);
71
72  /* imprime na tela do console a matriz MatR lida do disco  */
73  puts("Matriz lida do disco:");
74  FloatMatPrint(MatR,Nlinhas_R,NColunas_R);
75  }
```

I/O de disco

```
76  /*************************************************************************
77  * FUNC: float **FloatMatCAlloc(unsigned NLinhas,unsigned NColunas)
78  *
79  * DESC: Aloca memoria para a matriz float Mat[NLinhas][NColunas].
80  *************************************************************************/
81  float **FloatMatCAlloc(unsigned NLinhas,unsigned NColunas)
82  {
83  float  *V; // vetor V[]
84  float  **Mat; // matriz Mat[][]
85  register unsigned i;
86
87  /* aloca memoria para todos os elementos da matriz Mat[NLinhas][NColunas]
88  na forma de um vetor V[NLinhasxNColunas] - ver slides 19,20 e 21 do Cap II.1 */
89  V = (float *)calloc((unsigned long)(NLinhas*NColunas),sizeof(float));
90  if(!V){puts("Nao consigo alocar memoria para o vetor V[]!");exit(1);}
91
92  /* aloca memoria para o vetor de ponteiros Mat[NLinhas], vetor que armazenarah
93  os endereços do primeiro elemento de cada linha da matriz Mat[NLinhas][NColunas] */
94  Mat=(float  **)calloc((unsigned long)NLinhas,sizeof(float *));
95  if(!Mat){puts("Nao consigo alocar memoria para o vetor Mat[]!");exit(1);}
96
```

```
 97  /* atribui a cada elemento do vetor de ponteiros Mat[NLinhas] o endereço do primeiro
 98  elemento de cada linha da matriz Mat[NLinhas][NColunas] */
 99  for(i=0;i<NLinhas;i++){
100  Mat[i]=(float *)&V[i*NColunas];
101  }
102
103  /* retorna o endereço do primeiro elemento do vetor Mat[NLinhas] e que correponde
104  ao endereco do primeiro elemento da primeira linha da matriz Mat[NLinhas][NColunas] */
105  return Mat;
106  }
```

I/O de disco

```
107 /*************************************************************************
108  * FUNC: void FloatMatFree(float **Mat)
109  *
110  * DESC: libera memoria alocada p/ a matrix float Mat[][].
111  *************************************************************************/
112 void FloatMatFree(float **Mat)
113 {
114 free(Mat[0]); /* libera vetor float Mat[0]=&V[0]=V alocado atraves de
115 V = (float *)calloc((unsigned long)(NLinhas*NColunas),sizeof(float)) em
116  FloatMatCAlloc() */
117 free(Mat); /* libera vetor de ponteiros Mat alocado atraves de
118 Mat=(float  **)calloc((unsigned long)NLinhas,sizeof(float *)) em
119 em FloatMatCAlloc() */
120 }
```

I/O de disco

```
121 /*****************************************************************************
122 * FUNC: void FloatMatPrint(float **Mat,unsigned NLinhas,unsigned NColunas)
123 *
124 * DESC: Imprime a matriz float Mat[NLinhas][NColunas] na tela do console.
125 *****************************************************************************/
126 void FloatMatPrint(float **Mat,unsigned NLinhas,unsigned NColunas)
127 {
128 register unsigned i,j;
129
130 for(i=0;i<NLinhas;i++){
131     for(j=0;j<NColunas;j++){
132     printf("%8.6g\t",Mat[i][j]);
133     }
134 printf("\n");
135 }
136 }
```

I/O de disco

```
137 /*************************************************************************
138 * FUNC: void FloatMatWrite(float **Mat,char *FileName,char *buffer,unsigned NRows,unsigned NCol
139 *
140 * DESC: Escreve (grava) a matriz float Mat[NLinhas][NColunas] no arquivo texto ASCII "FileName'
141 * Se FileName=NULL, escreve Mat p/ stdout.
142 *************************************************************************/
143 void FloatMatWrite(float **Mat,char *FileName,char *buffer,unsigned NLinhas,unsigned NColunas)
144 {
145 register unsigned i,j;
146 FILE *out;
147 int test;
148
149 if(FileName!=NULL){// se FileName!=NULL escreve Mat no arquivo FileName
150
151 /* abre arquivo p/ escrita de texto ASCII */
152 if((out=fopen(FileName,"wt"))==NULL){
153   printf("Nao posso abrir para escrita o arquivo %s!\n",FileName);exit(1);}
154
155 /* estabelece buffer de I/O de tamanho BUFSIZE para maximizar a velocidade de escrita */
156 if (setvbuf(out, buffer, _IOFBF, BUFSIZE) != 0){
157  printf("Nao consigo estabelecer o buffer p/ o arquivo %s!\n",FileName);exit(1);}
158 }
159
160 else out=stdout; // escreve Mat[NLinhas][NColunas] no stdout (tela do console)
```

I/O de disco

```
161
162  /* escreve Mat[NLinhas][NColunas] no stream out */
163  for(i=0;i<NLinhas;i++){
164      for(j=0;j<NColunas;j++){
165      fprintf(out,"%8.6g\t",Mat[i][j]);
166      }
167  /* escreve o newline '\n' e testa a escrita */
168      test=fprintf(out,"\n");
169      if(test!=sizeof(char)){
170   printf("Erro ao escrever no arquivo %s!\n",FileName);exit(1);}
171  }
172
173  if(FileName!=NULL)fclose(out);// fecha o stream
174
175  }
```

I/O de disco

```
176 /*************************************************************************
177 * FUNC: float  **FloatMatReadCheck(float **Mat,char *FileName,char *buffer,unsigned *NLinhas,unsigned *NC
178 *
179 * DESC: Le do arquivo texto ASCII FileName a matriz float Mat[*NLinhas][*NColunas].
180 *       Os arguments *NLinhas and *NColunas retornam as dimensoes de Mat lida.
181 *       FloatMatReadCheck() faz a verificacao de conformidade dos dados lidos e
182 *       retorna Mat[*NLinhas][*NColunas] para a funcao chamadora.
183 *************************************************************************/
184 float  **FloatMatReadCheck(float **Mat,char *FileName,char *buffer,unsigned *NLinhas,unsigned *NColunas)
185 {
186 register unsigned i,j;
187 FILE *inp;
188 long NumSamples;
189 unsigned long NumLines;
190 int Test;
191 double Value;
192
193 /* abre arquivo FileName p/ leitura de texto */
194 if((inp=fopen(FileName,"rt"))==NULL){
195   printf("Nao posso abrir para leitura o arquivo %s!\n",FileName);exit(1);}
196
197 /* estabelece buffer de I/O de tamanho BUFSIZE para maximizar a velocidade de leitura */
198 if (setvbuf(inp, buffer, _IOFBF, BUFSIZE) != 0){
199  printf("Nao consigo estabelecer o buffer p/ o arquivo %s!\n",FileName);exit(1);}
200
```

I/O de disco

```
201 /* determina o numero de amostras (numeros) float no arquivo apontado por inp */
202 NumSamples=GetFileNumSamples(inp);
203 if(NumSamples==-1){
204  printf("Nao consigo determinar o numero de amostras no arquivo %s!\n",FileName);exit(1);}
205
206 /* determina o numero de linhas no arquivo apontado por inp */
207  NumLines=GetFileNumLines(inp);
208
209 /* verifica se a matriz eh retangular */
210 if(NumSamples%NumLines){
211  printf("A matriz lida de %s nao eh retangular!\n",FileName);exit(1);}
212
213 /* define/determina o numero de linhas e colunas da matriz */
214 *NLinhas=NumLines;
215 *NColunas=NumSamples/NumLines;
216
217 /* aloca memoria p/ a matriz Mat que serah lida do disco */
218 Mat=FloatMatCAlloc(*NLinhas,*NColunas);
219
220 /* le de disco os elementos da matriz gravada em disco e atribui aos respectivos
221 elementos Mat[i][j] da matriz Mat[*NLinhas][*NColunas] */
222 for(i=0;i<*NLinhas;i++){
223     for(j=0;j<*NColunas;j++){
224         Test=fscanf(inp,"%lf",&Value);
225         if(Test!=1){printf("Nao consigo ler a matriz no arquivo %s!\n",FileName);exit(1);}
226         if(fabs(Value)>MAXFLOAT){printf("Excedido o valor maximo p/ float no arquivo %s!\n",FileName);exit(1);}
227         Mat[i][j]=Value; // atribui double Value a float Mat[i][j]
228         }
229 }
230
231 fclose(inp); // fecha  o stream
232 return Mat; // retorna o endereco do 1° elemento de Mat[][] - eh o mesmo que return &Mat[0]
233 }
```

```
234
235 /*****************************************************************************
236 * FUNC: unsigned long GetFileNameNumLines(FILE *Fp)
237 *
238 * DESC: Retorna o numero de linhas de um arquivo texto apontado por Fp.
239 *****************************************************************************/
240 #define MAXLINESIZE 65536 /* maximo numero de caracteres em cada linha lida.
241                              Cada linha eh terminada por '\n' */
242 unsigned long GetFileNumLines(FILE *Fp)
243 {
244
245   char BufferDeLinha[MAXLINESIZE];
246   register unsigned long Count;
247
248   Count=0;
249   while (fgets(BufferDeLinha,MAXLINESIZE-2,Fp)){/* fgets() Le uma linha do
250   arquivo apontado por fp e armazena na string BufferDeLinha[MAXLINESIZE].
251   fgets() termina a leitura da linha quando: (1) MAXLINESIZE-2 caracteres sao
252   lidos, (2) o caractere '\n' eh lido ou (3) o EOF eh lido - o que ocorrer primeiro.
253   O '\n' é copiado para a string BufferDeLinha[MAXLINESIZE]. Um caractere
254    nulo  '\0' é anexado ao final da string BufferDeLinha[MAXLINESIZE] */
255   Count++;
256   }
257
258   if(strlen(BufferDeLinha)==1)Count--;/* p/ descontar um eventual ultimo '\n'
259                                          sem caracteres previos na linha lida do arquivo,
260                                          como acontece, por exemplo, no caso em que um
261                                          operador humano tenha digitado <ENTER> no final
262                                          do arquivo texto lido do disco */
263
264   rewind(Fp); // reseta o indicador de posicao
265   return Count; // retorna o numero de linhas lidas
266 }
```

I/O de disco

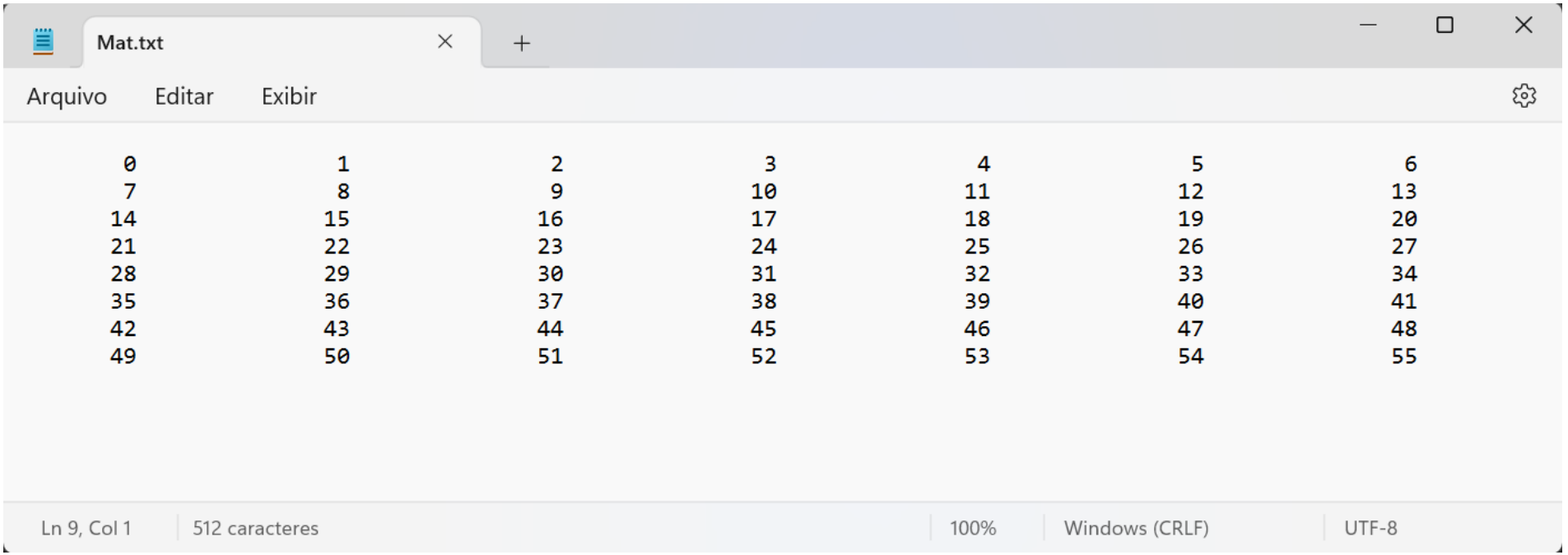
```
267 /*****************************************************************************
268 * FUNC: long GetFileNameNumSamples(FILE *Fp)
269 *
270 * DESC: Retorna o numero de elementos float lidos do arquivo texto apontado por Fp.
271 *       Retorna -1 em caso de erro de leitura.
272 *****************************************************************************/
273 long GetFileNumSamples(FILE *Fp)
274 {
275
276 register long Count;
277 float x;
278 int test;
279
280 Count=0;
281
282 while((test=fscanf(Fp,"%f",&x))!=-1){// le elementos float ateh o EOF
283 if(test!=1)return -1; // retorna -1 em erro de leitura
284 Count++; // conta o numero de elementos lidos
285 }
286
287 rewind(Fp); // reseta o indicador de posicao do arquivo
288 return Count; // retorna o numero de elementos lidos
289
290 }
```

Execução na tela do console:

```
C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\Exemplos>prgm
Matriz original:
      0       1       2       3       4       5       6
      7       8       9      10      11      12      13
     14      15      16      17      18      19      20
     21      22      23      24      25      26      27
     28      29      30      31      32      33      34
     35      36      37      38      39      40      41
     42      43      44      45      46      47      48
     49      50      51      52      53      54      55
Matriz lida do disco:
      0       1       2       3       4       5       6
      7       8       9      10      11      12      13
     14      15      16      17      18      19      20
     21      22      23      24      25      26      27
     28      29      30      31      32      33      34
     35      36      37      38      39      40      41
     42      43      44      45      46      47      48
     49      50      51      52      53      54      55
```

I/O de disco

Abrindo o arquivo Mat.txt com o Bloco de Notas (Notepad.exe):



Usando fseek() para setar o indicador de posição de um arquivo:

Nota: Ver significado do verbo "setar" em

https://www.dicionarioinformal.com.br/setar/#

O protótipo de **fseek()** é definidos no *header* stdio.h, e, especificamente, é conforme segue:

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

onde

- **fp** é um ponteiro de arquivo do tipo **FILE** para um *stream* previamente aberto por **fopen()**.
- (long int) **numbytes** é o número de bytes a partir da **origem** em que se deseja setar a posição do indicador de posição do arquivo.
- **origem** é qualquer uma das seguintes macros definidas em stdio.h.

Origem	Nome da Macro	Valor numérico
começo do arquivo	**SEEK_SET**	0
posição corrente	**SEEK_CUR**	1
fim do arquivo	**SEEK_END**	2

Portanto:

Para setar a posição do indicador de posição do arquivo **numbytes** a partir do começo do arquivo, **origem** deve ser **SEEK_SET**.

Para setar a posição do indicador de posição do arquivo **numbytes** a partir da posição corrente, **origem** deve ser **SEEK_CUR**.

Para setar a posição do indicador de posição do arquivo **numbytes** de trás para a frente a partir do final do arquivo, **origem** deve ser **SEEK_END**.

I/O de disco

O código fonte https://www.fccdecastro.com.br/CursoC&C++/C/ledisco.c a seguir demonstra o uso de **fseek()** para setar a posição do indicador de posição de um arquivo especificado na linha de comando:

```
/*****************************************************************************
* Le um setor de 128 bytes do disco onde estah gravado um arquivo especificado
* na linha de comando. Os 128 bytes lidos sao mostrados em hexadecimal e em char
* ASCII imprimivel.
*****************************************************************************/
/*****************************************************************************
* HEADERS:
*****************************************************************************/
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
/*****************************************************************************
* PROGRAM DEFINITIONS:
*****************************************************************************/
#define TAMANHO 128  /* tamanho do setor lido em bytes */
/*****************************************************************************
* FUNCTION PROTOTYPES:
*****************************************************************************/
void Display(unsigned char *Buffer, long int NumBytesLido);
```

I/O de disco

```
20  /*****************************************************************************
21  * main():
22  *****************************************************************************/
23  void main(int argc, char *argv[])
24  {
25  FILE *fp;
26  long int setor, NumBytesLido;
27  unsigned char buffer[TAMANHO];
28
29  /* help linha de comando*/
30  if(argc!=2){puts("Uso: prgm nome_do_arquivo");exit(1);}
31
32  /* abre o arquivo no modo leitura binaria */
33  if((fp=fopen(argv[1],"rb"))==NULL){
34  puts("Erro ao abrir arquivo!"); exit(1);}
35
36  while(1){
37  printf("Informe indice (0,1,2...) do setor de 128 bytes a ser lido do disco (-1 p/ terminar):");
38  scanf("%ld",&setor);
39  if(setor<0) break;
40
41  /* seta o localizador de posição 'setor*TAMANHO bytes' adiante do inicio do arquivo (SEEK_SET) */
42  if(fseek(fp, setor*TAMANHO , SEEK_SET)){puts("Erro de busca!");exit(1);}
43
44  /* le 128 bytes (=TAMANHO) a partir do localizador de posicao setado por fseek() */
45  if((NumBytesLido = fread(buffer,1,TAMANHO,fp)) != TAMANHO ) puts("EOF encontrado!");
46
47  /* imprime na tela do console */
48  printf("Bytes lidos mostrados em hexadecimal:\t\t\tBytes lidos mostrados em char ('.'-> char nao imprimivel):\n");
49
50  /* Mostra na tela do console os bytes lidos do setor de 128 bytes do disco */
51  Display(buffer,NumBytesLido);
52  }// while(1)
53  }
```

I/O de disco

```
54  /*****************************************************************************
55  * FUNC: void Display(unsigned char *Buffer, long int NumBytesLido)
56  *
57  * DESC: Imprime na tela do consoe os 128 bytes lidos. Os bytes lidos sao mostrados
58  *       em hexadecimal e em char ASCII imprimivel.
59  *****************************************************************************/
60  void Display(unsigned char *Buffer, long int NumBytesLido)
61  {
62  long int i,j;
63
64  for(i=0; i<NumBytesLido/16 ; i++) {/* i indexa as linhas: 0 a 7 (128/16=8) */
65      for(j=0; j<16; j++) printf("%02X ", Buffer[i*16+j]); /* imprime 16 bytes em hexa */
66      printf("\t"); /* separa os valores mostrados em hexa dos caracteres char ASCII */
67      for(j=0; j<16; j++) {
68          /* imprime 16 bytes em char ASCII (soh imprimiveis) */
69          if(isprint(Buffer[i*16+j]))printf("%c",Buffer[i*16+j]);
70          else printf("."); // imprime ponto se o caractere nao eh imprimivel
71      }
72      printf("\n"); // nova linha
73  }// for i
74  }
```

ver slides 19,20 e 21 do Cap II.1

ver slides 19,20 e 21 do Cap II.1

I/O de disco

Execução na tela do console, sendo mat.dat (ver exemplo slide 73) o arquivo especificado na linha de comando:

```
C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\Exemplos>prgm mat.dat
Informe indice (0,1,2...) do setor de 128 bytes a ser lido do disco (-1 p/ terminar):0
Bytes lidos mostrados em hexadecimal:                   Bytes lidos mostrados em char ('.'-> char nao imprimivel):
08 00 00 00 07 00 00 00 00 00 00 00 00 80 3F            ...............?
00 00 00 40 00 00 40 40 00 00 80 40 00 00 A0 40         ...@..@@...@...@
00 00 C0 40 00 00 E0 40 00 00 00 41 00 00 10 41         ...@...@...A...A
00 00 20 41 00 00 30 41 00 00 40 41 00 00 50 41         .. A..0A..@A..PA
00 00 60 41 00 00 70 41 00 00 80 41 00 00 88 41         ..`A..pA...A...A
00 00 90 41 00 00 98 41 00 00 A0 41 00 00 A8 41         ...A...A...A...A
00 00 B0 41 00 00 B8 41 00 00 C0 41 00 00 C8 41         ...A...A...A...A
00 00 D0 41 00 00 D8 41 00 00 E0 41 00 00 E8 41         ...A...A...A...A
Informe indice (0,1,2...) do setor de 128 bytes a ser lido do disco (-1 p/ terminar):1
EOF encontrado!
Bytes lidos mostrados em hexadecimal:                   Bytes lidos mostrados em char ('.'-> char nao imprimivel):
00 00 F0 41 00 00 F8 41 00 00 00 42 00 00 04 42         ...A...A...B...B
00 00 08 42 00 00 0C 42 00 00 10 42 00 00 14 42         ...B...B...B...B
00 00 18 42 00 00 1C 42 00 00 20 42 00 00 24 42         ...B...B.. B..$B
00 00 28 42 00 00 2C 42 00 00 30 42 00 00 34 42         ..(B..,B..0B..4B
00 00 38 42 00 00 3C 42 00 00 40 42 00 00 44 42         ..8B..<B..@B..DB
00 00 48 42 00 00 4C 42 00 00 50 42 00 00 54 42         ..HB..LB..PB..TB
Informe indice (0,1,2...) do setor de 128 bytes a ser lido do disco (-1 p/ terminar):-1
```

Estruturas

Em C, uma estrutura **struct** é uma coleção de variáveis referenciadas sob um único nome, provendo um meio de agregar de forma conjunta um rol de variáveis que representam informações relacionadas entre si. Uma declaração **struct** permite que se crie variáveis internas às estruturas. As variáveis internas que compreendem uma **struct** são denominadas de **membro da estrutura**. Por exemplo, considere a seguinte declaração **struct** que representa os dados do endereço de uma pessoa:

```
struct endereco {
char nome[30];
char rua[40];
char cidade[20];
char estado[3];
unsigned long int cep;
};
```

O nome **endereco** é denominado de ***tag* da estrutura**. O ***tag*** é opcional e apenas dá um nome ao tipo de estrutura, podendo ser usado em referências subsequentes ao tipo de estrutura. Note que a declaração acima não aloca qualquer memória para os membros da estrutura. No entanto, a memória é alocada no momento em que declaramos variáveis do tipo da estrutura, denominadas **variáveis de estrutura**, como por exemplo :

```
struct endereco  info_endereco; // declara a variável info_endereco como sendo do tipo struct endereco
```

ou

```
struct endereco  clientes[10]; // declara o vetor clientes[10], com 10 elementos do tipo struct endereco
```

Alternativamente, podemos declarar variáveis de estrutura juntamente com a declaração da estrutura, situação que aloca memória para cada variável de estrutura declarada:

```
struct endereco {
char nome[30];
char rua[40];
char cidade[20];
char estado[3];
unsigned long int cep;
} info_endereco, clientes[10]; // declaração das variáveis de estrutura 'info_endereço' e 'clientes[10]'
```

Estruturas

Ou ainda, podemos usar **typedef** para criar um novo tipo de dado que é uma estrutura. Por exemplo:

```
typedef struct Cadastro {
char nome[80];  /* membro 'nome' da estrutura 'cadastro' */
char cidade[20];  /* membro 'cidade' da estrutura 'cadastro' */
unsigned patrimonio; /* membro 'patrimonio' da estrutura 'cadastro' */
unsigned spc :1;  /* membro 'spc' da estrutura 'cadastro', um 'bit-field' de 1 bit */
} CADASTRO;  /* esta declaracao typedef criou um novo tipo de dado chamado 'CADASTRO', assim como existe
o tipo de dado 'float', 'int', etc. */
```

Ou seja, o tipo de dado **CADASTRO** é uma estrutura do tipo (*tag*) **Cadastro** . E daí podemos, por exemplo, declarar uma variável ponteiro p/ o novo tipo de dado e alocar dinamicamente um vetor de estruturas do tipo **Cadastro**:

```
CADASTRO *cliente; /* cliente eh um ponteiro (=vetor) p/ o tipo de dado 'CADASTRO' */

/* Aloca memoria para um vetor de 1000 elementos de do tipo de dado 'CADASTRO' e retorna o endereco
incial da area de armazenamento para o ponteiro 'cliente' */
cliente=(CADASTRO *)malloc(1000*sizeof(CADASTRO));
if(!cliente){puts("Memoria insuficiente!");exit(1);}
```

Se necessitamos declarar somente **uma variável de estrutura** de um determinado tipo de estrutura, o ***tag*** não é necessário. Por exemplo:

```
struct {
char nome[30];
char rua[40];
char cidade[20];
char estado[3];
unsigned long int cep;
}info_endereco;
```

Estruturas

A forma geral de uma declaração para uma estrutura é:

struct nome_do_tipo_da_estrutura {

tipo nome_do_membro_1;
tipo nome_do_membro_2;
tipo nome_do_membro_3;
.
.
.
tipo nome_do_membro_N;
}lista_de_variáveis_de estrutura;

onde o **nome_do_tipo_da_estrutura** (*tag*) pode ser omitido ou a **lista_de_variáveis_de estrutura** pode ser omitida, mas ambos não podem ser omitidos simultaneamente.

Lendo-valores-de e atribuindo-valores-para os membros de uma estrutura: operador ponto

Sintaxe:

nome_da_variável_de_estrutura . nome_do_membro

Por exemplo:

```
info_endereco.cep=91720260; /* atribui o valor 91720260 ao membro cep */
printf("%lu", info_endereco.cep); /* lê o valor do membro cep e imprime na tela */
printf("%s",info_endereco.nome); /* lê o valor do membro nome e imprime na tela */
```

onde **info_endereco** é a **variável de estrutura** declarada abaixo:

```
struct {
char nome[30];
char rua[40];
char cidade[20];
char estado[3];
unsigned long int cep;
}info_endereco;
```

Ponteiros para estruturas:

É absolutamente desaconselhável passar um vetor, uma matriz e em particular uma estrutura como valor para o argumento de uma função em razão da sobrecarga do *stack* que ocorreria devido ao significativo número de bytes que seriam armazenados no *stack*, implicando em um enorme tempo na transferência de dados entre a função chamadora e a função chamada, sem falar do risco óbvio de *overflow* do *stack*.

Em estruturas pequenas, com poucos membros, esta questão não chega a ser um problema. Mas, se a estrutura tem muitos membros ou se algum dos membros da estrutura é uma matriz ou vetor com um número significativo de elementos, a velocidade de execução do programa pode ser completamente degradada quando chamamos uma função e passamos seus argumentos utilizando o valor dos argumentos.

A solução é passar para a função chamada somente o endereço da estrutura argumento, endereço que é armazenado em um ponteiro para a estrutura.

Quando um ponteiro para estrutura é passado como argumento de uma função, somente o endereço da estrutura (i.e., somente 8 bytes no DevC++) é passado para a função chamada. Ou seja, somente o endereço da estrutura é empilhado no *stack*, resultando em uma chamada de função extremamente rápida.

Além disto, dentro do escopo da função chamada estaremos referenciando a estrutura argumento através de um endereço armazenado em um argumento ponteiro para a estrutura referenciada (e não através de uma cópia da estrutura no *stack*). Portanto, a função chamada terá condições de diretamente modificar o conteúdo dos membros da estrutura argumento referenciada através do ponteiro.

Este método que chama uma função e passa para a função chamada somente o **endereço** da estrutura argumento (ou vetor, matriz, etc ...) é denominado de **passagem de argumentos por referência** e o método desaconselhável que passa um vetor/ matriz/estrutura como **valor** para o argumento de uma função é denominado de **passagem de argumentos por valor**.

Estruturas

Para determinar o endereço de uma variável de estrutura, o operador & é colocado antes do nome da variável, conforme segue:

```
struct saldo_bancario {
float saldo;
char nome[80];
} cliente;

struct saldo_bancario *ptr; /* declara um ponteiro para estruturas tipo (tag) 'saldo_bancário' */

ptr= &cliente; /* armazena em 'ptr' o endereço da variável de estrutura 'cliente' */
```

Para atribuir o valor do membro 'saldo' através do ponteiro 'ptr' é possível usar duas sintaxes:

```
float x; /* declara uma variável float x */

x= (*ptr).saldo  /* atribui a x o saldo de cliente */

x= ptr->saldo /* atribui a x o saldo de cliente */
```

A segunda sintaxe de atribuição faz uso do chamado **operador seta** e é universalmente preferida em relação à primeira. Ver função **Inicializa_alt()** no slide 139 do Cap II.6 e compare com a função **Inicializa()** no slide 138.

Seguem dois exemplos com estruturas para representação de números complexos e operações entre eles, um utilizando passagem do argumento-estrutura por referência e o outro utilizando passagem do argumento-estrutura por valor**.** Ambos utilizam **typedef** para definir tipos de dados representativos de números complexos a partir de uma estrutura.

Estruturas

O código fonte https://www.fccdecastro.com.br/CursoC&C++/C/CpxOpVal.c a seguir demonstra a passagem por valor do argumento-estrutura de uma função:

```c
/*****************************************************************************
* Efetua as operacoes [+ - * /] entre dois numeros complexos para efeito de
* demonstracao da passagem por valor do argumento-estrutura de uma funcao.
*****************************************************************************/
/*****************************************************************************
* HEADERS:
*****************************************************************************/
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>
#include<math.h>
#include<conio.h>
/*****************************************************************************
* DATA TYPE STRUCTURES:
*****************************************************************************/
typedef struct ComplexRect{
double Re;
double Im;
}CPX_R;

typedef struct ComplexRectAndPolar{
double Re;
double Im;
double mag;
double ang;
}CPX_RP;
```

```
27  /*****************************************************************************
28  * MACROS:
29  *****************************************************************************/
30  static double sqrarg;
31  #define SQR(a) ((sqrarg=(a)) == 0.0 ? 0.0 : sqrarg*sqrarg)
32  /*****************************************************************************
33  * PROGRAM DEFINITIONS:
34  *****************************************************************************/
35  /*****************************************************************************
36  * FUNCTION PROTOTYPES:
37  *****************************************************************************/
38  CPX_RP Soma(CPX_R arg1,CPX_R arg2);
39  CPX_RP Subtracao(CPX_R arg1,CPX_R arg2);
40  CPX_RP Multiplicacao(CPX_R arg1,CPX_R arg2);
41  CPX_RP Divisao(CPX_R arg1,CPX_R arg2);
42  CPX_RP CoverteParaPolar(CPX_RP arg);
43  /*****************************************************************************
44  * main():
45  *****************************************************************************/
46  void main(void){
47  char op,sign;
48      CPX_R z1,z2;
49      CPX_RP zout;
```

```
50
51      do{
52      printf("\nQual operacao com numeros complexos eh desejada? [+ - * /] ");
53      op=getche();
54        }while(op!='+'&&op!='-'&&op!='*'&&op!='/');
55
56      puts("\n\nInforme dois numeros complexos Z1=R1+j*I1 e Z2=R2+j*I2");
57      puts("entrando na seguinte forma: R1 I1 R2 I2  ");
58      scanf("%lf %lf %lf %lf",&z1.Re,&z1.Im,&z2.Re,&z2.Im);
59
60      switch(op) {
61
62      case '+':
63      zout = Soma(z1,z2);
64      break;
65
66      case '-':
67      zout = Subtracao(z1,z2);
68      break;
69
70      case '*':
71      zout = Multiplicacao(z1,z2);
72      break;
73
74      case '/':
75      zout = Divisao(z1,z2);
76      break;
77     }
```

```
78
79      if(zout.Im<0){sign='-'; zout.Im= -(zout.Im);} /* sign dá o sinal da */
80      else{sign='+';}                                /* parte imaginária de */
81                          /* zout no formato da printf */
82
83      printf("\nZ1 %c Z2 = %lf %c j*%lf =>",op,zout.Re,sign,zout.Im);
84      printf(" modulo = %lf fase = %lf graus\n",zout.mag,zout.ang);
85
86  }
```

```
87  /*************************************************************************
88  * FUNC: CPX_RP Soma(CPX_R arg1,CPX_R arg2)
89  *
90  * DESC: Calcula arg1+arg2 e retorna o resultado em CPX_RP res.
91  *************************************************************************/
92  CPX_RP Soma(CPX_R arg1,CPX_R arg2)
93  {
94      CPX_RP res;
95
96      res.Re = arg1.Re + arg2.Re;
97      res.Im = arg1.Im + arg2.Im;
98      res = CoverteParaPolar(res);
99      return res;
100 }
101 /*************************************************************************
102 * FUNC: CPX_RP Subtracao(CPX_R arg1,CPX_R arg2)
103 *
104 * DESC: Calcula arg1-arg2 e retorna o resultado em CPX_RP res.
105 *************************************************************************/
106 CPX_RP Subtracao(CPX_R arg1,CPX_R arg2)
107 {
108     CPX_RP res;
109
110     res.Re = arg1.Re - arg2.Re;
111     res.Im = arg1.Im - arg2.Im;
112     res = CoverteParaPolar(res);
113     return res;
114 }
```

```
115 /*****************************************************************************
116 * FUNC: CPX_RP Multiplicacao(CPX_R arg1,CPX_R arg2)
117 *
118 * DESC: Calcula arg1*arg2 e retorna o resultado em CPX_RP res.
119 *****************************************************************************/
120 CPX_RP Multiplicacao(CPX_R arg1,CPX_R arg2)
121 {
122     CPX_RP res;
123
124     res.Re = (arg1.Re)*(arg2.Re)-(arg1.Im)*(arg2.Im);
125     res.Im = (arg1.Im)*(arg2.Re)+(arg1.Re)*(arg2.Im);
126
127        /* (a+jb)(c+jd) = (ac-bd) + j(bc+ad) */
128
129     res = CoverteParaPolar(res);
130     return res;
131 }
```

```
132 /*****************************************************************************
133 * FUNC: CPX_RP Divisao(CPX_R arg1,CPX_R arg2)
134 *
135 * DESC: Calcula arg1/arg2 e retorna o resultado em CPX_RP res.
136 *****************************************************************************/
137 CPX_RP Divisao(CPX_R arg1,CPX_R arg2)
138 {
139     CPX_RP res;
140     double den;
141
142     den = (arg2.Re*arg2.Re + arg2.Im*arg2.Im);
143
144     res.Re = (arg1.Re)*(arg2.Re)+(arg1.Im)*(arg2.Im);
145     res.Im = (arg1.Im)*(arg2.Re)-(arg1.Re)*(arg2.Im);
146
147     res.Re = res.Re/den;
148     res.Im = res.Im/den;
149
150     /* (a+jb)/(c+jd) = ((ac+bd) + j(bc-ad))/(c^2 + d^2) */
151
152     res = CoverteParaPolar(res);
153
154     return res;
155 }
```

```
156 /*****************************************************************************
157 * FUNC: CPX_RP CoverteParaPolar(CPX_RP arg)
158 *
159 * DESC: Converte para polar o argumento CPX_RP arg
160 *****************************************************************************/
161 CPX_RP CoverteParaPolar(CPX_RP arg)
162 {
163   CPX_RP res;
164   res.mag = sqrt((arg.Re)*(arg.Re)+(arg.Im)*(arg.Im));
165   res.ang =(180/M_PI)*atan2(arg.Im,arg.Re);
166   res.Re = arg.Re;
167   res.Im = arg.Im;
168   return res;
169  }
```

Execução na tela do console:

Qual operacao com numeros complexos eh desejada? [+ - * /] +

Informe dois numeros complexos Z1=R1+j*I1 e Z2=R2+j*I2
entrando na seguinte forma: R1 I1 R2 I2
1 2 3 4

Z1 + Z2 = 4.000000 + j*6.000000 => modulo = 7.211103 fase = 56.309932 graus

Qual operacao com numeros complexos eh desejada? [+ - * /] -

Informe dois numeros complexos Z1=R1+j*I1 e Z2=R2+j*I2
entrando na seguinte forma: R1 I1 R2 I2
1 2 3 4

Z1 - Z2 = -2.000000 - j*2.000000 => modulo = 2.828427 fase = -135.000000 graus

Qual operacao com numeros complexos eh desejada? [+ - * /] *

Informe dois numeros complexos Z1=R1+j*I1 e Z2=R2+j*I2
entrando na seguinte forma: R1 I1 R2 I2
1 2 3 4

Z1 * Z2 = -5.000000 + j*10.000000 => modulo = 11.180340 fase = 116.565051 graus

Qual operacao com numeros complexos eh desejada? [+ - * /] /

Informe dois numeros complexos Z1=R1+j*I1 e Z2=R2+j*I2
entrando na seguinte forma: R1 I1 R2 I2
1 2 3 4

Z1 / Z2 = 0.440000 + j*0.080000 => modulo = 0.447214 fase = 10.304846 graus

Estruturas

O código fonte https://www.fccdecastro.com.br/CursoC&C++/C/CpxOpRef.c a seguir demonstra a passagem por referência do argumento-estrutura de uma função:

```c
/*****************************************************************************
* Efetua as operacoes [+ - * /] entre dois numeros complexos para efeito de
* demonstracao da passagem por referencia do argumento-estrutura de uma funcao.
*****************************************************************************/
/*****************************************************************************
* HEADERS:
*****************************************************************************/
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>
#include<math.h>
#include<conio.h>
/*****************************************************************************
* DATA TYPE STRUCTURES:
*****************************************************************************/
typedef struct ComplexRect{
double Re;
double Im;
}CPX_R;

typedef struct ComplexRectAndPolar{
double Re;
double Im;
double mag;
double ang;
}CPX_RP;
```

```
27 /*****************************************************************************
28 * MACROS:
29 *****************************************************************************/
30 static double sqrarg;
31 #define SQR(a) ((sqrarg=(a)) == 0.0 ? 0.0 : sqrarg*sqrarg)
32 /*****************************************************************************
33 * PROGRAM DEFINITIONS:
34 *****************************************************************************/
35 /*****************************************************************************
36 * FUNCTION PROTOTYPES:
37 *****************************************************************************/
38 CPX_RP Soma(CPX_R *arg1,CPX_R *arg2);
39 CPX_RP Subtracao(CPX_R *arg1,CPX_R *arg2);
40 CPX_RP Multiplicacao(CPX_R *arg1,CPX_R *arg2);
41 CPX_RP Divisao(CPX_R *arg1,CPX_R *arg2);
42 CPX_RP CoverteParaPolar(CPX_RP *arg);
43 /*****************************************************************************
44 * main():
45 *****************************************************************************/
46 void main(void){
47 char op,sign;
48     CPX_R z1,z2;
49     CPX_RP zout;
```

```
50
51      do{
52      printf("\nQual operacao com numeros complexos eh desejada? [+ - * /] ");
53      op=getche();
54        }while(op!='+'&&op!='-'&&op!='*'&&op!='/');
55
56      puts("\n\nInforme dois numeros complexos Z1=R1+j*I1 e Z2=R2+j*I2");
57      puts("entrando na seguinte forma: R1 I1 R2 I2  ");
58      scanf("%lf %lf %lf %lf",&z1.Re,&z1.Im,&z2.Re,&z2.Im);
59
60      switch(op) {
61
62      case '+':
63      zout = Soma(&z1,&z2);
64      break;
65
66      case '-':
67      zout = Subtracao(&z1,&z2);
68      break;
69
70      case '*':
71      zout = Multiplicacao(&z1,&z2);
72      break;
73
74      case '/':
75      zout = Divisao(&z1,&z2);
76      break;
77     }
```

```
78
79      if(zout.Im<0){sign='-'; zout.Im= -(zout.Im);} /* sign dá o sinal da */
80      else{sign='+';}                               /* parte imaginária de */
81                      /* zout no formato da printf */
82
83      printf("\nZ1 %c Z2 = %lf %c j*%lf =>",op,zout.Re,sign,zout.Im);
84      printf(" modulo = %lf fase = %lf graus\n",zout.mag,zout.ang);
85
86  }
```

```
87  /*****************************************************************************
88  * FUNC: CPX_RP Soma(CPX_R *arg1,CPX_R *arg2)
89  *
90  * DESC: Calcula arg1+arg2 e retorna o resultado em CPX_RP res.
91  *****************************************************************************/
92  CPX_RP Soma(CPX_R *arg1,CPX_R *arg2)
93  {
94      CPX_RP res;
95
96      res.Re = arg1->Re + arg2->Re;
97      res.Im = arg1->Im + arg2->Im;
98      res = CoverteParaPolar(&res);
99      return res;
100 }
101 /*****************************************************************************
102 * FUNC: CPX_RP Subtracao(CPX_R *arg1,CPX_R *arg2)
103 *
104 * DESC: Calcula arg1-arg2 e retorna o resultado em CPX_RP res.
105 *****************************************************************************/
106 CPX_RP Subtracao(CPX_R *arg1,CPX_R *arg2)
107 {
108     CPX_RP res;
109
110     res.Re = arg1->Re - arg2->Re;
111     res.Im = arg1->Im - arg2->Im;
112     res = CoverteParaPolar(&res);
113     return res;
114 }
```

```
115 /*************************************************************************
116 * FUNC: CPX_RP Multiplicacao(CPX_R *arg1,CPX_R *arg2)
117 *
118 * DESC: Calcula arg1*arg2 e retorna o resultado em CPX_RP res.
119 *************************************************************************/
120 CPX_RP Multiplicacao(CPX_R *arg1,CPX_R *arg2)
121 {
122     CPX_RP res;
123
124     res.Re = (arg1->Re)*(arg2->Re)-(arg1->Im)*(arg2->Im);
125     res.Im = (arg1->Im)*(arg2->Re)+(arg1->Re)*(arg2->Im);
126
127        /* (a+jb)(c+jd) = (ac-bd) + j(bc+ad) */
128
129     res = CoverteParaPolar(&res);
130     return res;
131 }
```

```
132 /*****************************************************************************
133 * FUNC: CPX_RP Divisao(CPX_R *arg1,CPX_R *arg2)
134 *
135 * DESC: Calcula arg1/arg2 e retorna o resultado em CPX_RP res.
136 *****************************************************************************/
137 CPX_RP Divisao(CPX_R *arg1,CPX_R *arg2)
138 {
139     CPX_RP res;
140     double den;
141
142     den = (arg2->Re*arg2->Re + arg2->Im*arg2->Im);
143
144     res.Re = (arg1->Re)*(arg2->Re)+(arg1->Im)*(arg2->Im);
145     res.Im = (arg1->Im)*(arg2->Re)-(arg1->Re)*(arg2->Im);
146
147     res.Re = res.Re/den;
148     res.Im = res.Im/den;
149
150     /* (a+jb)/(c+jd) = ((ac+bd) + j(bc-ad))/(c^2 + d^2) */
151
152     res = CoverteParaPolar(&res);
153
154     return res;
155 }
```

```
156 /*****************************************************************************
157 * FUNC: CPX_RP CoverteParaPolar(CPX_RP *arg)
158 *
159 * DESC: Converte para polar o argumento CPX_RP arg
160 *****************************************************************************/
161 CPX_RP CoverteParaPolar(CPX_RP *arg)
162 {
163   CPX_RP res;
164   res.mag = sqrt((arg->Re)*(arg->Re)+(arg->Im)*(arg->Im));
165   res.ang =(180/M_PI)*atan2(arg->Im,arg->Re);
166   res.Re = arg->Re;
167   res.Im = arg->Im;
168   return res;
169  }
```

Execução na tela do console:

Qual operacao com numeros complexos eh desejada? [+ - * /] +

Informe dois numeros complexos Z1=R1+j*I1 e Z2=R2+j*I2
entrando na seguinte forma: R1 I1 R2 I2
1 2 3 4

Z1 + Z2 = 4.000000 + j*6.000000 => modulo = 7.211103 fase = 56.309932 graus

Qual operacao com numeros complexos eh desejada? [+ - * /] -

Informe dois numeros complexos Z1=R1+j*I1 e Z2=R2+j*I2
entrando na seguinte forma: R1 I1 R2 I2
1 2 3 4

Z1 - Z2 = -2.000000 - j*2.000000 => modulo = 2.828427 fase = -135.000000 graus

Qual operacao com numeros complexos eh desejada? [+ - * /] *

Informe dois numeros complexos Z1=R1+j*I1 e Z2=R2+j*I2
entrando na seguinte forma: R1 I1 R2 I2
1 2 3 4

Z1 * Z2 = -5.000000 + j*10.000000 => modulo = 11.180340 fase = 116.565051 graus

Qual operacao com numeros complexos eh desejada? [+ - * /] /

Informe dois numeros complexos Z1=R1+j*I1 e Z2=R2+j*I2
entrando na seguinte forma: R1 I1 R2 I2
1 2 3 4

Z1 / Z2 = 0.440000 + j*0.080000 => modulo = 0.447214 fase = 10.304846 graus

Matrizes dentro de estruturas:

```
int dado;

struct Cotas {
int Mat[1000][2000]; /* matriz de 1000x2000 inteiros representando as cotas altimétricas de um terreno retangular */
float b; /* media das cotas do terreno armazenada em Mat */
}TerrenoA, TerrenoB, TerrenoC; /* declara 3 variáveis de estruturas */

dado = TerrenoA.Mat[300][700]; /* atribui a 'dado' o valor do elemento Mat[300][700] da matriz Mat[][] que eh
membro da estrutura 'Cotas' instanciada pela variável de estrutura 'TerrenoA' */
```

Estruturas dentro de estruturas (estruturas aninhadas):

O código fonte https://www.fccdecastro.com.br/CursoC&C++/C/Cadastro.c a seguir demonstra o uso de estruturas aninhadas para formar um cadastro de clientes:

```
1  /*************************************************************************
2  * Demonstra o uso de estruturas aninhadas para formar um cadastro de clientes.
3  *************************************************************************/
4  /*************************************************************************
5  * HEADERS:
6  *************************************************************************/
7  #include<conio.h>
8  #include<stdio.h>
9  #include<ctype.h>
10 #include<string.h>
11 #include<stdlib.h>
12 /*************************************************************************
13 * PROGRAM DEFINITIONS:
14 *************************************************************************/
15 #define NUM_CLIENTES 4 // numero de elementos no vetor cliente[]
16 #define SIM 1
17 #define NAO 0
```

Estruturas

```
18 /*****************************************************************************
19 * DATA TYPE STRUCTURES:
20 *****************************************************************************/
21 typedef struct Parentes {
22 char pai[80];
23 char mae[80];
24 } PARENTES;
25
26 typedef struct Cadastro {
27 char nome[80];
28 char cidade[40];
29 PARENTES p; // estrutura PARENTES aninhada na estrutura CADASTRO
30 unsigned patrimonio;
31 unsigned serasa :1; // membro 'serasa' eh um bit-field de 1 bit - veremos adiante
32 } CADASTRO;
33 /*****************************************************************************
34 * FUNCTION PROTOTYPES:
35 *****************************************************************************/
36 void Inicializa(CADASTRO *Cliente,unsigned *Indice,char *Nome,char *Cidade,unsigned
37 void Inicializa_alt(CADASTRO *Cliente,unsigned *Indice,char *Nome,char *Cidade,unsig
38 void Imprime(CADASTRO *Cliente,unsigned NumClientes);
```

```
39  /*****************************************************************************
40  * main():
41  *****************************************************************************/
42  void main(void)
43  {
44  CADASTRO *cliente; /* declara 'cliente' como um ponteiro p/ o tipo de dado CADASTRO */
45  unsigned indice=0;
46
47  /* Aloca memoria para NUM_CLIENTES elementos no vetor CADASTRO cliente[] */
48  cliente=(CADASTRO *)malloc(NUM_CLIENTES*sizeof(CADASTRO));
49  if(!cliente){puts("Nao consigo alocar memoria!");exit(1);}
50
51  /* Inicializa NUM_CLIENTES elementos e armazena no vetor CADASTRO cliente[] */
52  Inicializa(cliente,&indice,"Gervasio","P.Alegre",40000,SIM,"Gervaziao","Gervazina");
53  Inicializa(cliente,&indice,"Helvirio","S.Paulo",55000,NAO,"Helviriao","Helvirina");
54  Inicializa_alt(cliente,&indice,"Algurbio","Recife",35000,SIM,"Algurbiao","Algurbina"); // alternativo
55  Inicializa(cliente,&indice,"Baiano","Salvador",25000,NAO,"Baianao","Baianinha");
56
57  /* Imprime indice final p/ efeito de verificacao de 'indice final = NUM_CLIENTES?' */
58  printf("Indice final = %u\tNumero de elementos do vetor cliente[] = %u\n\n",indice,NUM_CLIENTES);
59
60  /* Imprime na tela do console os NUM_CLIENTES elementos armazenados no vetor CADASTRO cliente[] */
61  Imprime(cliente,indice);
62  }
```

```
63  /*****************************************************************************
64  * FUNC: void Inicializa(CADASTRO *Cliente,unsigned *Indice,char *Nome,char *Cidade,unsigned Patrimonio,cl
65  *
66  * DESC: Recebe como argumento os valores para os membros de CADASTRO *Cliente e atribui estes
67  *       valores-argumento aos membros de CADASTRO *Cliente. Incrementa *indice a cada chamada da funcao.
68  *****************************************************************************/
69  void Inicializa(CADASTRO *Cliente,unsigned *Indice,char *Nome,char *Cidade,unsigned Patrimonio,char Seras
70  {
71  strcpy(Cliente[*Indice].nome,Nome);
72  strcpy(Cliente[*Indice].cidade,Cidade);
73  Cliente[*Indice].patrimonio=Patrimonio;
74  Cliente[*Indice].serasa=Serasa;
75  strcpy(Cliente[*Indice].p.pai,NomePai);
76  strcpy(Cliente[*Indice].p.mae,NomeMae);
77  (*Indice)++; // incrementa o valor *Indice
78  }
```

```
79  /*************************************************************************
80  * FUNC: void Inicializa_alt(CADASTRO *Cliente,unsigned *Indice,char *Nome,char *Cidade,unsigned Patrimon
81  *
82  * DESC: Mostra forma alternativa de acessar os membros de um vetor de estruturas definido por
83  *       um ponteiro p/ estruturas cujo endereço apontado é gerado por malloc()/calloc(). Ver
84  *       slide 115 do Cap II.6.
85  *       Recebe como argumento os valores para os membros de CADASTRO *Cliente e atribui estes
86  *       valores-argumento aos membros de CADASTRO *Cliente. Incrementa *indice a cada chamada da funcao.
87  *************************************************************************/
88  void Inicializa_alt(CADASTRO *Cliente,unsigned *Indice,char *Nome,char *Cidade,unsigned Patrimonio,char S
89  {
90  strcpy((&Cliente[*Indice])->nome,Nome);
91  strcpy((&Cliente[*Indice])->cidade,Cidade);
92  (&Cliente[*Indice])->patrimonio=Patrimonio;
93  (&Cliente[*Indice])->serasa=Serasa;
94  strcpy((&Cliente[*Indice])->p.pai,NomePai);
95  strcpy((&Cliente[*Indice])->p.mae,NomeMae);
96  (*Indice)++; // incrementa o valor *Indice
97  }
```

```
98  /*****************************************************************************
99  * FUNC: void Imprime(CADASTRO *Cliente,unsigned NumClientes)
100 *
101 * DESC: Imprime o vetor Cliente[] na tela do console
102 *****************************************************************************/
103 void Imprime(CADASTRO *Cliente,unsigned NumClientes)
104 {
105 unsigned register i;
106 for(i=0;i<NumClientes;i++){
107 printf("%u) %s, morador de %s, tem patrimonio de R$ %u,00.\n",i+1, Cliente[i].nome,Cliente[i].cidade,Cliente[i].patrimonio);
108 if(Cliente[i].serasa==SIM)printf("%s estah incluido na Serasa.",Cliente[i].nome);
109 else printf("%s nao estah incluido na Serasa.",Cliente[i].nome);
110 printf(" O pai chama-se %s e a mae %s.\n\n",Cliente[i].p.pai,Cliente[i].p.mae);
111 }
112 }
```

Execução na tela do console:

```
Indice final = 4        Numero de elementos do vetor cliente[] = 4

1) Gervasio, morador de P.Alegre, tem patrimonio de R$ 40000,00.
Gervasio estah incluido na Serasa. O pai chama-se Gervaziao e a mae Gervazina.

2) Helvirio, morador de S.Paulo, tem patrimonio de R$ 55000,00.
Helvirio nao estah incluido na Serasa. O pai chama-se Helviriao e a mae Helvirina.

3) Algurbio, morador de Recife, tem patrimonio de R$ 35000,00.
Algurbio estah incluido na Serasa. O pai chama-se Algurbiao e a mae Algurbina.

4) Baiano, morador de Salvador, tem patrimonio de R$ 25000,00.
Baiano nao estah incluido na Serasa. O pai chama-se Baianao e a mae Baianinha.
```

Bit-fields

Frequentemente os membros de uma estrutura necessitam de poucos bits para serem numericamente representados adequadamente, como, por exemplo, estruturas que representam a configuração do hardware de um dispositivo ou de um controlador/interface de uma máquina (ver https://en.wikipedia.org/wiki/Bit_field).

Um exemplo de uso de *bit-fields* não necessariamente ligado a hardware é a situação em que uma estrutura tem 3 membros: dia, mês e ano. Os 12 meses do ano necessitam de apenas 4 bits para serem representados dado que $2^4 = 16$, i.e., com 4 bits é possível contar de 0 a 15. Os 31 dias do mês necessitam de apenas 5 bits para serem representados dado que $2^5 = 32$, i.e., com 5 bits é possível contar de 0 a 31.

Mesmo se usássemos um dado padrão do tipo **char** para representar cada um dos membros mês e dia, ainda assim estaríamos gastando 8 bits, que é quase o dobro dos bits necessários e que permitiria contar de 0 a $2^8 - 1$, i.e., de 0 a 255 – um intervalo de contagem totalmente desnecessário e que ocupa quase o dobro da memória.

No entanto, se usarmos *bit-fields* (campos de bits) de 4 e 5 bits para representar respectivamente os membros mês e dia da estrutura, o problema de gasto desnecessário de memória fica minimizado. Obviamente este problema de memória é insignificante para o caso de uma única estrutura, mas se torna incisivo na situação em que temos um vetor de estruturas de centenas de milhares de elementos.

A forma geral da declaração de uma estrutura cujos membros são *bit-fields* é conforme segue:

```
struct nome_do_tipo_estrutura {
    tipo nome_1   : numero_de_bits;
    tipo nome_2   : numero_de_bits;
                  .
                  .
                  .
    tipo nome_N   : numero_de_bits;
  } lista_de_variáveis_de estrutura;
```

Cada membro que é um *bit-field* deve ser declarado como **unsigned** ou como **int**. *Bit-Fields* de tamanho 1 devem ser declarados como **unsigned**, já que um único bit não pode ter sinal. Podemos misturar membros normais da estrutura com membros do tipo *bit-field* , conforme vimos no slide 136 do Cap II.6.

Bit-fields

O código fonte https://www.fccdecastro.com.br/CursoC&C++/C/Bitfield.c a seguir demonstra o uso de *bit-fields*:

```c
/* Demonstra o uso de bit fields */
#include <stdio.h>
typedef struct Data {
    unsigned dia :5; // dia varia de 1 a 31 entao 5 bits (0 a 31) eh adequado
    unsigned mes :4; // mes varia de 1 a 12 entao 4 bits (0 a 15) eh adequado
    unsigned ano;
}DATA;

typedef struct DataX {
    unsigned dia;
    unsigned mes;
    unsigned ano;
}DATAX;

void main(void)
{
DATA Dt  = { 31, 12, 2023 }; // declara e inicializa a variavel de estrutura Dt do tipo DATA
DATAX Dtx = { 31, 12, 2024 }; // declara e inicializa a variavel de estrutura Dtx do tipo DATAX
printf("A memoria ocupada por Dt eh %u bytes.\n",sizeof(Dt));
printf("A memoria ocupada por Dtx eh %u bytes.\n",sizeof(Dtx));
printf("A data armazenada em Dt eh %u/%u/%u.\n", Dt.dia, Dt.mes, Dt.ano);
printf("A data armazenada em Dtx eh %u/%u/%u.\n", Dtx.dia, Dtx.mes, Dtx.ano);
}
```

Execução na tela do console:

```
A memoria ocupada por Dt eh 8 bytes.
A memoria ocupada por Dtx eh 12 bytes.
A data armazenada em Dt eh 31/12/2023.
A data armazenada em Dtx eh 31/12/2024.
```

Unions

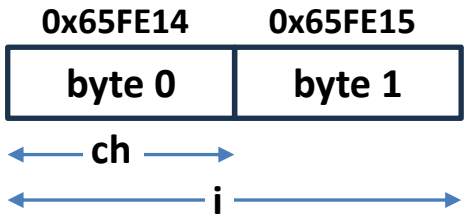
Uma **union** é um segmento na memória que armazena diferentes tipos de variáveis colocalizadas no mesmo endereço inicial de memória. A declaração de uma **union** é similar a de uma **struct**, conforme exemplo de declaração mostrado abaixo:

```
union CharInt {
short int i; // membro i da union Charint
char ch; // membro ch da union Charint
};
```

Assim como em estruturas, a declaração acima não declara qualquer variável e portanto não ocupa memória. Para declarar uma variável de **union** coloca-se o nome da variável no fim da declaração da **union** ou declara-se a variável em separado, conforme exemplo que segue. Por exemplo, para declarar uma variável da **union** do tipo **Charint** denominada '**Conv**' fazemos a seguinte declaração:

```
union CharInt Conv;
```

Na variável **Conv** o membro **short int i** e o membro **char ch** compartilham o mesmo segmento na memória. Note que **i** ocupa 2 bytes e **ch** ocupa 1 byte. A figura a seguir mostra como **i** e **ch** compartilham o mesmo endereço inicial na memória:



Portanto, quando uma variável de **union** é declarada o compilador aloca um segmento de memória grande o suficiente para armazenar o membro cujo tipo ocupa o maior tamanho em bytes na **union**, conforme figura acima.

Para acessar um membro de uma **union** usa-se a mesma sintaxe de estruturas: os operadores ponto e seta. Quando se opera diretamente na **union**, usa-se o operador ponto. Se a variável de **union** é acessada por meio de um ponteiro, usa-se o operador seta. Por exemplo, para atribuir o valor 10 ao membro **i** de **Conv** declaramos:

```
Conv.i=10;
```

Unions

Unions são frequentemente usadas quando são necessários conversões de tipo de dados não realizáveis com um **cast** simples, uma vez que unions permitem que se "visualize" de diversas maneiras uma mesma região na memória. Por exemplo, a função **putw()**, com protótipo definido em stdio.h, escreve a representação binária de um inteiro de 2 bytes (= word = 16 bits) para um arquivo em disco apontado por **FILE *fp**. Existem diversas maneiras de codificar esta função. Abaixo é mostrada uma codificação de **putw()** baseada em uma variável de **union** e na função **putc()** (ver slide 67 do Cap II.5). Primeiro é declarada uma **union** cujos dois membros são um **short int i** e um vetor **char ch[2]** , ambos os membros ocupando 2 bytes:

```
union pw {
short int i;
char ch[2];
 };
```

E daí a função **putw()** pode ser definida usando-se a variável de union **'word'** do tipo **union pw**, conforme abaixo:

```
void putw(union pw word, FILE *fp)
 {
putc(word.ch[0],fp); /* escreve em fp a  1ª metade  de word*/
putc(word.ch[1],fp); /* escreve em fp a 2ª metade de word */
}
```

Note que o valor inteiro a ser gravado em disco deve ser previamente atribuído ao membro **word.i**.

Unions

O código fonte https://www.fccdecastro.com.br/CursoC&C++/C/DEC2BIN.c a seguir demonstra o uso de uma **union** para converter um numero decimal para representação binária:

```c
/* dec2bin.c: converte de decimal para binario */
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<string.h>

typedef struct BitUShort {
unsigned b0  :1;
unsigned b1  :1;
unsigned b2  :1;
unsigned b3  :1;
unsigned b4  :1;
unsigned b5  :1;
unsigned b6  :1;
unsigned b7  :1;
unsigned b8  :1;
unsigned b9  :1;
unsigned b10  :1;
unsigned b11  :1;
unsigned b12  :1;
unsigned b13  :1;
unsigned b14  :1;
unsigned b15  :1;
}BIT_USHORT;

```

```
26  typedef union UShort_BitUShort {
27  unsigned short i;
28  BIT_USHORT b;
29  }USHORT_BITUSHORT;
30
31  int main(int argc,char *argv[])
32  {
33  USHORT_BITUSHORT Num;
34  char Bits[17]=""; // inicializa a string com o '\0';
35
36  if(argc!=2){puts("DEC2BIN: NumeroDecimal");exit(1);}
37
38  Num.i=atoi(argv[1]); /* converte argv[1] p/ unsigned e
39                          atribui o valor a Num.i */
40
```

```
41  /* le os valores de cada um dos 16 Bits em Num.b e concatena
42   o valor do bit na string Bits[17] */
43  if(Num.b.b15)strcat(Bits,"1");else strcat(Bits,"0");
44  if(Num.b.b14)strcat(Bits,"1");else strcat(Bits,"0");
45  if(Num.b.b13)strcat(Bits,"1");else strcat(Bits,"0");
46  if(Num.b.b12)strcat(Bits,"1");else strcat(Bits,"0");
47  if(Num.b.b11)strcat(Bits,"1");else strcat(Bits,"0");
48  if(Num.b.b10)strcat(Bits,"1");else strcat(Bits,"0");
49  if(Num.b.b9)strcat(Bits,"1");else strcat(Bits,"0");
50  if(Num.b.b8)strcat(Bits,"1");else strcat(Bits,"0");
51  if(Num.b.b7)strcat(Bits,"1");else strcat(Bits,"0");
52  if(Num.b.b6)strcat(Bits,"1");else strcat(Bits,"0");
53  if(Num.b.b5)strcat(Bits,"1");else strcat(Bits,"0");
54  if(Num.b.b4)strcat(Bits,"1");else strcat(Bits,"0");
55  if(Num.b.b3)strcat(Bits,"1");else strcat(Bits,"0");
56  if(Num.b.b2)strcat(Bits,"1");else strcat(Bits,"0");
57  if(Num.b.b1)strcat(Bits,"1");else strcat(Bits,"0");
58  if(Num.b.b0)strcat(Bits,"1");else strcat(Bits,"0");
59
60  /* imprime no console o valor de Num.i e a correspondente
61  palavra binaria armazenada na string Bits[17] */
62  printf("Numero decimal %u em binario eh %s.",Num.i,Bits);
63
64  return 0;
65  }
```

Execução na tela do console:

C:\Exemplos>dec2bin 3

Numero decimal 3 em binario eh 0000000000000011.

Unions

O código fonte https://www.fccdecastro.com.br/CursoC&C++/C/BIN2DEC.c a seguir demonstra a conversão de um número binário para representação decimal. O programa não usa unions (usa operadores lógicos bit a bit) – este código fonte é aqui apresentado apenas como um complemento ao código fonte decbin.c mostrado no slide 147.

```
1  /* bin2dec.c: converte de binario para decimal */
2  #include<stdio.h>
3  #include<stdlib.h>
4  #include<math.h>
5  #include<string.h>
6
7  void main(int argc,char *argv[])
8  {
9  unsigned n=0;
10 int i;
11 unsigned lsb;
12
13 if(argc!=2){puts("BIN2DEC: NumeroBinario");exit(1);}
14
```

```
15  lsb=strlen(argv[1])-1; /* lsb=>bit menos significativo a ser considerado em
16                            argv[1], isto eh, o mais a direita em argv[1] */
17
18  for(i=lsb;i>=0;i--){
19
20      switch(argv[1][i]){
21
22          case '1': n=n|(1<<(lsb-i)); /* "acrescenta" (OR = |) a n o numero 1 deslocado */
23          break;                      /* de lsb-i bits p/ esquerda */
24
25          case '0':
26          break;      /* nao faz nada porque inicialmente fizemos n=0 */
27
28          default:
29          puts("O numero nao eh binario!");
30          exit(1);
31      }
32  }
33  printf("Numero binario %s em decimal eh %u.",argv[1],n);
34  }
```

Execução na tela do console:

C:\Exemplos> bin2dec 0000000000000011
Numero binario 0000000000000011 em decimal eh 3.

Enumerations

Enumeração ou **enum** em C é um tipo particular de dado definido pelo programador que consiste de constantes inteiras que recebem um nome associado ao que o valor das constantes representam. O uso de **enum** em C para designar valores inteiros aumenta a clareza do código fonte, facilitando a leitura e interpretação do mesmo. A sintaxe é:

enum nome_da_enumeracao{int_const1, int_const2, int_const3, …. int_constN};

Por exemplo: **enum Carros{Audi, BMW, Ferrari, Aston_Martin, Mercedes_Benz};**

Na declaração acima os valores inteiros *default* associados aos nomes são Audi=0, BMW=1, Ferrari=2, Aston_Martin =3, e Mercedes_Benz=4. É perfeitamente possível mudar os valores atribuídos a cada nome em uma **enum**, conforme exemplo que segue:

```
enum Carros{Audi=1,
            BMW=3,
            Ferrari=0,
            Aston_Martin=2,
            Mercedes_Benz=4};
```

O código fonte https://www.fccdecastro.com.br/CursoC&C++/C/READPAR.c que segue no próximo slide utiliza um tipo de dado **ERR** definido por **typedef** na linha 39 que é uma enumeração com os possíveis tipos de erro mais significativos que ocorrem na execução o programa. O tipo de dado **ERR** é usado como argumento da função **void Quit(ERR err, char *name)** na linha 130, função que faz o tratamento unificado dos possíveis erros, imprime na tela do console a *string* descritora do tipo de erro que ocorreu e encerra a execução do programa. O tratamento unificado dos erros feito por **Quit()** evita que o mesmo tipo de erro precise que as mesmas *strings* descritoras do erro tenham que ser declaradas no escopo de funções diferentes do programa, assim economizando memória.

Especificamente, o código fonte READPAR.c utiliza um tipo de dado **PARFR** definido por **typedef** na linha 53 que é uma estrutura que reúne um conjunto de membros com a finalidade de ler um arquivo de parâmetros para um programa (no exemplo em questão o arquivo de parâmetros é https://www.fccdecastro.com.br/CursoC&C++/C/param0.txt), e que evita o incômodo de termos de digitar muitos argumentos na linha de comando (ver slides 58 a 62) quando um programa necessita muitos parâmetros. Adicionalmente, READPAR.c verifica a conformidade da formatação e a coerência de valores dos parâmetros gravados no arquivo de parâmetros.

Enumerations

```
1  /*************************************************************************
2  * Read and handles a specific parameter file (param0.txt is the current example)
3  *************************************************************************/
4  /*************************************************************************
5  * HEADERS:
6  *************************************************************************/
7  #include<dos.h>
8  #include<stdio.h>
9  #include<stdlib.h>
10 #include<ctype.h>
11 #include<math.h>
12 #include<conio.h>
13 #include<string.h>
14 /*************************************************************************
15 * PROGRAM DEFINITIONS:
16 *************************************************************************/
17 #define BUFSIZE 0x8000 /* in&out file buffer size */
18 #define MAXLINESIZE 8192 /* max line size for  GetSheetNumLines() and NumberOfFl
19 #define MAXARGLEN 128   /* max length of a alphanumeric arg */
20 #define MAXFLOAT 3.4E38 /* 32 bits DJGPP max float */
21 #define NFLOATLINES 10 /* desired number of float lines in parameter file */
22 #define NALPHALINES 8 /* desired number of alphanumeric lines in parameter file
```

Enumerations

```
23  /*****************************************************************************
24  * COMPILING DIRECTIVES:
25  *****************************************************************************/
26  #pragma GCC diagnostic ignored "-Wwrite-strings"
27  /*****************************************************************************
28  * MACROS:
29  *****************************************************************************/
30  #define SWAP(a,b) temp=(a);(a)=(b);(b)=temp;
31  char *tempp;
32  #define SWAPP(a,b) tempp=(a);(a)=(b);(b)=tempp;
33  #define SIGN(a,b) ((b) >= 0.0 ? fabs(a) : -fabs(a))
34  static double aulimarg; /* absolute maximum upper limit */
35  #define AULIM(a) ( fabs(aulimarg=(a)) >= MAXFLOAT ? ( aulimarg< 0.0 ? (-MAXFLO
```

Enumerations

```
36  /*****************************************************************************
37  * DATA TYPE STRUCTURES:
38  *****************************************************************************/
39  typedef enum {
40      NO_ERROR,           /*  " ", */
41      MEMORY_ERR,         /*  "Not enough memory", */
42      READ_OPEN_ERR,      /*  "I can't find the file", */
43      READ_ERR,           /*  "Error reading file", */
44      WRITE_OPEN_ERR,     /*  "I can't open the file", */
45      WRITE_ERR,          /*  "I can't write the file", */
46      BUFFER_ERR,         /*  "I can't attach buffer to file" */
47      MAXIT_ERR,          /*  "Exceeded max number of iterations in function ", */
48      CMDLN_ERR,          /*  "Command line error", */
49      PARF_ERR            /*  "Wrong parameter file", */
50      /* ------------------------------------------------*/
51  }ERR;
```

Enumerations

```
52
53  typedef struct ParamFileReader{
54  char *ParameterFile; /* parameter file */
55  unsigned *FloatLines; /* the array of float lines indexes */
56  unsigned *AlphaLines; /* the array of alpha lines indexes */
57  unsigned NumFloatLines; /* the number of float lines */
58  unsigned NumAlphaLines; /* the number of alpha lines */
59  char **Sheet; /* parameters file sheet matrix */
60  char *BufferI; /* input buffer */
61  FILE *IFp;     /* input filepointer */
62  unsigned long NumLines;  /* the number of lines read */
63  unsigned LineMaxLength; /* Sheet max line length */
64  float **FloatDat; /* float data storage sheet */
65  char  **AlphaDat;  /* alpha data storage sheet */
66  unsigned *VectSize; /* the array that contains the size of each row of FloatDat */
67  }PARFR;
68
```

Enumerations

```
69  /*************************************************************************
70  * FUNCTION PROTOTYPES:
71  *************************************************************************/
72  void Quit(ERR err, char *name);
73  unsigned long GetSheetNumLines(FILE *Fp, unsigned *MaxLength);
74  char **CharMatCAlloc(unsigned long NRows,unsigned NCols);
75  unsigned NumberOfFloatsInLine(char *Line);
76  float *GetFloatVectFromLine(char *Line, unsigned *VectSize);
77  void InitParamFileReader(PARFR *Pfr,char *ParameterFile,unsigned *FloatLines,unsi
78  void ParamStorageID(PARFR *Pfr, unsigned LineNumber, unsigned *StorageIndex, char
79  /*************************************************************************
80  *                MAIN():
81  *************************************************************************/
82  int main(int argc, char *argv[])
83  {
84
85  unsigned register i,j;
86  unsigned Index;
87  char Type;
88
```

Enumerations

```
89  /* parameter file specification */
90  unsigned FloatLines[NFLOATLINES]={1,2,3,6,8,10,11,14,15,16};/* see param0.txt */
91  unsigned AlphaLines[NALPHALINES]={0,4,5,7,9,12,13,17}; /* see param0.txt */
92
93  PARFR Pfr;
94
95  /* Command Line */
96  if(argc!=2){
97  putch(7);
98  puts("Pread: ParameterFile");
99  exit(1);
100 }
101
102 /* initialize  ParamFileReader */
103 InitParamFileReader(&Pfr,argv[1],FloatLines,AlphaLines,NFLOATLINES,NALPHALINES);
104
105 /* scan parameter file */
106 for(i=0;i<Pfr.NumLines;i++){
107
108 /* identify the parameter/vector at the i_th line of the parameter file */
109 ParamStorageID(&Pfr,i,&Index,&Type);
```

```
110
111  /* write to stdout */
112  if(Type=='f'){
113  printf("Line%u: ",i);
114  for(j=0;j<Pfr.VectSize[Index];j++){
115  printf("%8.6g\t",Pfr.FloatDat[Index][j]);
116  }
117  printf("\n");
118  }
119  else{
120  printf("Line%u: ",i);
121  printf("%s\n",Pfr.AlphaDat[Index]);
122  }
123
124  }
125
126  /* an intentional error */
127  //ParamStorageID(&Pfr,Pfr.NumLines,&Index,&Type);
128  }
129
```

```
130 /*****************************************************************************
131  * FUNC: void Quit(int err, char *name)
132  *
133  * DESC: Prints Error message and quit.
134  *****************************************************************************/
135 void Quit(ERR err, char *name)
136 {
137     static char *ErrMess[] = {
138     " ",
139     "Not enough memory",
140     "I can't find the file",
141     "Error reading file",
142     "I can't open the file",
143     "I can't write the file",
144     "I can't attach buffer to file",
145     "Exceeded max number of iterations in function ",
146     "Command line error",
147     "Wrong parameter file",
148     /* ----------------------------------------------------*/
149     };
150
```

```
151     system("cls");
152
153     if (err != NO_ERROR) {
154
155         if(name[0]){
156         fprintf (stderr,"%s %s!\n", ErrMess[err], name);
157         putch(7);
158         exit (err);
159         }
160         else{
161         fprintf (stderr,"%s!\n", ErrMess[err]);
162         putch(7);
163         exit (err);
164         }
165     }
166     exit (0);
167 }
```

```
168 /*****************************************************************************
169  * FUNC: unsigned long GetSheetNumLines(FILE *Fp, unsigned *MaxLength)
170  *
171  * DESC: Return number of lines of the text sheet in file pointed by Fp.
172  *       MaxLength return the line max length, including the '\0'.
173  *****************************************************************************/
174 unsigned long GetSheetNumLines(FILE *Fp, unsigned *MaxLength)
175 {
176   char line[MAXLINESIZE];
177   unsigned register i;
178   unsigned long register Count;
179   unsigned Length;
180
181   *MaxLength=0;
182
183   Count=0;
184   while (fgets(line,MAXLINESIZE-2,Fp)){
185   if((Length=strlen(line))>*MaxLength)*MaxLength=Length;
186   Count++;
187   }
188
189   if(strlen(line)==1)Count--;/* to discount the last \n in file
190                                 for the case the user has typed it */
191
192   *MaxLength= (*MaxLength)+1; /* to handle the '\0' added by fgets() */
193
194   rewind(Fp);
195   return Count;
196 }
```

```
197 /*****************************************************************************
198 * FUNC: char **CharMatCAlloc(unsigned long NRows,unsigned NCols)
199 *
200 * DESC: Allocates memory for a NRows x NCols  char matrix.
201 *****************************************************************************/
202 char **CharMatCAlloc(unsigned long NRows,unsigned NCols)
203 {
204 char  *x;
205 char  **Mx;
206 unsigned register i;
207
208 x = (char  *)calloc((unsigned long)(NRows*NCols),sizeof(char));
209 if(!x)Quit(MEMORY_ERR,"");
210
211 Mx=(char  **)calloc((unsigned long)NRows,sizeof(char *));
212 if(!Mx)Quit(MEMORY_ERR,"");
213
214 for(i=0;i<NRows;i++){
215 Mx[i]=(char *)&x[i*NCols];
216 }
217
218 return Mx;
219 }
220
```

Enumerations

```
221 /*************************************************************************
222 * FUNC: unsigned NumberOfFloatsInLine(char *Line)
223 *
224 * DESC: Return the number of floats in Line, starting from the beginning.
225 *       Upon finding a non-readable character, the function ends the count.
226 *************************************************************************/
227 unsigned NumberOfFloatsInLine(char *Line)
228 {
229 char static Str[MAXLINESIZE];
230 char static Buf[MAXLINESIZE];
231
232 char *NonReadable;
233 double Val;
234 unsigned Count=0;
235
236 #if(0)
237 puts(Line);
238 #endif
239
240 while(1){
241 sprintf(Buf,"%s",Line);
242 if(Buf[0]=='\t'){Line++;continue;} /* found a tab */
243 if(Buf[0]==' '){Line++;continue;}  /* found a space */
244 if(!Buf[0])break; /* found a '\0' */
245 if(Buf[0]=='\n')break; /* found a '\n' */
246 sscanf(Buf,"%s",Str);
247 Val= strtod(Str, &NonReadable);
248
```

```
249  #if(0)
250  if(NonReadable[0]&&NonReadable[0]!='i')break; /* Check non-readable except 'i
251  #else
252  if(NonReadable[0])break; /* Check non-readable */
253  #endif
254
255  #if(0)
256  fprintf(stdout,"Val[%u]=%8.6lg%c\n",Count,Val,NonReadable[0]);
257  #endif
258
259  Line+=strlen(Str)+1;
260  Count++;
261  }
262
263  #if(0)
264  fprintf(stdout,"Line has %u floats.\n",Count);
265  #endif
266
267  return Count;
268  }
```

Enumerations

```
269 /*****************************************************************************
270 * FUNC: float *GetFloatVectFromLine(char *Line, unsigned *VectSize)
271 *
272 * DESC:  Convert the floats in Line into a float vector, starting from the beginning.
273 *        Upon finding a non-readable character, the function ends the count.
274 *        VectSize returns the size of the vector.
275 *****************************************************************************/
276 float *GetFloatVectFromLine(char *Line, unsigned *VectSize)
277 {
278 char static Str[MAXLINESIZE];
279 char static Buf[MAXLINESIZE];
280
281 char *NonReadable;
282 double Val;
283 unsigned NumFloats;
284 float *Vec;
285 unsigned Count=0;
286
287 *VectSize=NumberOfFloatsInLine(Line);
288 if(*VectSize==0)return NULL;
289
290 Vec=(float *)calloc(*VectSize,sizeof(float));
291 if(!Vec)Quit(MEMORY_ERR,"");
```

```
292
293 while(1){
294 sprintf(Buf,"%s",Line);
295 if(Buf[0]=='\t'){Line++;continue;} /* found a tab */
296 if(Buf[0]==' '){Line++;continue;}  /* found a space */
297 if(!Buf[0])break; /* found a '\0' */
298 if(Buf[0]=='\n')break; /* found a '\n' */
299 sscanf(Buf,"%s",Str);
300 Val= strtod(Str, &NonReadable);
301
302 #if(0)
303 if(NonReadable[0]&&NonReadable[0]!='i')break; /* Check non-readable except 'i'-> to handle comp
304 #else
305 if(NonReadable[0])break; /* Check non-readable */
306 #endif
307
308 Vec[Count]=AULIM(Val); /* check float absolute upper limit */
309 if(Vec[Count]!=(float)Val){putch(7);fprintf(stderr,"Exceeded float absolute value!");exit(1);}
310 Line+=strlen(Str)+1;
311 Count++;
312 }
313 return Vec;
314 }
```

Enumerations

```
315 /*************************************************************************
316 * FUNC: void InitParamFileReader(PARFR *Pfr,char *ParameterFile,unsigned *FloatLines,unsigned *Al
317 *
318 * DESC: Init parameter file reader.
319 *************************************************************************/
320 void InitParamFileReader(PARFR *Pfr,char *ParameterFile,unsigned *FloatLines,unsigned *AlphaLines
321
322 {
323
324 unsigned register i;
325
326 Pfr->ParameterFile=ParameterFile;
327 Pfr->FloatLines=FloatLines;
328 Pfr->AlphaLines=AlphaLines;
329 Pfr->NumFloatLines=NumFloatLines;
330 Pfr->NumAlphaLines=NumAlphaLines;
331
332 /* Alloc memory for disk I/O speed-up buffers */
333 Pfr->BufferI=(char *)calloc(BUFSIZE,sizeof(char));
334 if(!Pfr->BufferI)Quit(MEMORY_ERR,"");
335
336 /* Open parameter file */
337 if((Pfr->IFp=fopen(Pfr->ParameterFile,"rt"))==NULL)Quit(READ_OPEN_ERR,Pfr->ParameterFile);
338 if (setvbuf(Pfr->IFp, Pfr->BufferI, _IOFBF, BUFSIZE) != 0)Quit(BUFFER_ERR,Pfr->ParameterFile);
339
340 /* Get the total number of lines and the max line length */
341 Pfr->NumLines=GetSheetNumLines(Pfr->IFp,&Pfr->LineMaxLength);
```

```
342
343  /* check */
344  if(Pfr->NumLines!=(Pfr->NumFloatLines+Pfr->NumAlphaLines))Quit(PARF_ERR,Pfr->ParameterFile);
345
346  /* Alloc memory for Sheet */
347  Pfr->Sheet=CharMatCAlloc(Pfr->NumLines,Pfr->LineMaxLength);
348
349  /* Read sheet and store it into Sheet */
350  for(i=0;i<Pfr->NumLines;i++)fgets(Pfr->Sheet[i],Pfr->LineMaxLength,Pfr->IFp);
351
352  /* close parameter file */
353  fclose(Pfr->IFp);
354
355  /* Alloc memory for float FloatDat row pointers */
356  Pfr->FloatDat=(float **)calloc(Pfr->NumFloatLines,sizeof(float *));
357  if(!Pfr->FloatDat)Quit(MEMORY_ERR,"");
358
359  /* Alloc memory for alphanumeric data */
360  Pfr->AlphaDat=CharMatCAlloc(Pfr->NumAlphaLines,MAXARGLEN);
361
362  /* Alloc memory for the array that contains the size of each row of FloatDat */
363  Pfr->VectSize=(unsigned *)calloc(Pfr->NumFloatLines,sizeof(unsigned));
364  if(!Pfr->VectSize)Quit(MEMORY_ERR,"");
365
366  /* obtain float FloatDat sheet from char Sheet */
367  for(i=0;i<Pfr->NumFloatLines;i++){
368  Pfr->FloatDat[i]=GetFloatVectFromLine(Pfr->Sheet[Pfr->FloatLines[i]],&Pfr->VectSize[i]);
369  if(Pfr->FloatDat[i]==NULL)Quit(PARF_ERR,Pfr->ParameterFile);
370  }
```

```
371
372   /* obtain string AlphaDat sheet from first string in Sheet line */
373   for(i=0;i<Pfr->NumAlphaLines;i++){
374   sscanf(Pfr->Sheet[Pfr->AlphaLines[i]],"%s",Pfr->AlphaDat[i]);
375   }
376
377   }
```

Enumerations

```
378 /*****************************************************************************
379 * FUNC: void ParamStorageID(PARFR *Pfr, unsigned LineNumber, unsigned *StorageIndex, char *Type)
380 *
381 * DESC: StorageIndex return the index in respective (FloatDat/AlphaDat)
382 *       storage sheet that corresponds to line LineNumber.
383 *
384 *       Type returns the respective data type: 'f'->float 'a'->alpha
385 *****************************************************************************/
386 void ParamStorageID(PARFR *Pfr, unsigned LineNumber, unsigned *StorageIndex, char *Type)
387 {
388
389 unsigned register i;
390 int p=-1;
391 char ParType;
392
393 //printf("LineNumber=%u\n",LineNumber);
394
395 /* scan FloatLines index array */
396 for(i=0;i<Pfr->NumFloatLines;i++){
397 if(Pfr->FloatLines[i]==LineNumber){p=i;ParType='f';break;}
398 }
399
400 /* if not found, scan AlphaLines index array */
401 if(p==-1){
402 for(i=0;i<Pfr->NumAlphaLines;i++){
403 if(Pfr->AlphaLines[i]==LineNumber){p=i;ParType='a';break;}
404 }
```

```
405
406 }
407
408 /* not found */
409 if(p==-1){fprintf(stderr,"No such line number in parameter file %s!\n",Pfr->ParameterFile);putch(7);exit(1);}
410
411 /* found */
412 *StorageIndex=p;
413 *Type=ParType;
414
415 }
```

Arquivo de parâmetros param0.txt:

```
chan.txt    /* 0  Chan_InpFile */
1.32        /*1 CMADispConst */
16          /*2 EqlzrLength  */
1e-3        /*3 LrnRate */
eqimpr.txt  /* 4 FinalEqlzrImpResp_OutFile */
decimpr.txt /* 5 ChannelDecimImpResp_InpFile */
100         /*6  NumISIPoints */
isi.txt     /* 7 ISI_OutFile(Index-ISIe-ISIo) */
16          /*8 NumQAMlevels */
iserr.txt   /* 9 ISqErr_OutFile */
1           /*10 OrigQAMGenRandSeed */
8           /*11 InitSpikeIndex */
rgrdes.txt  /* 12 Regressor-Desired_OutFile */
sosf.txt    /* 13 some other stuff file */
1 2 -3 4    /*14 a int vector */
3.1 6.8     /*15 a float vector */
3.14156     /*16 a single value */
baba.txt    /* 17 a file */
```

Execução na tela do console:

```
C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\Exemplos>prgm param0.txt
Line0: chan.txt
Line1:     1.32
Line2:       16
Line3:    0.001
Line4: eqimpr.txt
Line5: decimpr.txt
Line6:      100
Line7: isi.txt
Line8:       16
Line9: iserr.txt
Line10:        1
Line11:        8
Line12: rgrdes.txt
Line13: sosf.txt
Line14:        1              2             -3              4
Line15:      3.1            6.8
Line16:  3.14156
Line17: baba.txt
```