Centro Universitário FACENS

PROGRAMAÇÃO ESTRUTURADA

Engenharia da Computação
Engenharia Elétrica
Engenharia Mecatrônica

Profas Andréa Braga / Talita Berbel / Lívia Gregorin

Sorocaba
2024

Facens

FUNÇÕES 5

1 - FUNÇÕES DA BIBLIOTECA 5
2 - FUNÇÃO DEFINIDA PELO USUÁRIO 5
2.1 – PROTÓTIPO DA FUNÇÃO 5
2.2 – DEFINIÇÃO DA FUNÇÃO 6
2.3 – CHAMANDO FUNÇÕES 6
2.4 – PASSANDO DADOS PARA UMA FUNÇÃO 7
2.5 – TIPOS DE FUNÇÕES CRIADAS PELO PROGRAMADOR 7
2.5.1 – FUNÇÕES QUE NÃO RETORNAM VALOR 7
2.5.2 – FUNÇÕES QUE RETORNAM VALOR 8
3 – FUNÇÕES RECURSIVAS 8
4 - ESTRUTURA DE UM PROGRAMA EM C 9
5 – EXERCÍCIO RESOLVIDO 10
6 - EXERCÍCIOS PROPOSTOS 10

1. ENDEREÇOS DE VARIÁVEIS 11

1.1 – VARIÁVEIS SIMPLES 11
1.2 – MATRIZES 11
1.3 – MATRIZES COMO ARGUMENTOS DE FUNÇÕES 12

2. STRINGS 13

2.1 – VARIÁVEIS CHAR 13
2.2 – DEFINIÇÃO DE STRING 13
2.3 – STRINGS CONSTANTES OU LITERAIS 13
2.4 – MATRIZES DE CARACTERES 13
2.5 – INICIALIZAÇÃO DE MATRIZES DE CARACTERES 14
2.6 – LENDO STRINGS 14
2.6.1 – SCANF 14
2.6.2 – GETS 15
2.6.3 – FGETS 15
2.7 – IMPRIMINDO STRINGS 16
2.7.1 – PRINTF 16
2.7.2 – PUTS 16
2.8 – ALGUNS CUIDADOS 17
2.9 – OUTRAS FUNÇÕES 18
2.10 – EXERCÍCIO RESOLVIDO 19
2.11 – EXERCÍCIOS PROPOSTOS 20

3. PONTEIROS 21

3.1 - DEFINIÇÃO 21
3.2 - PONTEIROS CONSTANTES 21
3.3 - PONTEIROS VARIÁVEIS 21
3.4 – CRIANDO UM PONTEIRO 22
3.5 - DECLARANDO PONTEIROS 22
3.6 – INICIALIZANDO PONTEIROS 23

3.7 – USANDO PONTEIROS 23
3.8 - PONTEIROS E TIPOS DE VARIÁVEIS 24
3.9 – PONTEIROS E MATRIZES 25
3.10 – OPERAÇÕES ARITMÉTICAS COM PONTEIROS 25
3.10.1 – INCREMENTANDO PONTEIROS 25
3.10.2 – DECREMENTANDO PONTEIROS 26
3.10.3 – DIFERENCIAÇÃO COM PONTEIROS 27
3.11 – RESUMO 27
3.12 – EXEMPLOS 28
3.12.1 – VERSÃO UTILIZANDO MATRIZES 28
3.12.2 – VERSÃO UTILIZANDO PONTEIROS 28
3.13 – EXERCÍCIOS PROPOSTOS 29

4. CHAMADA POR REFERÊNCIA 31

4.1 - DEFINIÇÃO 31

5. STRINGS E PONTEIROS 32

5.1 - INICIALIZAÇÃO DE STRINGS ATRAVÉS DE PONTEIROS 32
5.2 - INICIALIZAÇÃO DE UMA MATRIZ DE PONTEIROS PARA STRINGS 33
5.2.1 - VERSÃO MATRIZ DE STRINGS 34
5.2.2 - VERSÃO PONTEIROS 34

6. ALOCAÇÃO DINÂMICA 35

6.1 - INTRODUÇÃO 35
6.2 - ALOCANDO ESPAÇO PARA STRINGS DURANTE A COMPILAÇÃO 35
6.3 - ALOCANDO ESPAÇO PARA STRINGS DURANTE A EXECUÇÃO DO PROGRAMA 36
6.4 - HEAP 36
6.5 - FUNÇÃO MALLOC( ) 36
6.6 - FUNÇÃO CALLOC( ) 39
6.7 - FUNÇÃO REALLOC( ) 41
6.8 - FUNÇÃO FREE( ) 42
6.9 – ALOCAÇÃO DE MEMÓRIA NA MAIN X NA FUNÇÃO 44
6.10 – EXERCÍCIOS PROPOSTOS 48

7. ESTRUTURAS 50

7.1 – INTRODUÇÃO 50
7.2 – DEFININDO UM TIPO ESTRUTURA 50
7.3 - DECLARANDO AS VARIÁVEIS DO TIPO ESTRUTURA 51
7.4 - DEFININDO E DECLARANDO ESTRUTURAS 52
7.5 - ACESSANDO MEMBROS DA ESTRUTURA 53
7.6 - MÚLTIPLAS ESTRUTURAS DE MESMO TIPO 53
7.7 - DEFINIÇÃO DE ESTRUTURAS SEM RÓTULO OU ETIQUETA 54
7.8 - ESTRUTURAS QUE CONTÉM MATRIZES 54
7.9 – EXEMPLO - CRIANDO UMA LISTA DE LIVROS 55
7.10 - INICIALIZANDO ESTRUTURAS 56

7.11 - ATRIBUIÇÕES ENTRE ESTRUTURAS 57
7.12 - ESTRUTURAS ANINHADAS - ESTRUTURAS QUE CONTÉM ESTRUTURAS 57
7.13 - MATRIZES DE ESTRUTURAS 58
7.14 - INICIALIZANDO ESTRUTURAS COMPLEXAS 60
7.15 – EXERCÍCIOS PROPOSTOS 61

8. ESTRUTURAS COMPLEXAS 62

8.1 - ESTRUTURAS E PONTEIROS 62
8.2 - PONTEIROS PARA ESTRUTURAS 62
8.2.1 - OPERADOR DE ACESSO INDIRETO 63
8.2.2 - OPERADOR DE INDIREÇÃO (*) 63
8.2.3 - NOME DA ESTRUTURA 63
8.3 - PONTEIROS E MATRIZES DE ESTRUTURAS 64
8.4 - PASSANDO ESTRUTURAS COMO ARGUMENTOS PARA FUNÇÕES 66
8.5 – ESTRUTURAS E ALOCAÇÃO DINÂMICA 68
8.8 – EXERCÍCIOS PROPOSTOS 70

9. ARQUIVOS EM I/O 73

9.1 - INTRODUÇÃO 73
9.2 – ALTO-NÍVEL 73
9.3 – BAIXO-NÍVEL 73
9.4 – TEXTO X BINÁRIO 73
9.5 – OPERAÇÕES COM ARQUIVOS EM DISCO – ALTO-NÍVEL 74
9.5.1 – PONTEIRO PARA ARQUIVO 74
9.5.2 – FUNÇÃO FOPEN( ) 74
9.5.3 – FUNÇÃO FCLOSE( ) 75
9.5.4 – LEITURA E GRAVAÇÃO – REGISTROS 75
9.5.5 – EXEMPLOS 76
9.6 – ACESSO ALEATÓRIO – ALTO-NÍVEL 77
9.6.1 – PONTEIRO PARA ARQUIVO 77
9.6.2 – FUNÇÃO FSEEK( ) 78
9.6.3 – FUNÇÃO FTELL( ) 78
9.6.4 – FUNÇÃO REWIND( ) 78
9.6.5 – EXEMPLOS DE ACESSO ALEATÓRIO 79
9.6.6 – EXERCÍCIOS 80

FUNÇÕES

Em C temos:

- funções da biblioteca
- funções definidas pelo usuário

1 - Funções da Biblioteca

Exemplos de funções da biblioteca: printf( ), scanf( ) – para utilizar estas funções, deve-se incluir o arquivo de cabeçalho <**stdio.h**>.

2 - Função definida pelo usuário

É uma seção de código independente e autônoma, escrita para desempenhar uma tarefa específica. Deve conter protótipo e definição da função.

Pode receber informações através de argumentos, cada função tem um nome **único** e suas instruções são executadas sempre que esse nome é mencionado no programa, isto é conhecido como "**chamar uma função**".

2.1 – Protótipo da função

Fornece ao compilador a descrição de uma função que será definida posteriormente no programa.

sintaxe:
tipo_retorno nome_função(tipo_param $nome_1$, ..., tipo_param $nome_n$);

tipo_retorno – tipo de dado que a função retornará. Pode ser char, int, float, double e void (se não retornar nada).
nome_função – descreve o que a função faz.
tipo_param – tipo e nome das variáveis que serão passados para a função. Pode ser char, int, float, double, void.
$nome_n$

- **Sempre termina com ; (ponto e vírgula), geralmente vem nas componentes iniciais ou antes da função ser utilizada.**

Ex.:

```
int produto( int x, int y);
float metade(int k);
void moldura( );
int seleciona(void);
void func1(int x, int y, int z);
```

2.2 – Definição da função

É a função propriamente dita. Contém as instruções que serão executadas.

É composta de cabeçalho + o corpo da função. A 1ª linha chamada **cabeçalho da função** é idêntica ao protótipo com exceção do **;** (ponto e vírgula). O **corpo da função** necessariamente deve estar entre **{ }**(chaves), contém instruções e/ou **variáveis locais** (variáveis que só são conhecidas desta função).

Aparecem no mesmo arquivo de código-fonte e após o final de main( ) ou em arquivo independente, porém será visto mais adiante, pois utiliza diretivas de pré-processamento e/ou outros recursos.

sintaxe:

```
tipo_retorno    nome_função(tipo_param  nome1, ...., tipo_param nomen)    // cabeçalho
{
instruções;                                                               // corpo da função
}
```

tipo_retorno – tipo de dado que a função retornará. Pode ser char, int, float, double e void (se não retornar nada).
nome_função – descreve o que a função faz.
tipo_param – tipo e nome das variáveis que serão passados para a função. Pode ser char, int, float, double, void.
nomen

- **Não termina com ; (ponto e vírgula), o corpo deve estar entre { } e vem após o final da função main , ou seja, após }// main (chave final).**

Ex.:

```
void moldura( )                    // cabeçalho
{                                  // abertura de chaves
printf("********\n");
printf("*       *\n");             // corpo da função
printf("********");
}                                  // fechamento de chaves
```

2.3 – Chamando funções

É a referência a função dentro do programa. "**Chamamos**", ou seja, executamos uma função, simplesmente usando seu nome, seguido de parênteses, podendo conter ou não a lista de argumentos.
Ex.:

```
moldura( );          /* nome seguido de parênteses */
produto(9, 654);     /* nome, seguido de parênteses contendo lista de argumentos */
metade(x);
func1(a, b, c);
```

2.4 – Passando dados para uma função

O mecanismo usado para transmitir informações para uma função é chamado **argumento**. Os argumentos podem ser constantes, variáveis, expressões matemáticas, expressões lógicas e até outra função. Os argumentos devem ser compatíveis com os parâmetros em quantidade e tipo de dado.
Ex.:

```
b = quadrado (metade (y));          // a = metade (y);
                                    // b = quadrado (a)
```

chamada: func1(a, b, c);

cabeçalho: void func1(int x, int y, int z)

a, b, c (argumentos) – devem ser compatíveis com os **parâmetros x, y, z** em quantidade e tipo.

Nota: **Parâmetros e Argumentos são diferentes:**

Parâmetros: são "guardadores de lugar" para um argumento, são fixos, contém o tipo e o nome dos dados.
Argumentos: são valores reais.

Ex.:
chamadas:

```
metade(x);              /* k = x */
metade(y);              /* k = y */
float metade (float k)          /* cabeçalho da função metade */
```

2.5 – Tipos de funções criadas pelo programador

Há 2 tipos de funções:

- funções que **não** retornam nada
- funções que retornam valores

2.5.1 – Funções que NÃO retornam valor

São aquelas declaradas como **void** e que no corpo da sua função contém apenas instruções e/ou comandos.

sintaxe:
protótipo: void moldura( );
referência a função dentro do programa: moldura( );
definição:

```
void moldura( )
{
printf("*********\n");
printf("*       *\n");
printf("*********");
}
```

Saída na tela :

```
********
*      *
********
```

2.5.2 – Funções que RETORNAM valor

São aquelas declaradas como **char, int, float, ou double**, o **valor de retorno** deve ser armazenado em uma **variável** ou passado como **argumento** e no corpo da sua função além das instruções e/ou comandos, contém **necessariamente** o comando **return( )**.

protótipo: float divisão(int x, int y);

referência a função dentro do programa: d=divisão(a, b); ou printf("%f", divisão(a, b));

definição:

```
float divisão(int x, int y)
{
return(x / y);
}
```

ou

```
float divisão(int x, int y)
{
float resp;
resp = x/y;
return resp;
}
```

- **Diferença: precisa do return( ) para devolver o resultado.**

Nota: Comando return

- **return(expressão) ou return variável** – comando que devolve um **único** valor e retorna, imediatamente, para a próxima instrução do código de chamada**.**
- **return** – sem qualquer valor ou variável, causa uma saída imediata da função na qual ela se encontra, porém **sem** retornar qualquer valor.

3 – Funções Recursivas

Funções que chamam a si própria direta ou indiretamente:

- **Recursão indireta** – uma função chama a outra que, por sua vez chama a primeira.
- **Recursão direta** – uma função chama a si mesma. ex.: fatorial

Fatorial: n! = n * (n-1)!

3! = 3*2*1 ou 3! = 3*2!
2!= 2*1!
1! = 1*0!
0! = 1

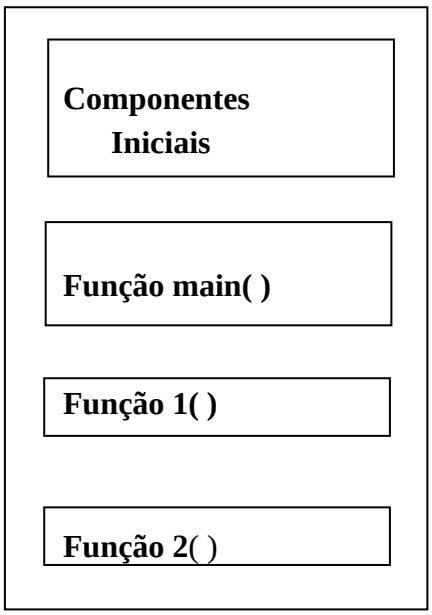
Ex.:
protótipo: long int fatorial (long int num);

referência a função dentro do programa: fat=fatorial(n);ou printf("%i", fatorial(n));

definição:

```
long int fatorial (long int num)
{
if (num == 0)
        return 1;                       //0! = 1
return (n*fatorial(n-1));               //3! = 3*2!
}
```

4 - Estrutura de um Programa em C



contém: arquivos de cabeçalho, declaração de constantes, protótipos e variáveis globais

contém: variáveis locais e instruções do programa

contém: variáveis locais e definição da função 1

contém: variáveis locais e definição da função 2

5 – Exercício Resolvido

Resolver a expressão z = x + y onde x e y são definidos pelo usuário.

```
#include <stdio.h>                                          // Componentes Iniciais
float soma (float a, float b);        // protótipo da função soma
```

```
main( )                                                     // Função main
{
float  x,y,z;
x=y=z=0;
printf("\n Este programa calcula a expressão Z=X+Y ");
printf("\n\n Digite o valor para x=  ");
scanf("%f", &x);
printf("\n\n Digite o valor para y=  ");
scanf("%f", &y);
z=soma(x, y);                                  // chamada a função soma
printf("\n\n\n A expressão z=x+y para x=%.2f e y=%.2f é %.2f ", x, y, z);
} /* main */
```

```
float soma (float a, float b)        // definição da função soma        // Função soma
{
return(a+b);                         // corpo da função som
} // soma
```

6 - Exercícios Propostos

Todos os exercícios devem ser resolvidos, utilizando-se funções.

1. Calcular e mostrar o produto de 2 números.

2. Ler 2 notas e calcular a média.

3. Pedir a idade para o usuário e calcular quantos meses e dias de vida ele tem aproximadamente.

4. Calcular o consumo médio de gasolina de um tanque de automóvel. Pedir para o usuário entrar com a distância (km) e volume (litros). Cm = d (km) / v (litros)

5. Calcular a área de uma esfera. $A = 4 * \pi * r^2$

6. Receber um nº inteiro longo do usuário e calcular o fatorial. Utilizar função recursiva direta.

1. ENDEREÇOS DE VARIÁVEIS

1.1 – Variáveis Simples

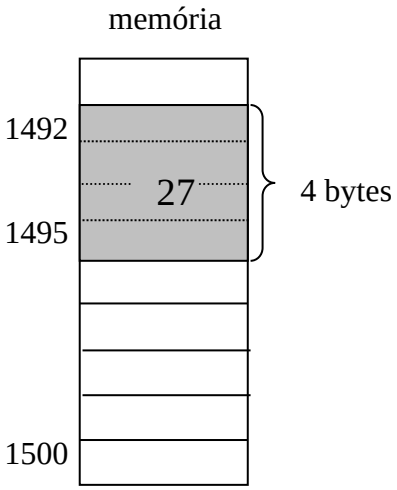
Ex.: Variável inteira

```
int          num = 27;
```

nome : num
valor : 27
endereço : 1492
nome do endereço : &num

portanto:

num = 27
&num = 1492



1.2 – Matrizes

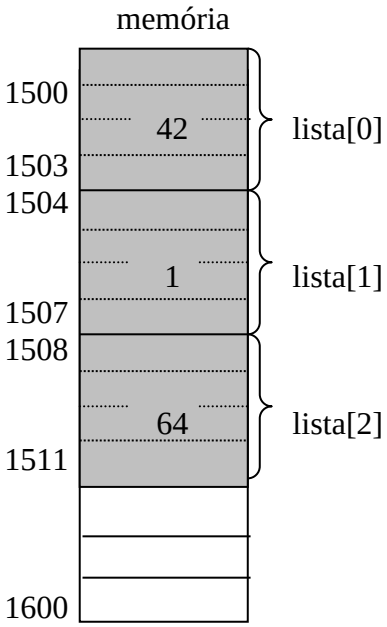
Ex.: Matriz inteira

```
int    lista[10] = {42, 1, 64, 36, ......};
```

matriz : lista[10]
endereço matriz : 1500
nome do endereço : lista

elemento 0 : lista[0]
valor : 42
endereço : 1500
nome do endereço : &lista[0]

elemento 1 : lista[1]
valor : 1
endereço : 1502
nome do endereço : &lista[1]



nós teremos:
&lista[0]
&lista[1]
.
.
.
&lista[9]

portanto:

visto que lista[0] é o 1º elemento da matriz, ele terá o mesmo endereço da própria matriz, ou seja,
lista = = &lista[0]

lista é o nome do endereço da matriz e não **&lista**

~~**&lista**~~

- **O nome de uma matriz desacompanhado de colchetes é equivalente ao** endereço **da matriz.**

1.3 – Matrizes como Argumentos de Funções

C permite passar uma matriz para uma função usando **unicamente o nome da matriz**. Mas, o que representa o nome da matriz sem colchetes?

O nome de uma matriz desacompanhado de colchetes é equivalente ao **endereço** da matriz.

Passando o nome de uma matriz para uma função, **NÃO** é criada uma nova cópia da matriz. Essa maneira é conhecida como **Chamada por Referência**, onde **NÃO** são passados os valores contidos na matriz e sim o **endereço** que acessa a matriz real.

OBS.: ver maiores detalhes no capítulo “Chamada por Referência”.

2. STRINGS

2.1 – Variáveis char

As variáveis alfanuméricas são especificadas por apenas 1 (um) único caracter delimitado por aspas simples.

Ex.:
```
char    letra = 'a';      // letra contém o caracter a que representa o cógigo 97 em ASCII
```

portanto:
```
printf("%c", letra);      // saída na tela: a
printf("%i", letra);      // saída na tela: 97
```

- **Como as variáveis do tipo char só podem conter um único caracter, se quisermos armazenar 2 ou mais caracteres teremos que fazer uso de uma string.**

2.2 – Definição de String

É uma seqüência de caracteres ou uma "matriz" de caracteres.

2.3 – Strings Constantes ou Literais

É uma seqüência de caracteres delimitada por aspas duplas. Sempre que o compilador encontrar qualquer coisa entre aspas duplas, ele reconhece como uma string constante e automaticamente acrescenta o caracter null (\0) no final da string.

Ex.:

"a" → 'a' + '\0' → a\0

"Alabama" ⟶ Alabama\0

O comprimento de um string é determinado através da localização do caracter null.

- **String é uma seqüência de caracteres terminando com o caracter null.**

2.4 – Matrizes de Caracteres

Para armazenar um string, devemos declarar uma matriz do tipo **char** com a quantidade de elementos que queremos mais um, ou seja, 1 para cada caracter e o último para o caracter null (\0).

Ex.: se quisermos armazenar um string com 7 caracteres, devemos utilizar uma matriz do tipo char com 8 elementos.

```
Alabama\0          char    string[8];
a\0                char    letra[2];
```

2.5 – Inicialização de Matrizes de Caracteres

Ex.:

```
char    string[8] = {'A', 'l', 'a', 'b', 'a', 'm', 'a', '\0'};
            ou
char    string[8] = "Alabama";    // string literal – automaticamente caracteres + \0
            ou
char    string[ ] =  "Alabama";    // string literal – automaticamente caracteres + \0
```

2.6 – Lendo Strings

Uma vez declarada a string, podemos recebê-la e para isto temos pelo menos 3 funções: scanf, gets e fgets, todas da biblioteca <**stdio.h**>.

2.6.1 – scanf

A função scanf( ) é limitada para leitura de strings, pois ela usa qualquer espaço em branco (espaço, caracter de tabulação ou nova linha) para terminar a entrada.

Por exemplo: se você quiser digitar seu nome e sobrenome numa mesma string não conseguirá, pois ao encontrar um espaço em branco, a função encerra e só armazena o nome, desconsiderando o que vier em seguida.

Ex.:

```
char        nome[80];
   printf("Digite seu nome e sobrenome");
   scanf("%s", nome);
   printf("\n Bom dia "%s", nome);
```

aparecerá na tela:

```
Digite seu nome e sobrenome        Hamilton Gomes
Bom dia        Hamilton
```

- **Note que não usamos o operador & precedendo o 2º argumento em scanf, pois vimos que o nome de uma matriz é o seu endereço inicial, portanto seria errado utilizar o operador &.**

~~scanf("%s", &nome);~~

A função scanf( ) é usada para ler uma mistura de tipos de dados numa mesma instrução.
Ex.:

```
char    mat[20];
int num;
float   preço;
        printf("Digite o material, seu nº de estoque e seu preço");
        scanf("%s    %i    %f", mat, &num, &preço);

char    nome[15], snome[15];
int     rg;
        printf("Digite seu nome, sobrenome e RG");
        scanf("%s    %s    %i", nome, snome, &rg);
        printf("\n Bom dia "%s", nome);
```

2.6.2 – gets

gets - lê **strings** do teclado.

sintaxe:

gets(nome_string);

A função gets é conveniente para a leitura de strings, pois ela lê caracteres até encontrar o caracter de nova linha('\n'), que é gerado quando a tecla ENTER é pressionada pelo usuário. Portanto caracteres brancos (espaços e tabulações) são aceitos como parte da string.
Ex.:

```
char    nome[20];
        printf("Digite seu nome e sobrenome");
        gets(nome);
        printf("\n Bom dia "%s", nome);
```

aparecerá na tela:

Digite seu nome e sobrenome **Hamilton Gomes**
Bom dia **Hamilton Gomes**

2.6.3 – fgets

fgets - lê **strings** do teclado (entrada padrão) ou de um arquivo, o que for definido.

sintaxe:

fgets(nome_string, num, entrada);

onde:
num – inteiro que indica o limite de caracteres a serem lidos, não esquecer do '\0' (NULL).
entrada – corresponde a entrada padrão do teclado (**stdin**) ou de um arquivo.

A função fgets é conveniente para a leitura de strings, pois ela lê caracteres até encontrar o caracter de nova linha('\n'), que é gerado quando a tecla ENTER é pressionada pelo usuário ou até que o limite especificado (**num-1**) seja atingido, pois a função fgets lê **num-1** caracteres e coloca automaticamente o null (\0)**.** Portanto caracteres brancos (espaços e tabulações) são aceitos como parte da string.

Ex.:
```
char        nome[20];
   printf("Digite seu nome e sobrenome");
   fgets(nome, 11, stdin);
   printf("\n Bom dia "%s", nome);
```

aparecerá na tela:

```
   Digite seu nome e sobrenome Hamilton Gomes
   Bom dia          Hamilton G                    // foi especificado apenas 10 caracteres + \0
```

2.7 – Imprimindo Strings

As 3 principais funções são: printf, puts e fputs.

2.7.1 – printf

Permite exibir mais de um string, ou ainda misturá-los com texto e/ou variáveis numéricas.

Ex.:
```
char        banco[30] = "Banco do Brasil";
char        nome[80] = "Hamilton Gomes";
float       saldo = 978.36;
printf("O saldo de  %s  no  %s  é de  %.2f  reais", nome, banco, saldo);
```

aparecerá na tela:

```
   O saldo de  Hamilton Gomes  no  Banco do Brasil  é de  978.36  reais
```

2.7.2 – puts

puts - imprime uma única **string** por vez na tela e o arquivo de cabeçalho: **stdio.h**

sintaxe:

puts(nome_string);

A função puts( ) é o complemento de gets( ) e ela pula linha automaticamente após cada string exibida.

Ex.:
```
char        banco[30] = “Banco do Brasil”;
char        nome[80] = “Hamilton Gomes”;
    puts(nome);
    puts(banco);
```

aparecerá na tela:

```
    Hamilton Gomes
    Banco do Brasil
```

OBS.:
```
char        nome[80] = “Hamilton Gomes”;
    puts(nome);             // escreve a partir da 1ª posição
    puts(&nome[4]);         // escreve a partir da 4ª posição
```

aparecerá na tela:

```
    Hamilton Gomes
    ilton Gomes
```

Escrita no arquivo – grava informações no arquivo
sintaxe:

```
        fputs (str, fptr);          // armazena um caracter por vez no arquivo
```

onde:
str – string a ser armazenado
fptr – corresponde ao ponteiro do arquivo a ser gravado

2.8 – Alguns Cuidados

Os strings têm sua manipulação levemente dificultada. Não é possível, por exemplo, copiar o conteúdo de um string em outro através da seguinte instrução :

~~string1 = string2;~~

Devemos proceder como fazíamos com matriz, ou seja, teremos um for varrendo todos os índices.
Ex.:

```
        for (i=0; i<10; i++)
            string1[i] = string2[i];
```

Posições específicas de um string podem receber valores diretamente.
Ex.:

```
                string1[0] = ‘a’;   // a é um char (deve ser colocado entre aspas simples)
```

2.9 – Outras Funções

Existem conjuntos de funções específicas para fazer a manipulação de strings, que estão inseridas na biblioteca <string.h>. A seguir serão apresentadas algumas dessas funções:

strcpy(copia, original);
copia o conteúdo de **original** para **copia**.

strcat(copia, original);
coloca o conteúdo de **original** no final de **copia**.

strcmp(str1, str2);
compara **str1** com **str2**, retornando 0 (zero) se forem iguais, um valor positivo se **str1** for maior (comparação alfabética) ou negativo se **str2** for maior.

strlen(string);
retorna o tamanho (em caracteres) de **string**.

memcpy(copia, original, tamanho);
copia o conteúdo de **original** para **copia** até o **tamanho** especificado.

memcmp(str1, str2, tamanho);
compara **str1** com **str2** até o número de caracteres especificado em **tamanho**, retornando 0 (zero) se forem iguais, um valor positivo se **str1** for maior (comparação alfabética) ou negativo se **str2** for maior.

2.10 – Exercício Resolvido

Verifique a validade de uma senha fornecida pelo usuário. A senha será composta por 5 caracteres.

versão 1

```
#include <stdio.h>
#include <stdlib.h>
main()      {
char senha1[6], senha2[6];  // vetor de 6 elementos do tipo char –contém um string com 5 caracteres
int i, erro =0;
printf(“Cadastramento de Senhas”);
printf(“Digite sua senha”);
scanf(“%s”, senha1);
fflush(stdin);
printf(“Confirme sua senha”);
scanf(“%s”, senha2);
fflush(stdin);
for (i=0; i<5:i++)
    if(senha1[i] ! = senha2[i])
            erro =1;
if (erro = = 0)
    printf(“Senha Válida”);
else
    printf(“Senha Inválida”);
system(“pause”);
} // main
```

versão 2

```
#include <stdio.h>
#include <stdlib.h>
main()      {
char senha1[6], senha2[6];  // vetor de 6 elementos do tipo char –contém um string com 5 caracteres
printf(“Cadastramento de Senhas”);
printf(“Digite sua senha”);
gets(senha1);
fflush(stdin);
printf(“Confirme sua senha”);
gets(senha2);
fflush(stdin);
if (strcmp(senha1, senha2) = = 0)
    printf(“Senha Válida”);
else
    printf(“Senha Inválida”);
system(“pause”);
} // main
```

2.11 – Exercícios Propostos

1. Faça um programa que, dado um string qualquer, inverta todas as suas letras e apresente isso ao usuário.
   Exemplo:
   String: Alunos da Facens
   Resultado: snecaF ad sonulA

2. Melhore o programa, armazenando a nova seqüência em um string auxiliar.

3. Melhore o programa, trocando as maiúsculas por minúsculas e vice-versa, quando necessário.
   Exemplo - Resultado: Snecaf Ad Sonula

   - Será necessário utilizar as funções toupper(char) e tolower(char), que transformam um caracter para maiúscula e minúscula respectivamente, ou
   - Utilizar os valores da tabela ASCII: **A = 65, Z = 90** **a = 97, z = 122**, portanto para passar de MAIÚSCULA para minúscula basta somar 32 e o inverso, subtrair 32.

4. Faça um programa que, dado um string qualquer, inverta e armazene todas as suas letras sem auxílio de uma string auxiliar, ou seja, utilizando a mesma string. Apresente isso ao usuário.

5. Faça um programa que, dado um string qualquer, procure dentro dele a ocorrência de um determinado caracter. **Exemplo:** String: Facens – Computação Procurar pela letra: x
   Resultado: O caracter x não foi encontrado.

6. Receba uma frase e uma letra do usuário, conte quantas ocorrências da letra existem na frase.

7. Melhore o seu programa e mostre as posições em que as letras aparecem.

8. Faça um programa que, dado um string qualquer, procure dentro dele a ocorrência de outro string.
   Exemplo: String: Facens – Computação Procurar pelo string: put
   Resultado: O string put foi encontrado.

9. **Nome JEDI** – para compor um nome conforme o costume JEDI, do filme Guerra nas Estrelas, é necessário fazer a seguinte composição:

- Nome Jedi: as 3 primeiras letras do primeiro nome + as 2 primeiras letras do sobrenome
- Sobrenome Jedi: as 2 primeiras letras do nome da mãe + as 3 primeiras letras da cidade em que nasceu
- Nome composto: as 3 primeiras letras do nome Jedi + as 3 últimas letras do sobrenome Jedi.

Exemplo:
Nome: **Jos**é da **Si**lva
Nome da mãe: **Ma**ria da Silva
Cidade em que nasceu: **Sor**ocaba

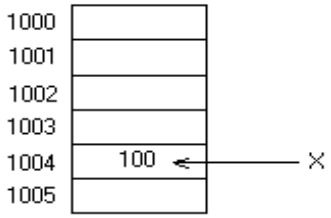
Nome Jedi: Jossi Masor
Nome composto: Jos-Sor

3. PONTEIROS

3.1 - Definição

Proporcionam um modo de acesso as variáveis sem referenciá-las diretamente, utilizando para isto o endereço da variável.

Em C, quando uma variável é declarada, o compilador reserva uma localização de memória com um endereço **único** para armazená-la. O compilador associa esse endereço ao nome da variável e quando no programa a variável é referenciada, automaticamente acessamos o seu endereço (ou localização correspondente na memória).



3.2 - Ponteiros Constantes

São aqueles que não podem ser alterados e permanecem imutáveis durante a execução do programa.

Ex.:

&x – contém o endereço da variável x

O nome de uma matriz sem os colchetes. Pois contém o endereço do 1º elemento dessa matriz

int lista[10];→ **lista** = =&lista[0]

Portanto lista é um ponteiro constante para a matriz lista[10] e realmente não pode ser alterado, pois se mudássemos o seu valor, passaria a apontar para outro lugar e não para a matriz.

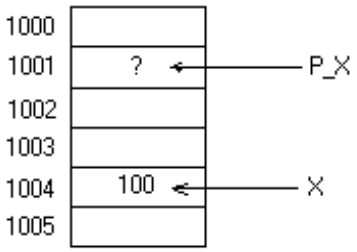
→ Ponteiro Constante é um endereço

3.3 - Ponteiros Variáveis

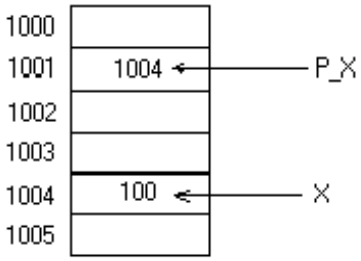
É um lugar para guardar endereços.

3.4 – Criando Um Ponteiro

Iremos criar uma variável para armazenar o endereço de x, iremos chamá-la de p_x. Inicialmente, p_x ainda não foi inicializada, apenas foi reservado um espaço para p_x.



O próximo passo é armazenar o endereço de x na variável p_x. Portanto dizemos que p_x agora **aponta** para x, ou é um ponteiro para x.



→ Ponteiro é uma variável que contém o endereço de outra variável.

3.5 - Declarando Ponteiros

Um ponteiro é uma variável numérica e, como todas as variáveis, deve ser declarado antes de ser usado.

sintaxe:

tipo *nome_ponteiro;

onde:

tipo – tipo da variável para a qual o ponteiro estará apontando. Pode ser um dos tipos de variáveis em C : char, int, float,...

* – operador de **indireção** e indica que nome_ponteiro é um ponteiro para uma variável daquele tipo e não uma variável desse tipo.

nome_ponteiro – segue as mesmas regras para os nomes de variáveis.

→ é declarado junto com as variáveis.

Ex.:
char *c; // indica que *c* é um ponteiro para uma variável do tipo char e não que *c* é char
float *****valor; // indica que *valor* é um ponteiro para uma variável do tipo float e não que *valor* é float

3.6 – Inicializando Ponteiros

Um ponteiro só é útil depois de inicializado, ou seja, que passa a conter um endereço de uma variável.

sintaxe:
nome_ponteiro = &variável;

Ex.:
int x, *p_x;

p_x = &x; // atribui o endereço de x para p_x

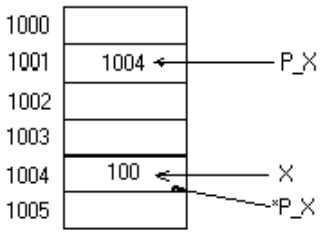
3.7 – Usando Ponteiros

Quando o operador ***** precede o nome de um ponteiro, ele se refere à variável para qual o ponteiro está apontando.

Ou seja, se escrevermos *p_x, estamos nos referindo a variável x, se quiséssemos imprimir o valor de x (que é 100 no exemplo), poderíamos escrever:

printf (“%i”, x); // **acesso direto** – acesso o conteúdo de uma variável usando o seu nome
printf (“%i”, *p_x); // **acesso indireto** - acesso ao conteúdo de 1 variável usando um ponteiro

As 2 expressões são equivalentes



Resumo:

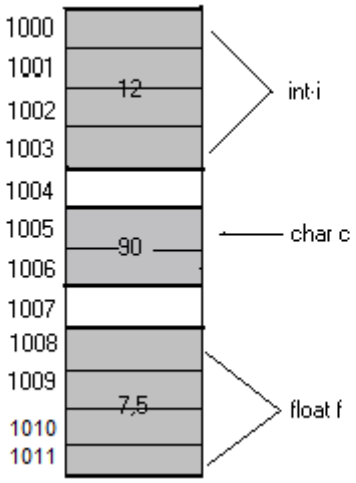
```
#include <stdio.h>
main( )          {
int var = 10;              // declaração e inicialização de uma variável inteira
int * ptr;                 // declaração de um ponteiro para uma variável inteira
ptr=& var;                 // inicialização de um ponteiro
printf (“Mostra o conteúdo de var”);
printf (“\n Acesso direto – var = %i”, var);
printf (“\n Acesso indireto var = %i”, *ptr);
printf (“\n\n\n Mostra o endereço de var”);
printf (“\n Ponteiro constante = %i”, &var);
printf (“\n Ponteiro variável = %i”, ptr);
} // mai
```

Portanto: → **tanto *ptr como var referem-se ao conteúdo de var**
→ **tanto ptr como &var referem-se ao endereço de var**

3.8 - Ponteiros e Tipos de Variáveis

Os tipos de variáveis ocupam diferentes quantidades de memória. Geralmente 1 variável do tipo char ocupa 1 byte(DOS e WINDOWS) e 2 bytes(LINUX), uma variável inteira ocupa 2 bytes(DOS) e 4 bytes(LINUX e WINDOWS), uma variável float 4 bytes(DOS, LINUX e WINDOWS). E como cada byte tem 1 endereço, teremos mais de 1 endereço para uma variável ou seja:
Ex.:

```
int     i  =  12;
char    c  =  90;
float   f  =  7,5;
```



Como, então, um ponteiro pode apontar para uma variável que ocupe múltiplos bytes?

→ O endereço de uma variável é apenas o endereço do seu byte de nº mais baixo, ou seja, para o primeiro de 2 bytes ou o primeiro de 4 bytes, etc.

Ao declararmos ponteiros para o exemplo anterior, teremos:

```
int     *p_i;
char    *p_c;
float   *p_f;
```

Portanto:

```
p_i  =  &i;          //  p_i  =  4 bytes começando em 1000
p_c  =  &c;          //  p_c  =  2 bytes começando em 1005
p_f  =  &f;          //  p_f  =  4 bytes começando em 1008
```

3.9 – Ponteiros e Matrizes

Em C, o compilador transforma matrizes em ponteiros, pois a arquitetura do micro computador entende ponteiros e não matrizes. Qualquer operação que pode ser feita com índices de uma matriz pode ser feita com ponteiros.

O nome de uma matriz sem os colchetes é um ponteiro para o 1º elemento dessa matriz. Porém, é um ponteiro constante, pois não pode ser alterado. Mas, podemos também declarar um ponteiro variável e inicializá-lo para apontar para uma matriz e assim realizarmos qualquer operação com esse ponteiro.

Ex.:

```
int dados[10], *p_dados;

dados  = =  &dados[0];          // dados     =  ponteiro constante
p_dados  =  dados;              // p_dados   =  ponteiro variável
```

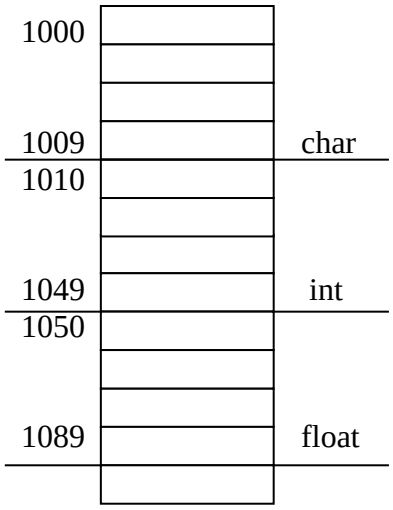
3.10 – Operações Aritméticas com Ponteiros

3.10.1 – Incrementando Ponteiros

Quando um ponteiro é incrementado, na verdade estamos incrementando o seu valor. Ou seja, em C se o ponteiro é incrementado em uma unidade, o endereço armazenado desse ponteiro será incrementado em um valor correspondente ao **tamanho** desse tipo de dados. E como C sabe qual é o tipo de dados? Ele se baseia na declaração do ponteiro. Na verdade, ao incrementarmos o ponteiro, podemos afirmar que ele apontará para o próximo elemento da matriz.

Ex.:

```
char    c[10],  *p_c;
int      i[10],  *p_i;
float    f[10],  *p_f;
```



Se incrementarmos os ponteiros em uma unidade todos apontarão para o próximo elemento da matriz.
então:

```
p_c ++;        // será incrementado em 1 byte, que é o tamanho de char
p_i ++ ;       // será incrementado em 4 bytes, que é o tamanho de int
p_f ++;        // será incrementado em 4 bytes, que é o tamanho de float
```

Se somarmos o valor de **n** ao ponteiro, estaremos incrementando o ponteiro com um valor correspondente a **n** elementos da matriz do tipo ao qual ele está associado.
então:

```
p_c   =  p_c + 4;        // passa a apontar para 4 elementos, i.e., 4 bytes mais adiante
p_i + =  5 ;             // passa a apontar para 5 elementos, i.e., 20 bytes mais adiante
p_f + =  8;              // passa a apontar para 8 elementos, i.e., 32 bytes mais adiante
```

3.10.2 – Decrementando Ponteiros

Semelhante ao incremento.

3.10.3 – Diferenciação com Ponteiros

Consiste na subtração de 2 ponteiros. Se tivermos 2 ponteiros para elementos diferentes de uma mesma matriz, podemos subtraí-los para descobrir qual a distância entre esses elementos.
então:

ptr1 – ptr2 // distância entre esses elementos – o resultado pode ser positivo ou negativo

→ As comparações (= =, !=, <, <=, >, >=) entre ponteiros só são válidas quando ambos estão apontando para a mesma matriz. Pois se, compararmos ponteiros que apontam para variáveis de tipos diferentes, teremos resultados sem sentido.

3.11 – Resumo

Há 6 operações que podem ser efetuadas com ponteiros:

atribuição: p_num = &num // variável simples
p_num = num // matriz

indireção: *p_num // conteúdo da variável apontada por p_num

endereço-de: &p_num // endereço do ponteiro
p_num // endereço da variável apontada

incremento: p_num++ ou p_num = p_num +n ou p_num + = n;

decremento: p_num- - ou p_num = p_num – n ou p_num - = n;

comparação: válida somente quando ambos ponteiros apontam para a mesma matriz.

OBS.:

int mat[10];

mat é o ponteiro constante , portanto não é permitido

~~mat++~~

“notação matriz” mat[2] = = *(mat+2) “notação ponteiro constante”
“notação matriz” &mat[2] = = mat+2 “notação ponteiro constante”

portanto

***(matriz + índice) = = matriz[índice]**

3.12 – Exemplos

3.12.1 – Versão Utilizando Matrizes

```
#include <stdio.h>

main ( )
{
int num[5], d;

printf (“ Lendo os elementos da matriz ”);
for (d = 0; d<5; d++)
                    {
          printf (“ \n  %i  elemento:”, d+1);
          scanf (“%i”, &num[d]);
          }
printf (“ Imprime os elementos da matriz ”);
for (d = 0; d<5; d++)
          printf (“ \n  num[%i] = %i”, d, num[d]);
}//main
```

3.12.2 – Versão Utilizando Ponteiros

```
#include <stdio.h>

main ( )
{
int num[5],  *p_num,  d;

p_num = num;                                    // inicialização do ponteiro
printf (“ Lendo os elementos da matriz ”);
for (d = 0; d<5; d++)
          {
          printf (“ \n  %i  elemento:”, d+1);
          scanf (“%i”, p_num ++);
          }
p_num = num;                                    // posiciona o ponteiro no endereço inicial
printf (“ Imprime os elementos da matriz ”);
for (d = 0; d<5; d++)
          printf (“\n  num[%i]  =  %i”, d,  * (p_num ++));
}//main
```

3.13 – Exercícios Propostos

1. Idem ao exercício resolvido, porém a quantidade de nºs reais deverá ser estipulada pelo usuário. Lembrar que a matriz ao ser definida deve conter necessariamente o tamanho.

2. Idem ao exercício resolvido, porém perguntar ao usuário se ele deseja continuar (S/N). Lembrar que a matriz ao ser definida deve conter necessariamente o tamanho.

3. Realizar a soma de 2 vetores A e B de nºs reais e de tamanho igual ou inferior a 100 (n<=100) e gerar um vetor resultante C onde

   C[i] = A[i] + B[i]

4. Ler e armazenar 2 notas, verificar se estão no intervalo de 0 à 10. Calcular a média para 100 alunos (utilizar matrizes) e verificar :
   se m >= 7 – imprimir APROVADO
   se 4 <= m < 7 – imprimir EXAME
   se m < 4 – imprimir REPROVADO.

5. Ler 10 elementos de um vetor. Verificar e mostrar o maior elemento.

6. Ler 10 elementos de um vetor. Verificar e mostrar a média dos elementos.

7. Ler 10 elementos de um vetor e colocá-los na ordem crescente.

8. Crie uma tabela (vetor) que armazene as temperaturas diárias registradas durante um mês e em seguida elabore um programa que calcule :
   - A maior temperatura e o dia em que ocorreu
   - A temperatura média mensal
   - As temperaturas inferiores à média e os dias em que ocorreram.

9. Ler um vetor (A) com 15 elementos. Construir um vetor B do mesmo tipo e tamanho, sendo que cada elemento de B seja o fatorial do elemento correspondente de A. (utilizar a função recursiva direta p/ fatorial).

Refazer os 9 exercícios de **Strings**, agora utilizando **Ponteiros**.

1. Faça um programa que, dado um string qualquer, inverta todas as suas letras e apresente isso ao usuário. **Exemplo -** String: Alunos da Facens
Resultado: snecaF ad sonulA

2. Melhore o programa, armazenando a nova seqüência em um string auxiliar.

3. Melhore o programa, trocando as maiúsculas por minúsculas e vice-versa, quando necessário. **Exemplo -** Resultado : Snecaf Ad Sonula

- Será necessário utilizar as funções toupper(char) e tolower(char), que transformam um caracter para maiúscula e minúscula respectivamente, ou
- Utilizar os valores da tabela ASCII: **A = 65, Z = 90** **a = 97, z = 122**, portanto para passar de MAIÚSCULA para minúscula basta somar 32 e o inverso, subtrair 32.

4. Faça um programa que, dado um string qualquer, inverta e armazene todas as suas letras sem auxílio de uma string auxiliar, ou seja, utilizando a mesma string. Apresente isso ao usuário.

5. Faça um programa que, dado um string qualquer, procure dentro dele a ocorrência de um determinado caracter. **Exemplo -** String : Facens – Computação Procurar pela letra : x
Resultado : O caracter x não foi encontrado.

6. Receba uma frase e uma letra do usuário, conte quantas ocorrências da letra existem na frase.

7. Melhore o seu programa e mostre as posições em que as letras aparecem.

8. Faça um programa que, dado um string qualquer, procure dentro dele a ocorrência de um outro string.
Exemplo: String : Facens – Computação Procurar pelo string: put
Resultado : O string put foi encontrado.

9. **Nome JEDI** – para compor um nome conforme o costume JEDI, do filme Guerra nas Estrelas, é necessário fazer a seguinte composição:

- Nome Jedi: as 3 primeiras letras do primeiro nome + as 2 primeiras letras do sobrenome
- Sobrenome Jedi: as 2 primeiras letras do nome da mãe + as 3 primeiras letras da cidade onde nasceu
- Nome composto: as 3 primeiras letras do nome Jedi + as 3 últimas letras do sobrenome Jedi.

Exemplo:
Nome: **Jos**é da **Si**lva
Nome da mãe: **Ma**ria da Silva
Cidade em que nasceu: **Sor**ocaba

Nome Jedi: Jossi Masor
Nome composto: Jos-Sor

4. CHAMADA POR REFERÊNCIA

4.1 - Definição

Como C passa argumento para funções usando "**chamada por valor**", a função chamada **não** pode alterar diretamente uma variável na função chamadora.

Por exemplo, se quiséssemos permutar dois elementos com uma função chamada **troca( )**. O programa chamador passaria os valores a serem permutados:

troca(a, b);

Porém na definição teríamos:

```
void  troca(int x, int y)          //  x=a  e    y=b
{
  int temp;
  temp = x
  x = y;
  y = temp;
}
```

Por causa da chamada por valor, troca( ) **não** pode afetar os argumentos a e b na rotina que chama. Na verdade, como é feito uma cópia do valor do argumento para a nova variável, só esta é alterada, ou seja, trocamos apenas x e y.

Felizmente, há uma maneira de obter o efeito desejado. O programa chamador passa os endereços dos valores a serem permutados:

troca(&a,&b);

E na função troca( ), os argumentos são declarados como ponteiros, e as variáveis atuais são acessadas através deles.

```
void  troca(int   *ax, int   *ay)          // permuta   *ax   e   *ay
{
   int temp;
   temp = *ax;
   *ax = *ay;
   *ay = temp;
}
```

→ **Para que uma função gere o efeito de chamada por referência, ponteiros devem ser usados na declaração da lista de argumentos e a função chamadora deve mandar endereços como argumentos.**

5. STRINGS E PONTEIROS

Já sabemos que os strings são armazenados em matrizes do tipo char e seu final (que não precisa necessariamente corresponder ao final da matriz) é sinalizado pelo caractere null. Como o final do string está sinalizado, tudo o que precisamos fazer para acessar um string é usar algo que aponte para o seu início.

5.1 - Inicialização de Strings Através de Ponteiros

Temos 2 formas de inicializarmos uma string:

```
char·   mensagem[10 ] =  “Saudacoes”;

char    *mensagem     =  “Saudacoes”;
```

Estas duas formas provocam o mesmo efeito, mas são diferentes:

- a primeira declara mensagem como um ponteiro constante
- a segunda como um ponteiro variável

1000	S
1001	a
1002	u
1003	d
1004	a
1005	c
1006	o
1007	e
1008	s
1009	\0

O valor de um ponteiro constante não pode ser trocado, mas sendo mensagem declarado como ponteiro variável podemos escrever a seguinte expressão:

```
puts( + +mensagem);
```

que mostrará:

```
audacoes
```

As funções podem receber ponteiros, simplesmente declarando-os como argumentos, da seguinte forma:

```
void imprime_linha(char *linha)
{
//no exemplo, linha se comporta exatamente como um string declarado na forma //char linha[n]
        printf("%s\n", linha);
}
```

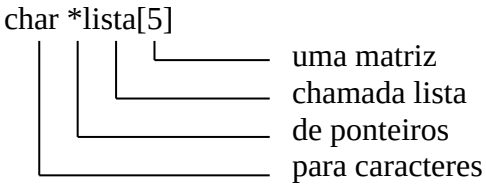
5.2 - Inicialização de uma Matriz de Ponteiros para Strings

No exemplo a seguir, iremos inicializar uma matriz de "strings" e depois modificaremos para que a matriz seja inicializada usando ponteiro.

```
char lista[5][10] =  { "Katarina",
                       "Diogo",
                       "Gustavo",
                       "Francisco",
                       "Airton"};

char *lista[5] =       { "Katarina",
                       "Diogo",
                       "Gustavo",
                       "Francisco",
                       "Airton"};
```

O que significa a expressão char *lista[5] ?



Na versão matriz as strings são guardadas na memória em um retângulo com 5 linhas e 10 colunas. Na nova versão as strings são guardadas de forma contínua na memória, entretanto uma matriz de ponteiros é criada:

5.2.1 - Versão Matriz de Strings

			0	1	2	3	4	5	6	7	8	9
lista[0]	→	1000	K	a	t	a	r	i	n	a	\0	
lista[1]	→	1010	D	i	o	g	o	\0				
lista[2]	→	1020	G	u	s	t	a	v	o	\0		
lista[3]	→	1030	F	r	a	n	c	i	s	c	o	\0
lista[4]	→	1040	A	i	r	t	o	n	\0			

5.2.2 - Versão Ponteiros

lista[0]	→	1000	K	a	t	a	r	i	n	a	\0	
lista[1]	→	1009	D	i	o	g	o	\0				
lista[2]	→	1015	G	u	s	t	a	v	o	\0		
lista[3]	→	1023	F	r	a	n	c	i	s	c	o	\0
lista[4]	→	1033	A	i	r	t	o	n	\0			

Note que a versão ponteiros otimiza o espaço na memória. Os nomes são armazenados a partir da posição 1000 até 1039, enquanto que na versão matriz os nomes são armazenados a partir da posição 1000 até 1049.

Uma das razões para se inicializar strings com ponteiros é a alocação mais eficiente. de memória. Uma outra razão é a de obter maior flexibilidade para manipular matrizes de strings.

OBS.: inicializar uma matriz de strings usando ponteiros aloca menos memória que a inicialização através de matriz.

6. ALOCAÇÃO DINÂMICA

6.1 - Introdução

A única finalidade da matriz do tipo char é alocar espaço suficiente na memória para armazenar strings. E se pudéssemos encontrar algum espaço disponível na memória sem ter que criar uma matriz? Nesse caso, poderíamos usá-lo para armazenar diretamente o string mais o caractere null e bastaria um ponteiro que indicasse a posição do primeiro caractere (para especificar o início do string), exatamente como se o string estivesse dentro de uma matriz previamente definida. Mas como fazer para localizar áreas de memória disponíveis para armazenagem? Há duas maneiras:

- uma aloca espaço para um string literal quando o programa estiver compilado
- usa a função malloc( ) – para alocar espaço na memória enquanto o programa está sendo executado – um processo conhecido como "alocação dinâmica".

6.2 - Alocando Espaço para Strings durante a Compilação

Como foi mencionado anteriormente, o início de um string será indicado através do uso de um ponteiro para uma variável do tipo char, da seguinte maneira:

```
char        *mensagem =  “Saudacoes”;
```

Quando esta instrução for executada, o string Saudacoes (com um caractere null final) será armazenado em algum ponto da memória e o ponteiro mensagem será inicializado para apontar para o seu primeiro caractere.

OBS.: A declaração/inicialização anterior é equivalente a:

```
char mensagem[ ] = "Saudacoes";
```

pois as duas notações, *mensagem e mensagem[ ], são equivalentes e significam "um ponteiro para".

Este método de alocação de espaço para a armazenagem de strings é ótimo quando sabemos o que será necessário enquanto o programa ainda está sendo escrito. Mas, e se as necessidades de armazenagem do programa não forem pré-fixadas, dependendo dos dados fornecidos pelo usuário ou de outros fatores que não possam ser previstos antecipadamente? Nesse caso, usaremos a função malloc( ), que permite alocar espaço de armazenagem "em tempo real".

6.3 - Alocando Espaço para Strings durante a Execução do Programa

A função malloc( ) é uma das funções de alocação da memória em C. Ao chamar esta função, você deve especificar o número de bytes necessários, malloc( ) localiza e reserva um bloco de memória no tamanho apropriado e **retorna o endereço do primeiro byte desse bloco**.

A função malloc( ) retorna um endereço, e seu tipo de retorno é um ponteiro para o tipo void. Por que void ? Porque um ponteiro para dados do tipo void é compatível com todos os tipos de dados, i.e., ele é genérico. Como a memória alocada por malloc( ) pode ser usada para armazenar qualquer dos tipos de dados válidos em C, o tipo de retorno void é o mais apropriado.

6.4 - Heap

O heap ou área de alocação dinâmica consiste de toda a memória disponível que não foi usada para um outro propósito. Em outras palavras, o heap é simplesmente o resto da memória.

A linguagem C oferece um conjunto de funções que permitem a alocação ou liberação dinâmica de memória do heap, como : malloc( ), calloc( ), realloc( ) e free( ).

6.5 - Função malloc( )

Arquivo de cabeçalho: stdlib.h
sintaxe:

```
void*   malloc(tamanho);
```

onde:
tamanho – corresponde a um inteiro sem sinal, que representa a quantidade em bytes de memória requerida.

A cada chamada de malloc( ) devemos informá-la do tamanho do bloco que queremos guardar. Esta informação pode ser conhecida do programador ou podemos utilizar o operador unário chamado **sizeof( )**, que fornece um inteiro, igual ao tamanho, em bytes, da variável ou do tipo do dado em questão.

A função malloc( ) retorna um ponteiro para o primeiro byte do bloco de memória alocado. Quando é incapaz de alocar a quantidade solicitada de memória, seu retorno é nulo (NULL). Sempre que tentarmos alocar memória, mesmo que seja uma pequena quantidade, devemos verificar o valor de retorno.

Por exemplo, podemos usar malloc( ) para alocar a memória para armazenar uma única variável do tipo char. Em primeiro lugar, declaramos um ponteiro para o tipo char:

```
char *ptr;
```

A seguir, chamamos malloc( ) passando o tamanho do bloco de memória desejado. Como um tipo char ocupa somente um byte, precisamos de um bloco de um byte. O valor retornado por malloc ( ) é atribuído ao ponteiro:

```
ptr = (char*) malloc(1);
```

Esta instrução aloca um bloco de memória com apenas um byte e atribui seu endereço ao ponteiro ptr. Ao contrário das variáveis que são declaradas no programa, esse byte de memória **não tem qualquer nome** e só pode ser referenciado através do ponteiro. Por exemplo, para armazenar o caractere 'x' nesse byte, devemos escrever:

```
*ptr = 'x';
```

A alocação de strings usando malloc( ) é quase idêntica a de uma única variável do tipo char. A principal diferença é que precisamos saber antecipadamente quantos bytes deverão ser alocados - ou seja, qual será o número máximo de caracteres no string. Esse valor máximo depende das necessidades do programa. Para exemplificar, iremos alocar um string de 99 caracteres, além de um caractere null de término, o que resulta em 100 bytes. Em primeiro lugar, teríamos que declarar um ponteiro para o tipo char, e depois chamar malloc( ).

```
char *ptr;
ptr = (char *) malloc(100);
```

Agora ptr está apontando para um bloco reservado de 100 bytes, que pode ser usado para a armazenagem e manipulação de strings. Podemos usar o ponteiro ptr exatamente como faríamos se o programa tivesse alocado explicitamente o mesmo espaço através de uma declaração de matriz:

```
char ptr[100];
```

O uso de malloc( ) permite que o programa aloque espaço conforme seja necessário. Evidentemente, o espaço disponível para armazenagem não é ilimitado; ele depende tanto da quantidade de memória instalada no computador quanto das outras necessidades de armazenagem do próprio programa. Se a memória disponível não for suficiente, malloc( ) retornará um valor 0 (nulo). Portanto, devemos **testar** o valor de retorno de malloc ( ) para ter certeza de que a memória solicitada foi alocada adequadamente. O valor de retorno de malloc( ) deve sempre ser testado contra a constante simbólica NULL que é definida em **stdlib.h**.

Obs.:
1) Nunca devemos presumir que a função malloc ( ) tenha conseguido **reservar** a memória que lhe pedimos. Pois de fato, não ordenamos à função que aloque memória; apenas lhe perguntamos se isso pode ser feito.

2) O que significa a expressão (char *) ou (int *) ou (float *) precedendo malloc( ) ?

A função malloc( ) retorna um ponteiro para o tipo void, portanto esse ponteiro deve ser moldado para o tipo apropriado antes de ser usado. E para isso usamos a conversão de tipos, na verdade um operador unário chamado **operador de molde**, que consiste em colocar um parênteses envolvendo o tipo de dado desejado.
Ex.: A função sqrt( ), retorna a raiz quadrada de um número do tipo double e temos uma variável float:

```
float   n;
double  resposta;

resposta = sqrt ((double)n);          //  converte n antes de usá-lo na função sqrt
```

Portanto, devemos indicar que o valor retornado por malloc( ) será do tipo ponteiro para char ou int ou float.

Exemplo 1 – aloca memória para um string com 99 caracteres. (100 = 99 caracteres + 1 NULL)

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

char        *str;

main( )              {
str  =  (char  *)  malloc(100);
if  (str  = =  NULL)
        printf("Espaço insuficiente para alocar buffer \n");
else
        printf("String foi alocado");

}// main
```

Exemplo 2 – aloca memória para uma matriz com 50 valores do tipo int.

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

int *numeros;

main( )
{
if  ( (numeros  =  (int  *) malloc(50  *  sizeof(int) ) )  = =  NULL)
        printf("Espaço insuficiente para alocar buffer \n");
else
        printf("Matriz foi alocada");
}// main
```

Exemplo 3 – aloca memória para uma matriz com 10 valores do tipo float.

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

float *numeros;

main( )
{
if  ( (numeros  =  (float  *) malloc(10  *  sizeof(float) ) )  = =  NULL)
        printf("Espaço insuficiente para alocar buffer \n");
else
        printf("Matriz foi alocada");
}// main
```

Exemplo 4 – programa usando malloc( ), que preenche um string com os códigos ASCII para A-Z (valores 65 a 90). Temos 26 letras + 1 NULL

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

char contagem, *ptr, *p;

main( )
{
ptr =   (char   *) malloc(27 * sizeof(char));
if    (ptr   = =   NULL)
          puts(“Erro na alocação de memória.");
else
    {
      p = ptr;          // p é um ponteiro usado para avançar através do string, pois ptr deve
                        //permanecer apontando para o início
     for (contagem = 65; contagem < 91; contagem++)
          *p++ = contagem;
    *p = ‘\0’;                          //  acrescenta o caractere null de encerramento
    puts(ptr);
    }//else
}//main
```

→ **O ponteiro** *ptr* **aponta para o valor inicial (letra A) e utilizamos outro ponteiro** *p* **para inserir valores no espaço reservado, ou seja, incrementamos** *p* **e preservamos** *ptr*, **caso contrário perderíamos o valor inicial da memória alocada. Devemos lembrar que este bloco de memória** *não contém nome*, **e o programa apenas sabe onde ele se encontra.**

6.6 - Função calloc( )

A função calloc ( ) ao invés de alocar um grupo de bytes como malloc ( ), ela aloca memória para um grupo de objetos.

Arquivo de cabeçalho: stdlib.h
sintaxe:

```
void*       calloc(num, tamanho);
```

onde:
num - corresponde a um inteiro sem sinal, é o número de objetos para os quais a memória deve ser alocada.
tamanho - corresponde a um inteiro sem sinal, é o tamanho de cada objeto em bytes.

Se a alocação for bem-sucedida, toda a memória alocada é inicializada com o valor 0 e a função retorna um ponteiro para **void** para o primeiro byte. Se a alocação fracassar ou se num ou size forem iguais a zero, a função retorna NULL.

→ **Devemos usar o operador** *molde* **se quisermos um ponteiro para um tipo diferente.**

Exemplo 1 - aloca memória para 100 elementos do tipo long

```
long        *ptr;
ptr  =  (long *)  calloc (l00, sizeof (long));
if  (ptr  = =  NULL)
        printf("Espaço insuficiente  \n");
else
        printf("Memoria alocada");
```

O primeiro argumento é o número de células de memória desejada. O segundo argumento é o tamanho de cada célula em bytes.

Neste caso, long usa quatro bytes, então esta instrução alocará espaço para 100 unidades de quatro bytes ou seja 400 bytes.

Exemplo 2 – programa que aloca a quantidade de variáveis int estipulada pelo usuário.

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

main()
{
unsigned num;
int *ptr;

printf("Digite o numero de variáveis do tipo int: ");
scanf("%i", &num);
ptr  =  (int  *) calloc(num, sizeof(int));
if (ptr  !=  NULL)
        puts("Memória alocada sem problemas.");
else
        puts("Erro na alocação de memória.");
}//main
```

Este programa recebe um valor do usuário. Esse número determina a quantidade de espaço a ser alocado. O programa tenta alocar memória suficiente para conter o número especificado de variáveis. Se a alocação fracassar, o valor de retorno de calloc( ) é NULL; caso contrário, é um ponteiro para a área de memória alocada. Neste programa, o valor de retorno de calloc ( ) é colocado em um ponteiro para o tipo int chamado ptr. Uma instrução if verifica o resultado da alocação com base no valor de ptr e imprime uma mensagem apropriada.

Digite diferentes valores e veja qual é o máximo de memória que você consegue alocar sem problemas. Esse limite depende, até certo ponto, da configuração do sistema. Em alguns sistemas, a alocação de espaço para até 25.000 ocorrências do tipo int é feita sem problemas, mas a alocação de espaço para 30.000 fracassa.

6.7 - Função realloc( )

Muda o tamanho de um bloco de memória que tenha sido alocado anteriormente com malloc( ) ou calloc( ).

Arquivo de cabeçalho: stdlib.h
sintaxe:
void* realloc(*ptr, tamanho);

onde:
ptr – corresponde a um ponteiro para o tipo void, aponta para o bloco original de memória.
tamanho - corresponde a um inteiro sem sinal, especifica o novo tamanho desejado, em bytes.

Os resultados possíveis do uso realloc( ) são:

- se houver espaço suficiente para expandir o bloco de memória referenciado por *ptr*, a memória adicional é alocada e a função retorna *ptr*.
- se não houver espaço suficiente para expandir o bloco atual, um novo bloco do tamanho especificado em *tamanho* é alocado e os dados existentes são copiados do bloco original para o início do novo bloco. A seguir, o bloco original é liberado e a função retorna um ponteiro para o novo bloco.
- se o argumento *ptr* for NULL, a função atua como malloc( ), alocando um bloco de *tamanho* bytes e retornando um ponteiro para ele.
- se o argumento *tamanho* for 0, a memória indicada por *ptr* é liberada e a função retorna NULL.
- se não houver memória suficiente para a realocação (tanto expandindo o bloco original quanto alocando um novo bloco), a função retorna NULL e o bloco original permanece inalterado.

Exemplo:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
main( )
{
char buf[80], *mensagem;

puts("Digite uma linha de texto");
gets(buf);                                  //  recebe um string
mensagem  =  (char  *) realloc (NULL,  strlen (buf )+1);
strcpy(mensagem,  buf );
puts(mensagem);                             //  exibe a mensagem
puts("Digite outra linha de texto”);
gets(buf);
mensagem = (char  *) realloc(mensagem, (strlen(mensagem)+strlen(buf)+1));//aumenta memo
strcat(mensagem, buf);                      //  concatena o novo string
puts(mensagem);
}// main
```

Este programa recebe um string. O string é lido e armazenado em uma matriz de caracteres chamada buf. A seguir, esse valor é copiado para uma localização da memória indicada por mensagem, que foi alocada usando realloc( ).

→ realloc( ) foi chamada embora não tenha havido qualquer alocação anterior; quando recebe NULL como primeiro parâmetro, a função sabe que esta é a primeira alocação.

Um segundo string é recebido e inserido no buffer de texto. Esse string é concatenado ao string que já está armazenado em mensagem. Como mensagem só tem espaço suficiente para conter o primeiro string, é necessário realocar esse espaço para que ele possa conter simultaneamente os dois strings. Ao terminar, o programa imprime o string concatenado final.

6.8 - Função free( )

Ao usarmos malloc( ), calloc( ) ou realloc( ) alocamos parte do total de memória dinâmica disponível para o programa. Esse total, que costuma ser chamado de *heap*, é finito. Portanto, quando o programa já não estiver utilizando algum bloco da memória alocada, aconselhamos "desalocar", ou liberar, essa memória, tornando-a disponível para alocações futuras. Para liberarmos a memória que foi alocada dinamicamente, usamos a função free ( ).

Arquivo de cabeçalho: stdlib.h
sintaxe:

```
void   free (*ptr);
```

onde:
ptr – corresponde a um ponteiro para o tipo void, aponta para o início do bloco de memória a ser liberado.

A função free ( ) libera a memória referenciada por ptr. Essa memória deve ter sido alocada usando malloc( ), calloc( ) ou realloc( ). Se ptr for NULL, free ( ) não faz nada.
A função free( ) declara o seu argumento como um ponteiro para void. A vantagem desta declaração é que ela permite que a chamada à função seja feita com um argumento ponteiro para qualquer tipo de dado.

Exemplo1 - libera memória alocada por função desenvolvida em calloc( ).

```
liberamem()
{
long *ptr, *alocamem();

  ptr  =  alocamem();        // alocamem() - função que aloca memória usando calloc( ) – exemplo2
  free(ptr);
  }
```

Exemplo2 - este programa tenta alocar dois blocos de memória.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define  TAMBLOCO  30000

main( )
{
void   *ptrl,   *ptr2;

ptr1   =   malloc(TAMBLOCO);                    //   tenta alocar o 1º bloco
if  (ptr1  = =  NULL)
          printf("\n Tentativa de alocar %i bytes falhou", TAMBLOCO);
else
  {
  printf("\n Primeira alocação de %i bytes realizada", TAMBLOCO);
  ptr2  =  malloc(TAMBLOCO);            //  tenta alocar o 2º bloco
  if  (ptr2  !=  NULL)
     printf("\n Segunda alocação de %i bytes realizada", TAMBLOCO);
  else
    {
    printf("\n Segunda tentativa de alocar %i bytes falhou", TAMBLOCO);
    free (ptr1);
    printf("\n Liberando o primeiro bloco”);
    ptr2 = malloc(TAMBLOCO);
    if  (ptr2  !=  NULL)
          printf("\n Depois de free(), alocação de %i bytes realizada", TAMBLOCO);
    }// else
  }// else
}// main
```

Este programa tenta alocar dinamicamente dois blocos de memória, usando a constante TAMBLOCO para determinar o tamanho de cada bloco. Porém, a 2ª alocação só ocorrerá se a 1ª for bem sucedida.

- ptr1 corresponde a 1º alocação. Determinamos o resultado da alocação verificando se o valor de retorno foi NULL. Se a alocação não foi bem sucedida (valor de retorno = NULL) o programa é encerrado. Caso contrário, tentaremos alocar um 2º bloco de memória.

- ptr2 corresponde a 2ª alocação. Novamente verificamos o resultado, porém agora se o procedimento foi bem sucedido (valor de retorno diferente de NULL). Se a 2º alocação ocorreu sem problemas, o programa é finalizado, caso contrário, o primeiro bloco é liberado com free ( ) e é feita uma nova tentativa de alocação.

6.9 – Alocação de Memória na Main X na Função

Para alocarmos memória dinamicamente na **main**, devemos declarar um ponteiro e utilizar malloc(), calloc() ou realloc(). Porém, quando utilizamos **função**, devemos lembrar que a memória não pode ser "conhecida" apenas na função, mas também na main. Portanto, devemos declarar um ponteiro na main e a função pode ser feita de 2 formas:

- **Chamada por Referência** – devemos passar o endereço do ponteiro declarado na main, portanto, a função receberá como parâmetro ponteiro para ponteiro.
- **Chamada por Valor** – só conseguimos trabalhar dessa forma, se a função **retornar o endereço** alocado para o ponteiro declarado na main, permitindo assim, o acesso deste à memória alocada. Caso contrário o ponteiro na main continuará NULL e ao sairmos da função perdemos a referência da memória alocada. Lembre-se que as variáveis (inclusive ponteiro) declaradas dentro da função, deixam de existir assim que a função termina.

Versão utilizando Chamada por Referência

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

void aloca(int **p, int tam);              //passa o endereço do ponteiro declarado na main

main( )
{
int *ptr=NULL;
printf("\nAloca memoria na Funcao e na Main\n");
printf("\nChamada por Referencia - passa ENDERECO do ponteiro\n");
printf("\nFuncao main - antes de alocar");
printf("\nEndereco ptr = %u \nConteudo %u = %u",&ptr,&ptr,ptr);
aloca(&ptr, 1);                                  //chamada por referencia
printf("\n\nFuncao main - depois de alocar");
printf("\nEndereco ptr = %u \nConteudo %u = %u\n\n\n",&ptr,&ptr,ptr);
system("pause");
}//main

void aloca(int **p, int tam)
{
printf("\n\nFuncao aloca - antes de alocar");
printf("\nEndereco p = %u \tConteudo p = %u  (Endereco ptr)",&p,p);
printf("\nConteudo %u = %u\n",p,*p);
if((*p=(int*)realloc(*p, tam*sizeof(int)))== NULL)
  {
   printf("Erro de alocacao");
   exit(1);
  }
printf("\n\nFuncao aloca - depois de alocar");
printf("\nEndereco p = %u \tConteudo p = %u  (Endereco ptr)",&p,p);
printf("\nConteudo %u = %u\n",p,*p);
}//aloca
```

Versão utilizando Chamada por Valor – com RETORNO do endereço alocado

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

int*        aloca(int *p, int tam);                    //retorna o endereço da memória alocada

main( )
{
int *ptr=NULL;
printf("\nAloca memoria na Funcao e RETORNA para a Main\n");
printf("\nChamada por Valor - passa CONTEUDO do ponteiro\n");
printf("\nFuncao main - antes de alocar");
printf("\nEndereco ptr = %u \nConteudo %u = %u",&ptr,&ptr,ptr);
ptr = aloca(ptr, 1);                                   //chamada por valor
printf("\n\nFuncao main - depois de alocar");
printf("\nEndereco ptr = %u \nConteudo %u = %u\n\n\n",&ptr,&ptr,ptr);
system("pause");
}//main

int*        aloca(int *p, int tam)
{
printf("\n\nFuncao aloca - antes de alocar");
printf("\nEndereco p = %u \nConteudo p = %u   (Conteudo ptr)",&p,p);
if((p=(int*)realloc(p, tam*sizeof(int)))== NULL)
  {
   printf("Erro de alocacao");
   exit(1);
  }
printf("\n\nFuncao aloca - depois de alocar");
printf("\nEndereco p = %u \nConteudo p = %u   (Conteudo ptr)",&p,p);
return p;
}//aloca
```

ERRO: Versão utilizando Chamada porValor – sem RETORNO do endereço alocado

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

void aloca(int *p, int tam);                  //NÃO retorna o endereço da memória alocada

main()
{
int *ptr=NULL;
printf("\nERRO pois aloca memoria apenas na Funcao e nao na Main\n");
printf("\nChamada por Valor - passa CONTEUDO do ponteiro\n");
printf("\nFuncao main - antes de alocar");
```

```
printf("\nEndereco ptr = %u \nConteudo %u = %u",&ptr,&ptr,ptr);
aloca(ptr, 1);                                          //chamada por valor
printf("\n\nFuncao main - depois de alocar");
printf("\nEndereco ptr = %u \nConteudo %u = %u\n\n\n",&ptr,&ptr,ptr);  //ptr continua NULL
system("pause");
}//main

void aloca(int *p, int tam)
{
printf("\n\nFuncao aloca - antes de alocar");
printf("\nEndereco p = %u \nConteudo p = %u   (Conteudo ptr)",&p,p);
if((p=(int*)realloc(p, tam*sizeof(int)))== NULL)
  {
  printf("Erro de alocacao");
  exit(1);
  }
printf("\n\nFuncao aloca - depois de alocar");
printf("\nEndereco p = %u \nConteudo p = %u",&p,p);
}//aloca
```

Exemplo de Alocação de Memória na Main X na Função

Alocar espaço para 10 números inteiros. Mostrar o endereço e receber do usuário os valores. Em seguida mostrar novamente o endereço e o respectivo valor de cada elemento.

Versão para Alocação feita na main( )

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

void recebe (int *p, int tam);
void imprime (int *p, int tam);

main( )
{
int *ptr=NULL;

if((ptr = (int *) realloc(ptr, 10 * sizeof(int))) == NULL)
   {
    printf(“Erro na alocação);
    exit(1);
   }
recebe(ptr,10);
imprime(ptr,10);
system(“pause”);
}//main
```

```
void recebe (int *p, int tam)
{
int    i;

printf("\nFuncao recebe – Digite os valores \n");
for(i=0;i<tam;i++)
   {
   printf("Endereco: %u – Valor = ",p+i);
   scanf("%i",p+i);
   }
}//recebe

void imprime (int *p, int tam)
{
int    i;

printf("\nFuncao imprime – Mostra os valores \n");
for(i=0;i<tam;i++)
   printf("Endereco: %u – Valor = %i",p+i,*(p+i));
}//imprime
```

Versão para Alocação feita por função( )

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

void aloca (int **p, int tam);       //ponteiro p/ ponteiro – recebe endereço do ponteiro
void recebe (int *p, int tam);
void imprime (int *p, int tam);

main( )       {
int *ptr=NULL;
aloca(&ptr,10);                      //chamada por referência – endereço do ptr
recebe(ptr,10);
imprime(ptr,10);
system("pause");
}//main

void aloca (int **p, int tam)
{
if((*ptr = (int *) realloc(*ptr, 10 * sizeof(int))) == NULL)
   {
    printf("Erro na alocação);
    exit(1);
   }
printf("\nFuncao aloca – Endereco: %u", *p);
}//aloca
```

```
void recebe (int *p, int tam)
{
int    i;
printf("\nFuncao recebe – Digite os valores \n");
for(i=0;i<tam;i++)
   {
   printf("Endereco: %u – Valor = ",p+i);
   scanf("%i",p+i);
   }
}//recebe

void imprime (int *p, int tam)
{
int    i;
printf("\nFuncao imprime – Mostra os valores \n");
for(i=0;i<tam;i++)
   printf("Endereco: %u – Valor = %i",p+i,*(p+i));
}//imprime
```

6.10 – Exercícios Propostos

1. Alocar espaço para 10 números reais. Mostrar o endereço e receber do usuário os valores. Em seguida, alterar esses valores, somando 30 a cada elemento. Mostrar novamente o endereço (que deve ser o mesmo) com o novo valor. **Utilizar função para alocação dinâmica.**

2. Aloque espaço para uma seqüência de números reais escolhida pelo usuário. Receba os valores e imprima-os em ordem inversa. **Utilizar função para alocação dinâmica.**

3. Idem ao exercício anterior, porém a seqüência de números reais deverá ser digitada pelo usuário, até que este escolha a opção **N** (Deseja continuar? (S/**N**)). **Utilizar função para alocação dinâmica.**

4. Aloque dinamicamente a seqüência de números reais digitados pelo usuário, até que este escolha a opção **N** (Deseja continuar? (S/**N**)). Em seguida, exiba a média dos elementos (não considerar as ocorrências com valor = 0). **Utilizar função para alocação dinâmica.**

5. Alocar espaço para armazenar as temperaturas diárias registradas durante um mês. O programa deve mostrar **diariamente** (a medida que as temperaturas são incluídas) a maior temperatura e o dia em que ocorreu. **Utilizar função para alocação dinâmica.**

6. Alocar espaço para 10 letras. Mostrar o endereço e receber do usuário as letras. Em seguida, alterá-las da seguinte maneira: se o usuário digitou MAIÚSCULA deverá transformar para minúscula e vice-versa. Mostrar novamente o endereço (que deve ser o mesmo) com as letras modificadas. **Utilizar função para alocação dinâmica.**

   Dica: valores em ASCII

A = 65	a = 97	MAIÚSCULA p/ minúscula = +32
Z = 90	z = 122	minúscula p/ MAIÚSCULA = -32

7. Receber do usuário seu primeiro nome em uma string auxiliar, alocar espaço suficiente para armazenar o nome. Mostrar o endereço e o nome armazenado. Em seguida, alterar o nome da seguinte maneira: 1ª letra com a última, a 2ª com a penúltima e assim sucessivamente (utilizar uma variável auxiliar). Mostrar novamente o endereço (que deve ser o mesmo) com o nome modificado. **Utilizar função para alocação dinâmica.**

8. Aloque dinamicamente espaço e receba ‘nomes’ digitados pelo usuário. O usuário poderá entrar com novos ‘nomes’, até que este escolha a opção **N** (Deseja continuar? (S/**N**)). Em seguida, determine quantos nomes foram alocados e quantos começam com a letra ‘A’ (maiúscula). Mostre na tela. **Utilizar função para alocação dinâmica.**

7. ESTRUTURAS

7.1 – Introdução

Quando nos deparamos com um problema onde desejamos agrupar um conjunto de tipos de dados **não similares** sob um único nome, nosso primeiro impulso seria usar uma matriz. Porém, como matrizes requerem que todos os seus elementos sejam do **mesmo** tipo, provavelmente forçaríamos a resolução do problema selecionando uma matriz para cada tipo de dado, resultando em um programa ineficiente.

O problema de agrupar dados desiguais em C é resolvido pelo uso de estruturas

Uma estrutura é uma coleção de uma ou mais variáveis, possivelmente de tipos diferentes, colocadas juntas sob um único nome. (Estruturas são chamadas "registros" em algumas linguagens). É uma outra maneira de representação de dados em C, que servirá para a elaboração de um arquivo de dados na memória acessado através de uma lista encadeada.

Um exemplo tradicional de uma estrutura é o registro de uma folha de pagamento, onde um funcionário é descrito por um conjunto de atributos tais como:

- nome (string)
- nº do seu departamento (int)
- salário (float) e assim por diante.

Provavelmente, haverá outros funcionários, e você vai querer que o seu programa os guarde formando uma matriz de estruturas.

O uso de uma matriz de várias dimensões não resolveria o problema, pois todos os elementos de uma matriz devem ser de um tipo único, portanto deveríamos usar várias matrizes:

- uma de caracteres para os nomes
- uma de inteiros para número do departamento
- uma float para os salários e assim por diante.

Esta não seria uma forma prática de manejar um grupo de características que gostaríamos que tivessem um único nome: **funcionário**.

Uma estrutura consiste de um certo número de itens de dados, chamados membros da estrutura, que não necessitam ser de mesmo tipo, agrupados juntos.

7.2 – Definindo um Tipo Estrutura

Primeiro, devemos definir o tipo da estrutura que queremos criar. Uma estrutura pode conter qualquer número de membros de diferentes tipos. O programa deve avisar o compilador de como é formada uma estrutura antes de seu uso.

sintaxe:

```
struct   nome_estrutura{
         membros_estrutura;
         };
```

onde:
nome_estrutura – rótulo ou etiqueta da estrutura, segue as mesmas regras de nomes de variáveis.
membros_estrutura – lista de variáveis, contém o tipo e o nome de cada variável da estrutura.
Ex.:

```
struct  facil  {
        int      num;
        char     c;
        };
```

Estas instruções definem um novo tipo de dado chamado **struct facil**. Cada variável deste tipo será composta por dois elementos:

- uma variável inteira chamada **num**
- e uma variável char chamada **c**

Esta instrução não declara qualquer variável, e então não é reservado nenhum espaço de memória. Somente é mostrado ao compilador como é formado o tipo struct facil.

A palavra **struct** informa ao compilador que um tipo de dado está sendo declarado e o nome **facil** é chamado rótulo e nomeia a estrutura que está sendo definida.

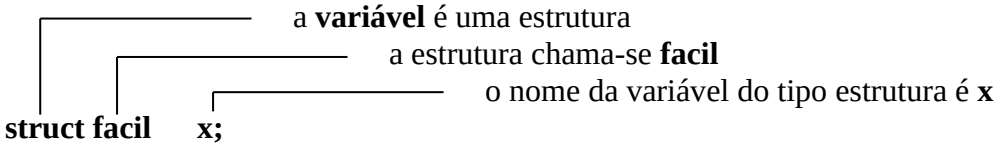
O rótulo não é o nome de uma variável, mas é o nome de um tipo. Os membros da estrutura devem estar entre chaves, e a instrução termina por ponto e vírgula.

UMA ESTRUTURA É UM TIPO DE DADO CUJO FORMATO É DEFINIDO PELO PROGRAMADOR.

7.3 - Declarando as Variáveis do Tipo Estrutura

Após definirmos nosso novo tipo de dado. Podemos, então, declarar uma ou mais variáveis deste tipo.

Ex.:



Em nosso exemplo, declaramos a variável **x** como sendo do tipo **struct facil**.
Esta instrução executa uma função similar às declarações de variáveis que já conhecemos como:

```
float    f;
int      num;
```

Ela solicita ao compilador a alocação de espaço de memória suficiente para armazenar a variável **x** que é do tipo **struct facil**, neste caso 5 bytes (4 bytes para o inteiro e 1 byte para o caractere).

7.4 - Definindo e Declarando Estruturas

Se preferirmos, podemos definir e declarar a estrutura de uma única vez.

Ex.:

```
struct   facil  {
         int      num;
         char     c;
         }x;
```

Esta instrução define o tipo de estrutura **facil** e declara uma estrutura desse tipo, chamada **x**.

Dizemos que x é instância do tipo facil e contém 2 membros, um int chamado num e outro char chamado c

OBS.: C permite definir explicitamente **novos nomes** aos tipos de dados, utilizando a palavra-chave **typedef**. Declarações com typedef **não produzem novos tipos de dados**, apenas cria sinônimos.

sintaxe:

```
typedef        tipo_existente        novonome;
```

Exemplo 1:

```
typedef        float    real;    //o compilador reconhece real como outro nome para float
```

portanto, podemos usar:

```
real     a, b;                    // declaração de 2 variáveis do tipo real (float)
```

Exemplo 2:

```
struct   facil  {
         int      num;
         char     c;
         };
typedef   struct facil    facil;   //o compilador reconhece facil como outro nome p/ struct facil
facil    x;
```

ou

```
typedef struct facil {
         int      num;
         char     c;
         }facil;

facil    x;
```

ou

```
typedef struct {                   //sem etiqueta
         int      num;
         char     c;
         }facil;

facil    x;
```

7.5 - Acessando Membros da Estrutura

Agora que já criamos uma variável do tipo estrutura, precisamos acessar (referenciar) os seus membros.

Quando usamos uma matriz, podemos acessar um elemento individual através do índice: matriz[7].

Estruturas usam uma maneira de acesso diferente: o **operador ponto (.)**, que é também chamado **operador de associação**.

Ex.: A sintaxe apropriada para referenciar num que é parte da estrutura x, é:

x.num

O nome da variável que precede o ponto é o nome da estrutura e o nome que o segue é o de um membro específico da estrutura. As instruções:

```
x.num  =  2;        // atribui 2 ao membro num da estrutura x
x.c  =  'Z'         // atribui 'Z' ao membro c da estrutura x
```

O OPERADOR (.) CONECTA O NOME DA VARIÁVEL ESTRUTURA A UM MEMBRO DA ESTRUTURA.

Para exibir na tela:

```
printf("x.num  =  %i, x.c  =  %c  \n", x.num, x.c);
```

A saída será:

```
x.num  =  2, x.c  =  Z
```

7.6 - Múltiplas Estruturas de Mesmo Tipo

Do mesmo modo como podemos ter várias variáveis do tipo int em um programa, podemos também ter qualquer número de variáveis do tipo de uma estrutura predefinida.

Ex.: Vamos declarar duas variáveis, **x1** e **x2**, do tipo **struct facil**.

```
struct   facil  {
         int      num;
         char     c;
         };
struct facil      x1, x2;
```

ou, se preferir:

```
struct   facil  {
         int      num;
         char     c;
         }x1, x2;
```

O efeito é o mesmo, e o programa torna-se mais compacto.

7.7 - Definição de Estruturas Sem Rótulo ou Etiqueta

A convenção normal é a de usar a etiqueta da estrutura quando a expectativa é criar várias variáveis do mesmo tipo estrutura. Porém, se você espera usar uma única declaração de variável do tipo estrutura, você pode combinar a declaração com a definição da estrutura e omitir a etiqueta:

```
struct  {
        int     num;
        char    c;
        } x1, x2;
```

7.8 - Estruturas que Contém Matrizes

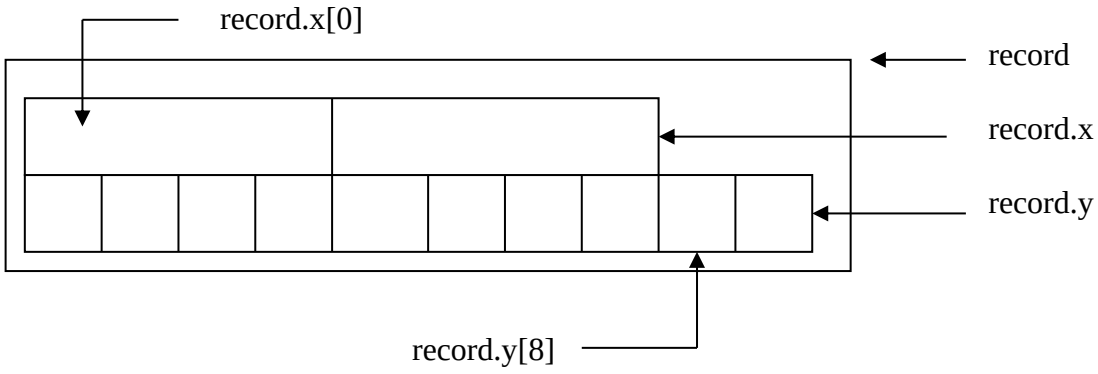
Podemos definir uma estrutura cujos membros incluam uma ou mais matrizes. A matriz pode ser de qualquer tipo de dados válido em C (char, int, float, double, etc.).
Ex.:

```
struct data{
        int     x[2];
        char    y[10];
        };
```

Definem uma estrutura do tipo **data** cujos membros são uma matriz inteira com 2 elementos chamada x e uma matriz de caracteres com 10 elementos chamada y. A seguir, poderíamos declarar uma estrutura chamada **record** baseada no tipo data:

```
struct data     record;
```

A organização desta estrutura é mostrada a seguir:



Observe que nessa figura os elementos da matriz x ocupam o quádruplo do espaço dos elementos da matriz y. Isto acontece porque, um elemento do tipo int normalmente exige 4 bytes, ao passo que um elemento do tipo char normalmente só exige 1 byte.

O acesso aos elementos individuais de uma matriz que é um membro de uma estrutura é feito através de uma combinação de operadores de membros e subscritos de matrizes:

```
record.x[0]  =  100;
record.y[1]  =  'x';
```

O nome de uma matriz, quando é usado sem os colchetes, é um ponteiro para essa matriz. Como isto também se aplica às matrizes que são membros de estruturas, a expressão

```
record.y
```

é um ponteiro para o primeiro elemento da matriz y dentro da estrutura record. Podemos, portanto, imprimir o conteúdo de y [ ] na tela usando a instrução

```
puts(record.y);
```

7.9 – Exemplo - Criando uma Lista de Livros

A lista de livros compreende uma variedade de informações como:

- título
- autor
- editora
- número de páginas
- número do registro de biblioteca
- preço, etc.

Nosso objetivo é desenvolver um programa que controle um arquivo simples e mostre um dos mais úteis caminhos de organização de dados em C: uma matriz de estruturas.

Para simplificar o problema, a nossa lista será formada, a princípio, somente pelo título do livro, representado por uma matriz de caracteres e o número do registro de biblioteca representado por um inteiro. Estas informações são fornecidas pelo usuário através da entrada padrão (teclado) e depois impressas na saída padrão (vídeo).

Ex.: Na primeira versão a nossa lista será limitada em apenas dois livros.

```
#include <stdio.h>
#include <stdlib.h>                    // função atoi( )
struct   livro  {
      char        titulo[30];
      int         regnum;
      };
main( )
{
struct livro      livro1,livro2;
char numstr[8];
printf("\n Livro  1  \n Digite titulo: ");
gets(livro1.titulo);
printf("Digite o numero do registro (3 digitos): ");
gets(numstr);
livro1.regnum  =  atoi(numstr);
```

```
printf("\n Livro  2  \n Digite titulo: ");
gets (1ivro2.titulo);
printf("Digite o numero do registro (3 digitos): ");
gets(numstr);
1ivro2.regnum  =  atoi (numstr);
printf("\n Lista de livros:\n");
printf("   Titulo: %s\n",livrol.titulo);
printf("   Numero do registro: %3i \n",livrol.regnum);
printf("   Titulo: %s\n",livro2.titulo);
printf("   Numero do registro: %3i \n",livro2.regnum);
}//main
```

Uma simples execução do programa terá a seguinte saída:

```
Livro  1.
Digite titulo: Helena
Digite o numero do registro (3 digitos): 102
Livro  2.
Digite titulo: Iracema
Digite o numero do registro (3 digitos): 321

Lista de livros:
Titulo: Helena
Numero do registro: 102
Titulo: Iracema
Numero do registro: 321
```

Poderíamos ter usado a função **scanf()** para ler o número do registro dos livros, como na instrução:

```
scanf("%i", &livro1.regnum);
```

que surtiria o mesmo efeito do uso de **gets()** e **atoi()**. Porém, preferimos usar gets() e atoi(), pois não provocam problemas com o buffer do teclado, se o usuário digitar espaços antes de digitar o número.

A função atoi( ), converte uma string ASCII num inteiro correspondente.

7.10 - Inicializando Estruturas

Nós já aprendemos como inicializar variáveis simples e matrizes:

```
int      num  =  5;
int      mat[ ]  =  {  1,  2,  3,  4,  5  };
```

Ex.: Uma versão modificada do programa, em que os dados dos 2 livros estão contidos na instrução de inicialização dentro do programa, em vez de serem solicitados ao usuário.

```
struct  livro  {
        char titulo[30];
        int regnum;
        };
```

```
main()
{
struct livro      livro1  =  {"Helena",102};
struct livro       livro2  =  {"Iracema",321};

printf("\nLista de livros:\n");
printf("Titulo: %s \n", livro1.titulo);
printf("Numero do registro: %3i \n", livrol.regnum);
printf("Titulo: %s \n", livro2.titulo);
printf("Numero do registro: %3i \n", livro2.regnum);
} // main
```

Aqui, depois da declaração usual do tipo estrutura, as duas variáveis estrutura são declaradas e inicializadas.

Da mesma forma como inicializamos matrizes, o sinal de igual é usado e em seguida a abertura da chave que irá conter a lista de valores. Os valores são separados por vírgulas.

7.11 - Atribuições entre Estruturas

Na versão original do C definida por Kernighan e Ritchie, é impossível atribuir o valor de uma variável estrutura a outra do mesmo tipo, usando uma simples expressão de atribuição.

Nas versões mais modernas de C, esta forma de atribuição já é possível. Isto é, se **livro1** e **livro2** são variáveis estrutura do mesmo tipo, a seguinte expressão pode ser usada:

livro2 = livro1;

O VALOR DE UMA VARIÁVEL ESTRUTURA PODE SER ATRIBUÍDO A OUTRA VARIÁVEL ESTRUTURA DO MESMO TIPO.

Quando executamos esta atribuição, os dados para os 2 livros serão exatamente os mesmos.

QUANDO ATRIBUÍMOS UMA ESTRUTURA A OUTRA, TODOS OS VALORES NA ESTRUTURA ESTÃO REALMENTE SENDO ATRIBUÍDOS PARA OS CORRESPONDENTES ELEMENTOS DA OUTRA ESTRUTURA.

Uma expressão de atribuição tão simples não pode ser usada para matrizes, que devem ser atribuídas elemento a elemento.

7.12 - Estruturas Aninhadas - Estruturas que contém Estruturas

Exatamente como podemos ter matrizes de matrizes, podemos ter estruturas que contêm outras estruturas. O que pode ser um poderoso caminho para a criação de tipos de dados complexos.

Como exemplo, imagine que os nossos livros são divididos em grupos, consistindo de um "dicionário" e um livro de "literatura".

Ex.: O programa cria uma estrutura de etiqueta **grupo**, que consiste de duas outras do tipo livro.

```
struct   livro  {
         char    titulo[30];
         int     regnum;
         };
struct   grupo  {
         struct livro     dicionario;
         struct livro     literatura;
         };
main()
{
struct grupo    grupo1 = {      {"Aurelio",134},
                                {"Iracema",321}       };
printf("\n Dicionario:\n");
printf("Titulo: %s \n", grupo1.dicionario.titulo);
printf("Nº do registro: %3i \n", grupo1.dicionario.regnum);
printf("\n Literatura:\n");
printf("Titulo: %s \n", grupo1.literátura.titulo);
printf("Nº do registro: %3i \n", grupo1.literatura.regnum);
}// main
```

Vamos analisar alguns detalhes deste programa:

- Primeiro, declaramos uma variável estrutura grupo1, do tipo grupo, e inicializamos esta estrutura com os valores mostrados. Quando uma matriz de várias dimensões é inicializada, usamos chaves dentro de chaves, do mesmo modo inicializamos estruturas dentro de estruturas.
- Segundo, note como acessamos um elemento da estrutura que é parte de outra estrutura. O operador ponto é usado duas vezes:

grupo1.dicionario.titulo

Isto referencia o elemento **titulo** da estrutura **dicionario** da estrutura **grupo1**.
Logicamente este processo não pára neste nível, podemos ter uma estrutura dentro de outra dentro de outra.
Tais construções aumentam o nome da variável, podendo descrever surpreendentemente o seu conteúdo.

7.13 - Matrizes de Estruturas

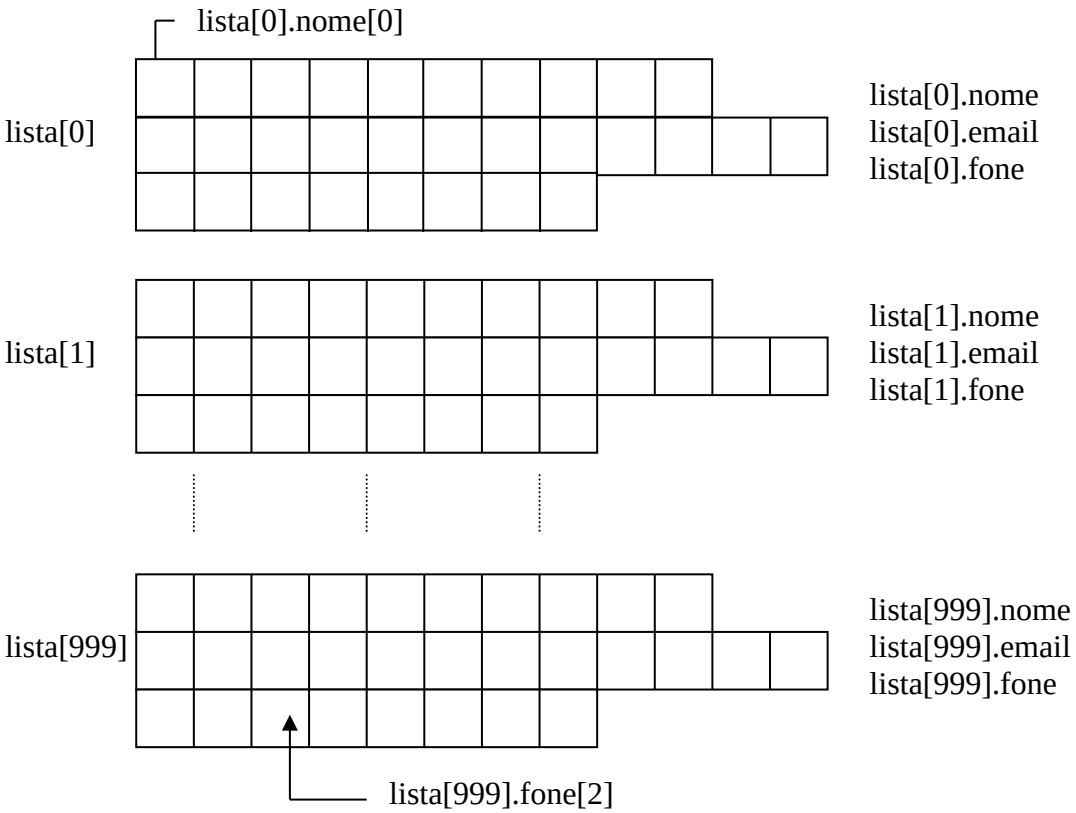
Por exemplo, em um programa que mantenha uma lista de emails e números de telefones, você poderia definir uma estrutura que armazenasse um nome, um email e um número:

```
struct entrada{
        char nome[10];
        char email[12];
        char fone[8];
        };
```

Porém, uma lista telefônica é composta de vários nomes, emails e números. Então, precisamos de uma matriz de estruturas. Depois que a estrutura estiver definida, podemos declarar uma matriz da seguinte maneira:

```
struct entrada          lista[1000];
```

Esta instrução declara uma matriz chamada lista que contém 1000 elementos. Cada elemento é uma estrutura do tipo entrada e é identificada por um índice, como em qualquer outra matriz. Cada uma dessas estruturas, por sua vez, contém três elementos, cada um dos quais é uma matriz do tipo char. Esse esquema relativamente complexo é ilustrado abaixo:



Da mesma forma que podemos atribuir o valor de uma variável estrutura a outra

```
lista[5]  =  lista[1]
```

Podemos transferir dados entre os elementos individuais das matrizes que são membros das estruturas:

```
lista[5].fone[1]  =  lista[2].fone[3];
```

Esta instrução move o segundo caractere do número de telefone armazenado em lista[5] para a quarta posição do número de telefone armazenado em lista[2]. (Não se esqueça que os índices começam em 0).

7.14 - Inicializando Estruturas Complexas

No caso de uma estrutura que contenha estruturas como membros, os valores de inicialização devem ser listados em ordem.

```
Ex.:    struct cliente {
                char firma[20];
                char contato[25];
                };
        struct venda {
                struct cliente   comprador;
                char item[20];
                float quantia;
                } minhavenda   =   { { "Acme Industrias", "George Adams"},
                                     "meias",
                                     1000   };
```

Estas instruções realizam as seguintes inicializações:

- o membro da estrutura **minhavenda.comprador.firma** é inicializado com o string "Acme Industrias".
- o membro da estrutura **minhavenda.comprador.contato** é inicializado com o string "George Adams".
- o membro da estrutura **minhavenda.item** é inicializado com o string "meias".
- o membro da estrutura **minhavenda.quantia** é inicializado com o valor 1000.

No caso de matrizes de estruturas. Os dados de inicialização serão aplicados, pela ordem, às estruturas da matriz.

Ex. - declaramos uma matriz de estruturas do tipo venda e inicializamos os dois primeiros elementos da matriz (ou seja, as duas primeiras estruturas):

```
struct cliente {
        char firma[20];
        char contato[25];
        };

struct venda {
        struct cliente   comprador;
        char item[20];
        float quantia;
        };

struct venda    A1990[100] = {
        { { "Acme Indústrias", "George Adams" },
          "meias",
           1000        }
        { { "Wilson & Cia", "Edi Wilson" },
           "brincos",
           290         }         };
```

- o membro da estrutura **A1990[0].comprador.firma** será inicializado com o string "Acme Industrias".
- o membro da estrutura **A1990[0].comprador.contato** será inicializado com o string "George Adams".
- o membro da estrutura **A1990[0].item** será inicializado com o string "meias".
- o membro da estrutura **A1990[0].quantia** será inicializado com valor 1000.

- o membro da estrutura **A1990[1].comprador.firma** será inicializado com o string "Wilson & Cia".
- o membro da estrutura **A1990[1].comprador.contato** será inicializado com o string "Edi Wilson".
- o membro da estrutura **A1990[1].item** será inicializado com o string "brincos".
- o membro da estrutura **A1990[1].quantia** será inicializado com valor 290.

7.15 – Exercícios Propostos

1. Fazer um programa para **Sistema de Conta Bancária** – este programa se destina a controlar as contas de 10 clientes.

   [1] Cadastro - receber os valores digitados pelo usuário. Apenas um registro é cadastrado por vez.
   [2] Depósito - o acesso deve ser feito através do nº de conta corrente. Buscar o registro, mostrar o nome do cliente e o saldo para simples conferência, pedir o valor do depósito, fazer as alterações e apresentar na tela o saldo atualizado.
   [3] Retirada - o acesso deve ser feito através do nº de conta corrente. Buscar o registro, mostrar o nome do cliente e o saldo para simples conferência, pedir o valor da retirada, fazer as alterações se possível (a retirada só será permitida, se houver saldo suficiente) e apresentar na tela o saldo atualizado.

```
struct cliente{
    char    nome[30];
    int     conta;
    float   saldo;
};
```

2. Fazer um programa para **Diário Eletrônico** – este programa se destina a controlar as notas e a média de 10 alunos.

   [1] Cadastro - receber os valores digitados pelo usuário, inicialmente notas e média=0. Apenas um registro é cadastrado por vez.
   [2] Controle de Notas - o acesso deve ser feito através do RA. Buscar o registro, mostrar o nome do aluno para simples conferência, fazer as alterações das notas, calcular a média e apresentar na tela as notas e a média.

```
struct aluno{
char        nome[80];
char        RA[7];
float       nota[2];        //notas de provas – considerar 2 provas
float       media;          //média aritmética das provas
};
```

8. ESTRUTURAS COMPLEXAS

8.1 - Estruturas e Ponteiros

Podemos declarar ponteiros que apontem para estruturas e também usar ponteiros como membros de estruturas, ou ambos.

8.2 - Ponteiros para Estruturas

Os ponteiros para estruturas geralmente são usados para passar uma estrutura como argumento para uma função. Além disso, os ponteiros para estruturas também são usados em um método muito poderoso de armazenagem de dados conhecido como listas encadeadas.

Como um programa pode criar e usar ponteiros para estruturas? Em primeiro lugar, devemos definir uma estrutura.

Ex.:

```
struct peça {
        int     numero;
        char    nome[10];
};
```

A seguir, devemos declarar um ponteiro para o tipo peca.

```
struct peça     *p_peça;
```

O operador de indireção (*) na declaração significa que *p_peça* é um ponteiro para o tipo *peça*, não uma instância da estrutura do tipo peça.

O ponteiro já pode ser inicializado? Não, porque a estrutura peça foi definida, mas nenhuma instância dessa estrutura foi declarada ainda. Lembre-se que é a declaração, não a definição, que reserva espaço na memória para a armazenagem de um objeto de dados. Como o ponteiro precisa de um endereço na memória para o qual possa apontar, teremos que declarar uma instância do tipo peça antes que algo possa apontar para ela.

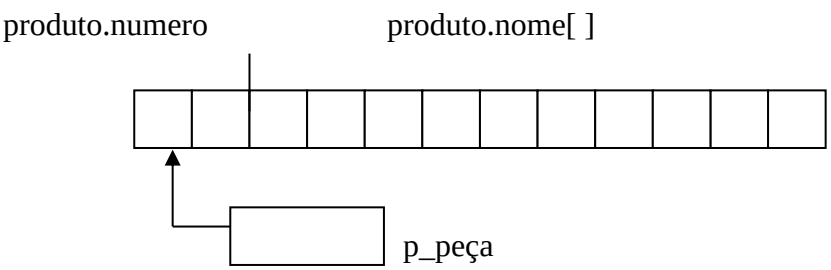
Portanto, esta é a declaração:

```
struct peça     produto;
```

Agora já podemos inicializar o ponteiro:

```
p_peça  =  &produto;
```

Essa instrução atribui o endereço de produto a p_peça. A relação entre uma estrutura e um ponteiro para essa estrutura é mostrada na figura abaixo:



Um ponteiro para uma estrutura aponta para o primeiro byte dessa estrutura.

Agora que já temos um ponteiro apontando para a estrutura produto, podemos acessar os membros de uma estrutura usando um ponteiro de 3 maneiras:

8.2.1 - Operador de Acesso Indireto

É o mais comum. Que consiste dos símbolos -> (um sinal de menos seguido do símbolo de "maior que" - quando estes dois caracteres são usados juntos desta maneira, a linguagem C os interpreta como um único operador, não dois). O símbolo é colocado entre o nome do ponteiro e o nome do membro da estrutura.
Ex.:

p_peça->numero /* acessa o membro número da estrutura produto */

8.2.2 - Operador de Indireção (*)

Ex.:
p_peça é um ponteiro para a estrutura peça; portanto,
*p_peça refere-se a produto

Devemos aplicar o operador ponto (.) para acessarmos elementos individuais de produto.

(*p_peça).numero = 100; /* atribui o valor 100 a produto.numero */

O item *p_peça* deve estar entre parênteses porque o operador (.) tem maior precedência que o operador (*).

8.2.3 - Nome da Estrutura

Ex.:
produto.numero

Portanto, sendo p_peça um ponteiro para a estrutura peça, todas estas expressões são equivalentes:
produto.numero
(*p_peça).numero
p_peça->numero

8.3 - Ponteiros e Matrizes de Estruturas

Tanto as matrizes de estruturas como os ponteiros para estruturas são ferramentas de programação extremamente poderosas. Podemos também combinar esses dois métodos usando ponteiros para acessar estruturas que são elementos de matrizes.

Para ilustrar isso, vamos usar esta definição de estrutura de um exemplo anterior:

```
struct peça {
        int     numero;
        char    nome[10];
        };
```

Depois que a estrutura foi definida, podemos definir uma matriz do tipo peça, chamada dados:

```
struct peça     dados[100];
```

A seguir, podemos declarar um ponteiro para o tipo peça e inicializá-lo para apontar para a primeira estrutura contida na matriz:

```
struct peça     *p_peça;
p_peça  =  &dados[0];
```

Lembrando que o nome de uma matriz sem os colchetes é um ponteiro para o primeiro elemento dessa matriz. Portanto, a segunda linha também poderia ter sido escrita da seguinte forma:

```
p_peça  =  dados;
```

Agora já temos uma matriz de estruturas do tipo peça e um ponteiro para o primeiro elemento dessa matriz (ou seja, para a primeira estrutura da matriz). Poderíamos, por exemplo, imprimir o conteúdo desse primeiro elemento usando a instrução:

```
printf("%i      %s", p_peça->numero, p_peça->nome);
scanf("%i",     &(p_peça->numero));
```

E se quiséssemos imprimir todos os elementos da matriz? Nesse caso, provavelmente usaríamos um loop for. Para acessar os membros usando notação de ponteiros, teríamos que alterar o ponteiro p_peça para que, a cada iteração do loop, ele apontasse para o próximo elemento da matriz (ou seja, para a próxima estrutura contida na matriz).

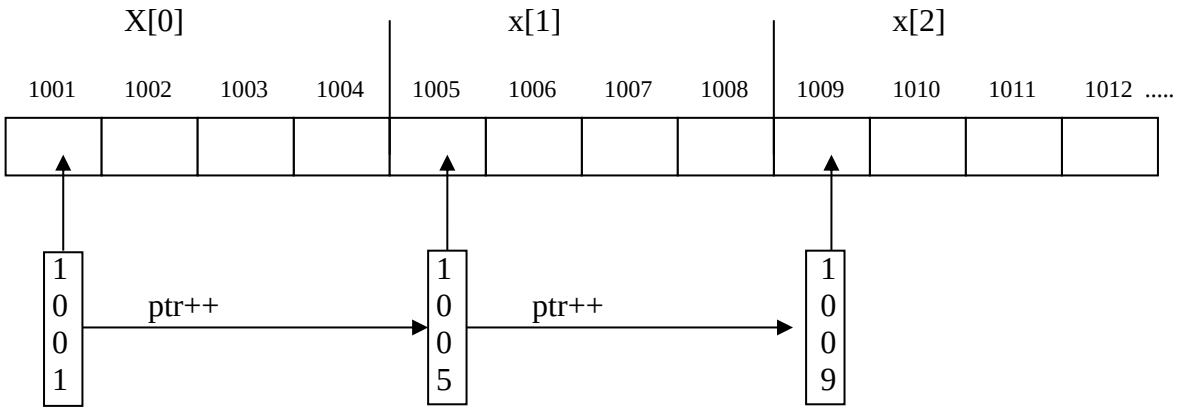
Para isso, a aritmética de ponteiros da linguagem C pode lhe ser útil. O operador unário de incremento (++) tem um significado especial quando é aplicado a um ponteiro, passando a significar "incrementar o ponteiro com um valor equivalente ao tamanho do objeto para o qual ele está apontando". Em outras palavras, se você tem um ponteiro ptr que aponta para um objeto de dados do tipo obj, a instrução:

```
ptr++;
```

tem o mesmo efeito de:

```
ptr  +=  sizeof (obj);
```

Os elementos de uma matriz sempre são armazenados sequencialmente na memória. Se um ponteiro estiver apontando para o elemento *n* de uma matriz, o uso do operador (++) para incrementá-lo fará com que ele passe a apontar para o elemento *n+1*. Isto é ilustrado na figura abaixo, que mostra uma matriz chamada x[ ] que consiste de elementos de quatro bytes (cada elemento pode ser, por exemplo, uma estrutura contendo quatro membros do tipo char, cada um dos quais ocupa um byte). O ponteiro **ptr** foi inicializado para apontar para x[0]; cada vez que é incrementado, **ptr** passa a apontar para o próximo elemento da matriz.



Isto significa que o seu programa pode avançar através de uma matriz de estruturas (ou, para ser exato, através de uma matriz de qualquer tipo) simplesmente incrementando um ponteiro. Este tipo de notação geralmente é mais fácil de usar e mais conciso do que o uso de subscritos de matrizes para realizar a mesma tarefa.

Ex. - acessa elementos sucessivos de uma matriz através do incremento de um ponteiro.

```
#include <stdio.h>
#define     MAX         4

struct peça {
          int numero;
          char nome[10];
          };
main()
{
struct peça       *p_peça,        dados[MAX] = {1, "camisa",
                                                2, "calça",
                                                3, "gravata",
                                                4, "casaco"      };
int       contagem;
p_peça  =  dados;                 // inicializa o ponteiro para o primeiro elemento da matriz

for (contagem = 0; contagem < MAX; contagem++)
   {
   printf("\n No endereço %u :          %i       %s", p_peça, p_peca->numero, p_peça->nome) ;
   p_peça++;
   }
}// main
```

Teremos como saída, por exemplo:

```
No endereço   96: 1    camisa
No endereço 120: 2    calça
No endereço 144: 3    gravata
No endereço 168: 4    casaco
```

Inicialmente, o programa declara e inicializa uma matriz de estruturas chamada dados. A seguir, é definido um ponteiro chamado p_peça para apontar para uma estrutura do tipo peças. A primeira tarefa da função main( ) é inicializar o ponteiro p_peça para que ele aponte para a matriz dados declarada anteriormente. A seguir, todos os elementos dessa matriz são impressos na tela usando um loop for que incrementa o ponteiro a cada interação. O programa exibe também o endereço de cada elemento da matriz.

Observando os endereços mostrados, percebemos que todos os incrementos serão feitos em quantidades iguais, correspondendo ao tamanho da estrutura peças (na maioria dos sistemas, esse incremento será de 24 bytes). Isto ilustra claramente o fato de que o incremento de um ponteiro é feito com um valor igual ao tamanho do objeto de dados para o qual ele está apontando.

8.4 - Passando Estruturas como Argumentos para Funções

Como qualquer outro tipo de dados, uma estrutura pode ser passada como um argumento para uma função. O programa abaixo mostra como isso é feito.

Ex.:

```
#include <stdio.h>

struct dados{
        float    quantia;
        char     nome[80];
        } ;
void    print_reg (struct dados          x);

main( )
{
struct dados    reg;
printf("Digite o nome do doador: ");
scanf("%s", reg.nome);
printf("\n Digite a quantia doada: ");
scanf("%f", &reg.quantia);
print_reg( reg );
}// main

void    print_reg(struct dados          x)
{
printf("\O doador %s   doou R$%.2f.", x.nome, x.quantia);
}
```

Teremos como saída na tela:

Digite o nome do doador: Carlos Silva
Digite a quantia doada: 1000.00
O doador Carlos Silva doou R$1000.00.

O protótipo da função deverá receber a estrutura, portanto devemos incluir os parâmetros adequados. No caso, é uma estrutura do tipo dados. As mesmas informações são repetidas no cabeçalho da função. Ao chamarmos a função, temos que informar o nome da instância dessa estrutura - no caso, **reg**. Isso é tudo. Passar uma estrutura para uma função não é muito diferente de passar uma variável simples. Alternativamente, podemos passar uma estrutura para uma função informando o seu endereço (ou seja, passando um ponteiro que aponte para a estrutura).

De fato, esta era a única maneira de usarmos uma estrutura como argumento nas versões mais antigas da linguagem C. Isso já não é necessário, mas é possível encontrarmos programas mais antigos que ainda usam esse método. Ao passarmos um ponteiro como argumento para uma função, teremos que usar o operador de acesso indireto (->) para acessarmos membros da estrutura de dentro da função.

Ex. – Novamente, usaremos o programa de livros, onde uma função obterá informações dos livros pelo usuário e outra irá imprimi-las.

```
#include <stdio.h>
#include <stdlib.h>
typedef struct 1ivro {
        char titulo[30];
        int regnum;
        }livro;
livro   novonome();             // função do tipo struct livro chamada novonome
void    listar (livro    liv);  // função void, cujo parâmetro é do tipo struct livro
main( )
{
livro   livro1, livro2;         // variável estrutura chamada livro1, livro2
livro1 = novonome();            // livro1 irá receber o retorno da função novonome
livro2 = novonome();            // livro2 irá receber o retorno da função novonome
listar(livro1);
listar(livro2);
} // main

livro   novonome()
{
char numstr[8];
livro   livr;                   // variável estrutura chamada livr
printf("\n Novo livro \n Digite titulo: ");
gets(livr.titulo);
printf("Digite o numero do registro (3 dígitos): ”);
gets(numstr);
livr.regnum = atoi(numstr);
return(livr);
}// novonome
```

```
void listar (livro          liv)
{
printf("\n Livro: \n");
printf("Titulo: %s ", liv.titulo);
printf("\n Nº do registro: %3i", liv.regnum);
} // listar
```

Visto que as duas funções, como também o programa main(), devem conhecer a estrutura livro, ela deve ser definida antes da main().

As funções main(), novonome() e listar() declaram internamente suas próprias variáveis estrutura, chamadas livro1 e livro2, livr e liv.

A função listar() recebe uma cópia da estrutura e a coloca num endereço conhecido somente por ela; **não** é a mesma estrutura declarada em main().

A função novonome() é chamada pelo programa principal para obter informações do usuário sobre os 2 livros. Esta função guarda as informações em uma variável interna, livr, e retorna o valor desta variável para o programa principal usando o comando **return**, exatamente como faria para devolver uma simples variável. A função novonome( ) deve ser declarada como sendo do tipo struct livro, visto que ela retorna um valor deste tipo.

O programa principal atribui o valor retornado por novonome() à variável estrutura livro1 e livro2. Finalmente main() chama a função listar() para imprimir os valores de livro1 e livro2, passando os valores destas duas estruturas para a função como variáveis.

A função listar() atribui estes valores à variável estrutura interna liv e acessa os elementos individuais desta estrutura para imprimir seus valores.

8.5 – Estruturas e Alocação Dinâmica

Para alocarmos memória dinamicamente na **main**, devemos declarar um ponteiro e utilizar malloc(), calloc() ou realloc(). Porém, quando utilizamos **função**, devemos lembrar que a memória não pode ser "conhecida" apenas na função, mas também na main. Portanto, devemos declarar um ponteiro na main e a função pode ser feita de 2 formas:

- **Chamada por Referência** – devemos passar o endereço do ponteiro declarado na main, portanto, a função receberá como parâmetro ponteiro para ponteiro.

- **Chamada por Valor** – só conseguimos trabalhar dessa forma, se a função **retornar o endereço** alocado para o ponteiro declarado na main, permitindo assim, o acesso deste à memória alocada. Caso contrário o ponteiro na main continuará NULL e ao sairmos da função perdemos a referência da memória alocada. Lembre-se que as variáveis (inclusive ponteiro) declaradas dentro da função, deixam de existir assim que a função termina.

Versão para Alocação feita na main( )

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

typedef struct 1ivro {
        char titulo[30];
        int regnum;
        }livro;

main( )
{
livro    *ptr=NULL;
if((ptr = (livro *) realloc(ptr, 10 * sizeof(livro))) == NULL)       //aloca 10 elementos
  {
   printf("Erro na alocação);
   exit(1);
  }
}//main
```

Versão utilizando Chamada por Referência

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

typedef struct livro {
        char titulo[30];
        int regnum;
        }livro;

void aloca(livro **p, int tam);                  //passa o endereço do ponteiro declarado na main

main( )
{
livro    *ptr=NULL;
aloca(&ptr, 10);                                 //chamada por referencia
}//main

void aloca(livro **p, int tam)
{
if((*p=(livro*)realloc(*p, tam*sizeof(livro)))== NULL)
  {
   printf("Erro de alocacao");
   exit(1);
  }
}//aloca
```

Versão utilizando Chamada por Valor – com RETORNO do endereço alocado

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

typedef struct livro {
        char titulo[30];
        int regnum;
        }livro;

livro*   aloca(livro *p, int tam);              //retorna o endereço da memória alocada

main()
{
livro    *ptr=NULL;
ptr = aloca(ptr, 10);                          //chamada por valor
}//main

livro*   aloca(livro *p, int tam)
{
if((p=(livro*)realloc(p, tam*sizeof(livro)))== NULL)
  {
   printf("Erro de alocacao");
   exit(1);
  }
return p;
}//aloca
```

8.8 – Exercícios Propostos

1. Dado a estrutura abaixo, implemente uma rotina de cadastro, deve-se consultar o usuário para continuar. O registro deve ser gerado automaticamente pelo sistema. **Utilizar alocação dinâmica e ponteiros para a estrutura.**

```
struct agenda{
        int reg;
        char nome[80];
        float nota;
        };
```

2. Fazer um programa para **Sistema de Conta Bancária** – este programa se destina a controlar as contas de clientes. Consultar o usuário para continuar. **Utilizar alocação dinâmica e ponteiros para a estrutura.**

[1] Cadastro - receber os valores digitados pelo usuário. Apenas um registro é cadastrado por vez.

[2] Depósito - o acesso deve ser feito através do nº de conta corrente. Buscar o registro, mostrar o nome do cliente e o saldo para simples conferência, pedir o valor do depósito, fazer as alterações e apresentar na tela o saldo atualizado.

[3] Retirada - o acesso deve ser feito através do nº de conta corrente. Buscar o registro, mostrar o nome do cliente e o saldo para simples conferência, pedir o valor da retirada, fazer as alterações se possível (a retirada só será permitida, se houver saldo suficiente) e apresentar na tela o saldo atualizado.

```
struct cliente{
        char nome[30];
        int conta;
        float saldo;
        };
```

3. Fazer um programa para **Diário Eletrônico** – este programa se destina a controlar as notas e a média dos alunos. Consultar o usuário para continuar. **Utilizar alocação dinâmica e ponteiros para a estrutura.**

[1] Cadastro - receber os valores digitados pelo usuário, inicialmente notas e média=0. Apenas um registro é cadastrado por vez.

[2] Controle de Notas - o acesso deve ser feito através do RA. Buscar o registro, mostrar o nome do aluno para simples conferência, fazer as alterações das notas, calcular a média e apresentar na tela as notas e a média.

```
struct aluno{
char nome[80];
char RA[7];
float nota[2];          //notas de provas – considerar 2 provas
float media;            //média aritmética das provas
};
```

4. Fazer um programa para **Controle de Hotel** - este programa se destina a controlar o check-in (cadastro de hóspedes) de um hotel. O hotel possui 15 quartos. **Utilizar alocação dinâmica e ponteiros para a estrutura.**

[1] Check-in - alocar dinamicamente espaço, receber os valores digitados pelo usuário, se o hóspede não tiver acompanhantes atribuir categoria **S**olteiro, caso contrário **F**amiliar, buscar o número do quarto disponível, de acordo com a categoria na estrutura **quartos.** Apenas um hóspede é cadastrado por vez. Não esquecer de atualizar o quarto da estrutura **quartos** para **O**cupado.

[2] Check-out – encerra a estadia e apresenta o relatório, de acordo com o quarto. Apenas um registro é acessado por vez, buscar e mostrar o número do quarto, o nome do hóspede, quantidade de acompanhantes, a categoria (Solteiro ou Familiar,o tempo de permanência em dias e o valor a ser pago.

[3] Fim

Dica:

- No check-in - não esquecer de verificar se na estrutura **hospede** há um espaço vago (cujo quarto = -1), se houver o novo hóspede deverá ser ali armazenado, caso contrário, acrescentar no final da estrutura.

```
struct hospede{
int quarto;                 // número do quarto
char nome[80];
int acompanhante;           // quantidade de acompanhantes
char categoria;             // [S]olteiro / [F]amiliar
int dias;                   // tempo de permanência – em dias
};

struct quarto{
int num;                    // número do quarto
char categoria              // [S]olteiro / [F]amiliar
char status                 // [L]ivre / [O]cupado
};
```

Categoria de quarto:
[**S**]olteiro – diária R$ 85,00 por pessoa
[**F**]amiliar – diária R$ 45,00 por pessoa

9. ARQUIVOS EM I/O

9.1 - Introdução

Operações em discos são executadas em entidades chamadas "**arquivos**". Arquivo é uma coleção de bytes referenciados por um nome único.

A linguagem C divide as categorias de acesso a disco em 2 grupos: alto-nível e baixo-nível.

9.2 – Alto-Nível

Também chamada de **bufferizada**, permite acessar arquivos de 4 modos diferentes:

- Os dados são lidos e escritos **um caracter por vez** – semelhante ao getch() e putch()
- Os dados são lidos e escritos como **strings** – semelhante ao gets() e puts()
- Os dados são lidos e escritos de **modo formatado** – semelhante ao scanf() e printf()
- Os dados são lidos e escritos como **registro** ou **bloco** – os dados têm tamanho fixo e é utilizado para armazenar matrizes e/ou estruturas

9.3 – Baixo-Nível

Também chamada de **não bufferizada**, permite acessar arquivos de um único modo:

- Os dados são lidos e escritos através de **um buffer cheio de dados.** E o programador é quem deverá criar e manter o buffer, pois as funções não fazem esta manutenção.

Este modo é usado para criar as funções em alto-nível e não é portável, ou seja, não é reconhecido pelo padrão ANSI.

9.4 – Texto X Binário

É a forma como os arquivos são abertos e, constitui outra maneira de classificar o acesso à arquivos.

- **Modo Texto** – imita arquivos UNIX - o arquivo aberto deste modo é interpretado como sequências de caracteres agrupados em linhas, e é reconhecida a indicação de nova linha e de fim de arquivo. Neste modo os números são guardados como cadeias de caracteres.

- **Modo Binário** – imita arquivos MS-DOS - é mais simples que o modo texto, não há conversão, ou seja, qualquer caracter é lido ou gravado sem alteração e não temos indicação de fim de arquivo. Neste modo os números são guardados como estão na memória (respeitando o tipo básico e a quantidade de bytes).

9.5 – Operações com Arquivos em Disco – Alto-Nível

9.5.1 – Ponteiro para arquivo

Para trabalharmos com arquivos, precisamos declarar um ponteiro para o tipo FILE (arquivo).
Arquivo de cabeçalho:stdio.h
sintaxe:

```
FILE    *fptr ;
```

onde:
FILE – estrutura que contém informações específicas sobre o arquivo (tamanho, leitura ou escrita)
fptr – corresponde ao nome de um ponteiro para arquivo (FILE)

9.5.2 – Função fopen( )

Esta função é utilizada para criar (abrir) um arquivo e, retorna um ponteiro para o tipo FILE.
Arquivo de cabeçalho: stdio.h
sintaxe:

```
fptr = fopen ("xxxxxxxx.xxx", "tipo_abertura");
```

onde:
fptr – corresponde a um ponteiro para arquivo (FILE) a ser aberto ou criado
xxxxxxxx.xxx – nome e extensão do arquivo a ser aberto
tipo_abertura – contém o tipo de abertura do arquivo:

"r" – leitura **"w" – escrita** **"a"- adicionar dados**
além de 2 modificadores: **"+" – atualização** **"b"- modo binário**

Tipo	Ação
"rb"	Abre um arquivo binário para leitura – o arquivo deve existir
"wb"	Cria um arquivo binário para gravação – se o arquivo existir será destruído e reinicializado – se o arquivo não existir será criado
"ab"	Cria um arquivo binário para gravação – se o arquivo existir, os dados serão adicionados no fim – se o arquivo não existir será criado
"rb+"	Abre um arquivo binário para leitura e gravação – o arquivo deve existir, sendo atualizado
"wb+"	Cria um arquivo binário para leitura e gravação – se o arquivo existir será destruído e reinicializado – se o arquivo não existir será criado
"ab+"	Cria um arquivo binário para atualizações – se o arquivo existir, os dados serão adicionados no fim – se o arquivo não existir será criado

Cuidado: o fopen( ) dá erro:

- se tentarmos abrir um arquivo inexistente com o tipo **"rb"**ou **"rb+"**
- se o diretório for inexistente ou inválido
- se nome do arquivo inválido

Portanto, deve-se sempre fazer um teste, e caso o arquivo não possa ser aberto, a função fopen() devolve 0 (NULL).

9.5.3 – Função fclose( )

Esta função é utilizada para fechar um arquivo.

Arquivo de cabeçalho: stdio.h
sintaxe:

```
fclose (fptr);
```

onde:
fptr – corresponde ao ponteiro do arquivo a ser fechado

9.5.4 – Leitura e Gravação – Registros

Lê e grava os dados complexos como matrizes e estruturas, porém no **modo binário**.

Arquivo de cabeçalho: stdio.h

Leitura do arquivo – recebe informações do arquivo
sintaxe:

```
fread (p, tam, qtde, fptr);          // lê dados do arquivo
```

onde:
p – ponteiro do tipo void para o endereço da memória onde serão armazenados os dados lidos
tam – tamanho em bytes do tipo de dados a ser lido
qtde – quantidade de itens a serem lidos
fptr – corresponde ao ponteiro do arquivo a ser lido

Escrita no arquivo – grava informações no arquivo
sintaxe:

```
fwrite (p, tam, qtde, fptr);          // grava dados no arquivo
```

onde:
p – ponteiro do tipo void para o endereço da memória do dado a ser armazenado
tam – tamanho em bytes do tipo de dados a ser armazenado
qtde – quantidade de itens a serem gravados
fptr – corresponde ao ponteiro do arquivo a ser gravado

9.5.5 – Exemplos

Exemplo 1 Gravação no modo Registro – Matriz

```
#include <stdio.h>
main()
{
int       mat[10],arq[10],i;
FILE *fptr=NULL;                                    //declara ponteiro para arquivo

for(i=0;i<10;i++)
    {
    printf("\nmat[%i] = ",i);
    scanf("%i", mat+i);                             // armazena na memória RAM
    }

// rotina para gravar
if ((fptr = fopen("arqbin.bin", "wb")) == NULL)
    printf("Erro ao abrir o arquivo");
else
    fwrite(mat,sizeof(int),10,fptr);            // armazena no arquivo – todos de 1 vez
fclose (fptr);

// rotina para ler
if ((fptr = fopen("arqbin.bin", "rb")) == NULL)
    printf("Erro ao abrir o arquivo");
else
   {
    fread(arq,sizeof(int),10,fptr);             // retira do arquivo – todos de 1 vez
    fclose(fptr);
   }

//mostra que leu e armazenou corretamente
for(i=0;i<10;i++)
    printf("\n mat[%i] = %i\tarq[%i] = %i",i,*(mat+i),i,*(arq+i));
}//main
```

Exemplo 2 Gravação no modo Registro – Estrutura

```
#include <stdio.h>

typedef       struct dados{
     char     produto[30];
     int      num;
     float    preco;
     } dados;
```

```
main()
{
dados  p1,p2;
FILE *fptr=NULL;                          //declara ponteiro para arquivo

printf("\n Digite produto, registro e preco:");
scanf("%s      %i      %f", p1.produto, &(p1.num), &(p1.preco));    // armazena na RAM

// rotina para gravar
if ((fptr = fopen("arqbin.bin", "wb")) == NULL)
        printf("Erro ao abrir o arquivo");
else
        fwrite(&p1, sizeof (dados), 1, fptr);          // armazena no arquivo
fclose (fptr);

// rotina para ler
if ((fptr = fopen("arqbin.bin", "rb")) == NULL)
        printf("Erro ao abrir o arquivo");
else
        {
        fread (&p2, sizeof (dados), 1, fptr);          // retira do arquivo
        fclose(fptr);
        }

//mostra que leu e armazenou corretamente
printf("\n %s = %s", p1.produto, p2.produto);
printf("\n %i = %i", p1.num, p2.num);
printf("\n %f = %f", p1.preco, p2.preco);
}//main
```

9.6 – Acesso Aleatório – Alto-Nível

9.6.1 – Ponteiro para arquivo

Um ponteiro para o tipo FILE (arquivo) aponta para um byte particular chamado **posição atual.** Cada vez que lemos ou gravamos qualquer coisa no arquivo, o ponteiro é movido para o fim dessa informação, portanto a leitura ou armazenamento de novo dado, começará deste ponto.

QUANDO O ARQUIVO É ABERTO, O PONTEIRO É FIXADO NO INÍCIO DO MESMO. APENAS SE ABRIRMOS COM A OPÇÃO "A" (APPEND), É QUE ELE SERÁ POSICIONADO NO FIM DO ARQUIVO.

9.6.2 – Função fseek( )

Permite a movimentação do ponteiro para arquivo (posição atual).

Arquivo de cabeçalho: stdio.h

sintaxe:
fseek (fptr, offset, modo);

onde:
fptr – corresponde ao ponteiro para arquivo
offset – consiste no número de bytes de deslocamento, a partir do "modo" – deve ser do tipo **long int**
modo – especifica a posição desejada

Modo	Ação
0 ou SEEK_SET	Começo do arquivo
1 ou SEEK_CUR	Posição corrente do ponteiro
2 ou SEEK_END	Fim do arquivo

9.6.3 – Função ftell( )

Retorna a posição do ponteiro de um arquivo **binário** em relação ao seu começo.

Arquivo de cabeçalho: stdio.h

sintaxe:
pos = ftell (fptr);

onde:
pos – número de bytes do começo do arquivo até a posição atual – deve ser do tipo **long**
fptr – corresponde ao ponteiro para arquivo

9.6.4 – Função rewind( )

Reposiciona o ponteiro de um arquivo no início.

Arquivo de cabeçalho: stdio.h

sintaxe:
void rewind (fptr);

onde:
fptr – corresponde ao ponteiro para arquivo

9.6.5 – Exemplos de Acesso Aleatório

Exemplo1: Gravação no modo Registro - Estrutura

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

typedef  struct dados{
      char        produto[30];
      int         qtde;
      float       preco;
      }dados;

main()
{
dados *p=NULL;                    //ponteiro para estrutura
FILE *fptr=NULL;                  //ponteiro para arquivo
int i, total=0;
char op;
long int pos;
if ((p = (dados*) realloc(p, 1 * sizeof(dados))) == NULL)      //aloca 1 estrutura na memória
   {
    printf("\n Erro impossível alocar memória");
    exit(1);
    }
do{
   printf("\nDigite produto, quantidade e preco:");
   scanf("%s %i %f", p->produto, &(p->qtde), &(p->preco));          // armazena na memória RAM
   fflush(stdin);
// rotina para gravar
if ((fptr = fopen("arqbin.bin", "ab")) == NULL)
          printf("Erro ao abrir o arquivo");
else
  {
   fwrite(p,sizeof(dados),1,fptr);                                  //armazena no arquivo
   fclose(fptr);
   }
// rotina para ler
if ((fptr = fopen("arqbin.bin", "rb")) == NULL)
          printf("Erro ao abrir o arquivo");
else
  {
  fseek(fptr,0,2); //fim do arquivo posição
  pos = ftell(fptr); //qtde de bytes
  total = pos/sizeof(dados);
  for(i=0; i<total; i++)
     {
     fseek(fptr, i*sizeof(dados),0);          //posiciona a partir do início em cada elemento
     fread(p, sizeof(dados), 1, fptr);        //lê 1 de cada vez, pois só tenho 1 espaço alocado
```

```
    printf("\n%s %i %f",p->produto, p->qtde, p->preco);      //mostra o que leu do arquivo
    }//for
  fclose(fptr);
  }//else
  printf("\n\nDeseja continuar S/N: ");
  scanf("%c",&op);
  fflush(stdin);
  }while(op!='n' && op!='N');
system("pause");
}//main
```

9.6.6 – Exercícios

1. Dado a estrutura abaixo, implementar uma rotina de cadastro com alocação dinâmica, deve-se consultar o usuário para continuar. Caso não deseje mais cadastrar, salve todos os itens no arquivo (modo binário). O registro deve ser gerado automaticamente pelo sistema. Não esquecer de conferir se já existem elementos armazenados em arquivo, se sim, calcular o número do próximo registro.

```
struct agenda{
        int     reg;
        char    nome[80];
        float   nota;
        };
```

2. Idem ao anterior, porém o registro deve ser gravado cada vez que é criado, ou seja, a cada item cadastrado, deve-se salvar em arquivo. Alocar apenas 1 único espaço de memória.

3. Fazer um programa que aloca espaço para todos os itens gravados no arquivo binário anterior, lê todos na memória e busca qual será alterado (registro deve ser escolhido pelo usuário). Após a alteração grava os itens novamente. **DICA:** utilizar o modo de abertura do arquivo: **rb+** ou **wb**

4. Idem ao anterior, porém, o programa deve alocar espaço para apenas 1 item. Deve-se buscar o registro a ser alterado (que deve ser escolhido pelo usuário). Após a alteração gravar apenas o registro alterado. **DICA:** utilizar o modo de abertura do arquivo: apenas **rb+**

5. Refazer o programa para **Sistema de Conta Bancária**, com gravação em arquivo.

6. Refazer o programa para **Diário Eletrônico**, com gravação em arquivo.

7. Refazer o programa para **Controle de Hotel,** com gravação em arquivo.