



# Linguagem C



Ponteiros

# Ponteiros

---

- ▶ Ponteiros são a ferramenta mais poderosa oferecida pela linguagem C
  - ▶ São considerados pela maioria dos programadores como um dos tópicos mais difíceis nessa linguagem
  - ▶ Há dois principais motivos para isso
    1. Os conceitos embutidos em ponteiros podem ser novos para muitos programadores, visto que não são comuns em linguagens de alto nível
    2. Os símbolos usados para notação de ponteiros em C não são tão claros quanto poderiam ser
      - ▶ o mesmo símbolo é usado para duas finalidades diferentes
-

# O que são ponteiros?

---

- ▶ O nome de uma variável indica o que está armazenado nela
  - ▶ O endereço de uma variável é um *ponteiro*
  - ▶ Seu valor indica em que parte da memória do computador a variável está alocada
  - ▶ *Ponteiros* proporcionam um modo de acesso à variável sem referenciá-la diretamente (modo indireto de acesso)
-

# Por que os ponteiros são usados?

---

- ▶ Para dominar a linguagem C, é essencial dominar ponteiros
  - ▶ *Ponteiros* são utilizados em situações em que o uso do nome de uma variável não é permitido ou é indesejável
  - ▶ Podemos citar algumas razões para o uso de ponteiros
    - ▶ Fornecem maneiras com as quais as funções podem realmente modificar os argumentos que recebem (passagem por referência);
    - ▶ Passar matrizes e strings mais convenientemente de uma função para outra (usá-los no lugar de matrizes);
    - ▶ Manipular os elementos de matrizes mais facilmente por meio da movimentação de ponteiros, no lugar de índices entre colchetes;
    - ▶ Criar estruturas de dados complexas, como listas encadeadas e árvores binárias, em que um item deve conter referências a outro;
    - ▶ Alocar e desalocar memória dinamicamente do sistema;
    - ▶ Passar para uma função o endereço de outra função.
-

# Ponteiros variáveis

---

- ▶ Em C, há um tipo especial de variável, concebida para conter o endereço de outra variável, que se chama *ponteiro variável*
  - ▶ Um *ponteiro variável* armazena um endereço de memória, que é a localização de outra variável
    - ▶ Dizemos que uma variável aponta para outra variável quando a primeira contém o endereço da segunda
-

# Ponteiros constantes e o operador &

---

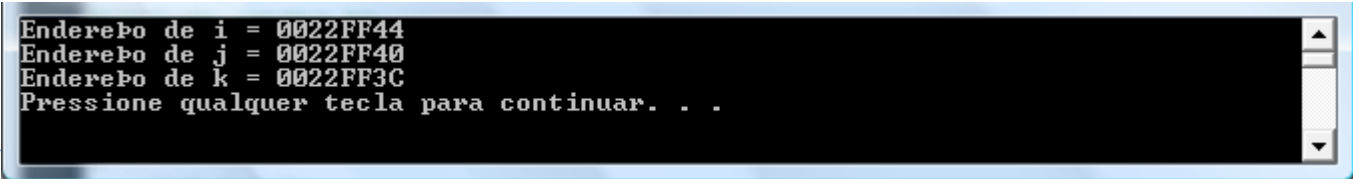
- ▶ Para conhecer o endereço ocupado por uma variável, usamos o operador de endereço **&**
- ▶ O resultado da operação é um ponteiro constante
- ▶ Eis um pequeno programa que mostra o seu uso

```
/* PonteiroCons.C */
/* Mostra ponteiros constantes */
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int i,j,k;

    printf("Endereço de i = %p\n", &i); /* %p para ponteiros */
    printf("Endereço de j = %p\n", &j);
    printf("Endereço de k = %p\n", &k);

    system("PAUSE");
    return 0;
}
```



```
Endereço de i = 0022FF44
Endereço de j = 0022FF40
Endereço de k = 0022FF3C
Pressione qualquer tecla para continuar. . .
```

# Passando argumentos por referência com ponteiros

---

- ▶ Um função pode receber diversos argumentos, mas só consegue retornar um único valor por meio do comando **return**
- ▶ Como fazer para que uma função retorne mais de um valor?
  - ▶ Usando ponteiros
- ▶ Há duas maneiras de passar argumentos para uma função
  - ▶ por valor e por referência por meio de ponteiros

# Passando argumentos por referência com ponteiros

---

- ▶ Para que uma função retorne mais de um valor para a função chamadora, devemos seguir dois passos
    1. A função chamadora passa os endereços das suas variáveis usando o operador de endereços, os quais indicam variáveis que queremos que a função chamada coloque os valores que devem ser retornados
      - ▶ Quando passamos endereços de variáveis, a função pode alterar a variável original
    2. A função chamada deverá criar variáveis para armazenar os endereços que estiver recebendo, enviados pela função chamadora
      - ▶ Essas variáveis são ponteiros variáveis
-



# Exemplo

---

```
/* Ponteiros.C */
/* Mostra o uso de ponteiros variáveis */
#include <stdio.h>
#include <stdlib.h>

void reajusta20( float *, float *); /* protótipo */

int main()
{
    float val_preco, val_reaj;
    do
    {
        printf("\nInsira o preco atual: ");
        scanf("%f", &val_preco);
        reajusta20(&val_preco, &val_reaj); /* Enviando endereços */
        printf("\nO preco novo e %.2f\n", val_preco);
        printf("O aumento foi de %.2f\n", val_reaj);
    } while( val_preco != 0.0);

    system("PAUSE");
    return 0;
}

/* reajusta20() */
/* Reajusta o preço em 20% */
void reajusta20(float *preco, float *reajuste) /* Recebendo ponteiros */
{
    *reajuste = *preco * 0.2;
    *preco *= 1.2;
}
```

---

# O operador indireto \*

---

- ▶ O operador indireto \* é unário e opera sobre um endereço ou ponteiro
- ▶ O resultado da operação é o nome da variável localizada nesse endereço
- ▶ O nome da variável representa o seu valor ou conteúdo

O operador de endereços & opera sobre o nome de uma variável e resulta o seu endereço, já o operador indireto \* opera sobre o endereço de uma variável e resulta o seu nome

---

# Permutação do valor de duas variáveis

---

- ▶ Por padrão, a linguagem C passa argumento para funções usando “*chamada por valor*”
- ▶ A função chamada não pode alterar diretamente uma variável da função chamadora
- ▶ Exemplo: uma rotina de ordenação poderia querer permutar dois elementos fora de ordem por meio de uma função chamada **troca( )**
  - ▶ Não seria suficiente escrever

`troca(a,b)`

em que a função **troca( )** é definida como

```
void troca(int x, int y) /* ERRADO */
{
    int temp
    temp = x;
    x = y;
    y = temp;
}
```

---

# Permutação do valor de duas variáveis

---

- ▶ Por causa da chamada por valor, **troca( )** não pode afetar os argumentos **x** e **y** da função que chama
- ▶ Ponteiros resolvem o problema
- ▶ O programa chamador passa os endereços dos valores a serem permutados

```
troca(&a, &b)
```

- ▶ E na função **troca( )**, os parâmetros são declarados como ponteiros, e as variáveis atuais são acessadas por meio deles

```
void troca(int *x, int *y) /* CORRETO */  
{  
    int temp  
    temp = *x;  
    *x = *y;  
    *y = temp;  
}
```

---

# Ponteiros sem funções

---

- ▶ O exemplo que apresentamos usou ponteiros como argumentos de funções
- ▶ O programa a seguir cria ponteiros como variáveis automáticas da função **main()**

```
/* PtrVar.C */
/* Mostra o uso de ponteiros declarados dentro da função main() */
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int x=4, y=7;
    int *px, *py;

    printf("&x = %p\t x = %d\n", &x , x);
    printf("&y = %p\t y = %d\n", &y , y);

    px = &x;
    py = &y;

    printf("px = %p\t*px = %d\n", px, *px);
    printf("py = %p\t*py = %d\n", py, *py);

    system("PAUSE");
    return 0;
}
```

---

# Ponteiros sem funções

---

- ▶ A instrução

```
int *px, *py
```

declara **px** e **py** como ponteiros para variáveis **int**

- ▶ Quando um ponteiro não é inicializado na instrução de sua declaração, o compilador inicializa-o com o endereço zero (NULL)
- ▶ A linguagem C garante que NULL não é um endereço válido, então, antes de usá-lo, devemos atribuir a eles algum endereço válido, isto é, feito pelas instruções

```
px = &x;  
py = &y;
```

- ▶ Um ponteiro pode ser inicializado na mesma instrução de sua declaração

```
int *px = &x, *py = &y;
```

- ▶ Observe que estamos atribuindo **&x** e **&y** a **px** e **py** respectivamente, e não a **\*px** e **\*py**
-

# Ponteiros sem funções

---

- ▶ C permite ponteiros de qualquer tipo, e as sintaxes de declaração possíveis são as seguintes

- ▶ **Operador indireto junto ao nome da variável**

```
int    *p;    /* Ponteiro int */  
char   *p;    /* Ponteiro char */  
String *p;    /* Ponteiro para um tipo */  
                /* Definido pelo usuário */
```

- ▶ **Operador indireto junto ao nome do tipo**

```
int*    p;    /* Ponteiro int */  
char*   p;    /* Ponteiro char */  
String* p;    /* Ponteiro para um tipo */  
                /* Definido pelo usuário */
```

---

# Ponteiros sem funções

---

- ▶ **Se vários ponteiros são declarados em uma mesma instrução, o tipo deve ser inserido somente uma vez; o asterisco, todas as vezes**

```
int* p, * p1, * p2;  
int *p, *p1, *p2;
```

- ▶ **Inicializando ponteiros**

```
int i;  
int *pi=&i;  
int *pj, *pi=&i, *px;
```

- ▶ **Ponteiros e variáveis simples declarados em uma única instrução**

```
int *p, i, j, *q;  
int *px=&x, i, j=5, *q;
```

---



# Ponteiros e variáveis apontadas

---

- ▶ Você pode usar ponteiros para executar qualquer operação na variável apontada

```
/* PtrVar1.C */
/* Mostra a inicialização do ponteiro */
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int x,y;
    int *px = &x;      /* Inicializa px com o endereço de x */

    *px = 14;           /* O mesmo que x = 14 */
    y = *px;            /* O mesmo que y = x */

    printf("y = %d\n", y);

    system("PAUSE");
    return 0;
}
```

---

# Ponteiros e variáveis apontadas

---

- ▶ Você pode usar ponteiros para executar qualquer operação na variável apontada

```
/* PtrVar1.C */
/* Mostra a inicialização do ponteiro */
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int x,y;
    int *px = &x;      /* Inicializa px com o endereço de x */

    *px = 14;           /* O mesmo que x = 14 */
    y = *px;            /* O mesmo que y = x */

    printf("y = %d\n", y);

    system("PAUSE");
    return 0;
}
```

- ▶ Nesse programa, usamos o ponteiro para atribuir um valor à variável **x**
  - ▶ Em seguida, usamos novamente o ponteiro para atribuir esse valor a **y**
  - ▶ O operador indireto resulta o nome da variável apontada
-

# Operações com ponteiros

---

- ▶ C permite várias operações básicas com ponteiros
- ▶ Nosso próximo exemplo imprime os resultados de cada operação, o valor do ponteiro, o valor da variável apontada e o endereço do próprio ponteiro

```

/* PtrOperacoes.C */
/* Mostra as operações possíveis com ponteiros */
#include <stdio.h>
#include <stdlib.h>

int main()
{
    unsigned int x=5, y=6;
    unsigned int *px, *py;

    px = &x;    /* Atribuições */
    py = &y;

    if( px < py) /* Comparações */
        printf("py-px= %u\n", (py-px)); /* Subtração */
    else
        printf("px-py= %u\n", (px-py));

    printf("px = %p", px);
    printf(", *px = %u", *px);    /* Op.Indireto */
    printf(", &px = %p\n", &px); /* Op.Endereços */

    printf("py = %p", py);
    printf(", *py = %u", *py);
    printf(", &py = %p\n", &py);

    py++; /* Incremento */

    printf("py = %p", py);
    printf(", *py = %u", *py);
    printf(", &py = %p\n", &py);

    px = py + 5; /* Somar inteiros */

    printf("px = %p", px);
    printf(", *px = %u", *px);
    printf(", &px = %p\n", &px);

    printf("px-py= %u\n", (px-py));

    system("PAUSE");
    return 0;
}

```

# Ponteiros no lugar de matrizes

---

- ▶ Em C, o relacionamento entre ponteiros e matrizes é tão estreito que estes deveriam ser realmente tratados juntos
  - ▶ O compilador transforma matrizes em ponteiros, pois a arquitetura do microcomputador compreende ponteiros, e não matrizes
  - ▶ Qualquer operação que possa ser feita com índices de uma matriz pode ser feita com ponteiros
  - ▶ Para esclarecer a relação entre ponteiros e matrizes vamos examinar um simples programa escrito com matrizes e também com ponteiros
-

# Ponteiros no lugar de matrizes

---

```
/* Matriz.C */
/* Imprime os elementos de uma matriz */
#include <stdio.h>
#include <stdlib.h>

int main()
{
    static int M[5]={92,81,70,69,58};
    int i;
    for(i=0; i<5; i++)
        printf("%d\n", M[i]); /* Notação matriz */

    system("PAUSE");
    return 0;
}
```

---

```
/* PMatriz.C */
/* Imprime os elementos de uma matriz usando notação ponteiro */
#include <stdio.h>
#include <stdlib.h>

int main()
{
    static int M[5]={92,81,70,69,58};
    int i;
    for(i=0; i<5; i++)
        printf("%d\n", *(M + i)); /* Notação ponteiro */

    system("PAUSE");
    return 0;
}
```

---

# Ponteiros no lugar de matrizes

---

- ▶ A expressão  $*(\mathbf{M} + \mathbf{i})$  tem exatamente o mesmo valor de  $\mathbf{M}[\mathbf{i}]$
- ▶ Você já sabe que  $\mathbf{M}$  é um ponteiro `int` e aponta para  $\mathbf{M}[0]$ , conhecendo também a aritmética com ponteiros
- ▶ Assim, se somarmos  $\mathbf{1}$  a  $\mathbf{M}$ , obteremos o endereço  $\mathbf{M}[1]$ 
  - ▶  $\mathbf{M} + 2$  é o endereço de  $\mathbf{M}[2]$
  - ▶ assim por diante
- ▶ Como regra geral, temos que

$\mathbf{M} + \mathbf{i}$  é equivalente a  $\&\mathbf{M}[\mathbf{i}]$ , portanto  
 $*\mathbf{M}(\mathbf{i} + 1)$  é equivalente a  $\mathbf{M}[\mathbf{i}]$

# Ponteiros constantes e ponteiros variáveis

---

- ▶ Analisando o exemplo anterior, você poderia perguntar se a instrução

```
printf("%d\n", *(M + i));
```

não poderia ser simplificada e substituída por

```
printf("%d\n", *(M ++));
```

A resposta é não!

A razão é que não podemos incrementar uma constante.

Da mesma forma que existem inteiros constantes e inteiros variáveis, existem ponteiros constantes e ponteiros variáveis.

O nome de uma matriz é um ponteiro constante

Isso vale para qualquer constante.

Um ponteiro variável é um lugar de memória que armazena um endereço.

Um ponteiro constante é um endereço, uma simples referência.

Vamos reescrever o programa anterior usando um ponteiro variável.

---



# Ponteiros constantes e ponteiros variáveis

---

```
/* PMatriz1.C */
/* Imprime os elementos de uma matriz usando notação ponteiro */
#include <stdio.h>
#include <stdlib.h>

int main()
{
    static int M[5]={92,81,70,69,58};
    int i, *p = M; /* cria e inicializa o ponteiro variável */

    for(i=0; i<5; i++)
        printf("%d\n", *(p++)); /* Notação ponteiro */

    system("PAUSE");
    return 0;
}
```

- ▶ Nesta versão, definimos um ponteiro para um **int** e o inicializamos com o nome da matriz
  - ▶ Agora, podemos usar **p** em todo lugar do programa que usava **M**
  - ▶ Como **p** é um ponteiro variável e não uma constante, podemos usar expressões como `*(p++)`
-

# Passando matrizes como argumento para funções

---

- ▶ Quando uma função recebe o endereço de uma matriz como argumento, ela o declara usando o nome do tipo e colchetes (**[ ]**)
  - ▶ Essa notação declara ponteiros constantes e não ponteiros variáveis
  - ▶ Entretanto, é mais conveniente usar a notação ponteiro no lugar da notação matriz
  - ▶ A notação ponteiro declara um ponteiro variável
- ▶ O próximo exemplo modifica o programa **Notas1.c** para que use ponteiros no lugar de matriz

# Exemplo – Notas1.c

---

```
/* Notas1.C */
/* Mostra passagem de matrizes para funções como argumento */
#include <stdio.h>
#include <stdlib.h>
#define TAMANHO 50

float media(float[], int); /* protótipo */

int main()
{
    float notas[TAMANHO] , m;
    int i=0;

    do
    {
        printf("Digite a nota do aluno %d: ", i+1);
        scanf("%f",&notas[i]);
    } while( notas[i++] >= 0.0);

    i--; /* remove o item de término */

    m = media( notas, i );

    printf("Media das notas: %.2f\n", m);

    system("PAUSE");
    return 0;
}

/* Calcula a média dos valores da matriz */
float media(float lista[], int tamanho)
{
    int i;
    float m=0.0;
    for(i=0; i < tamanho ; i++) m += lista[i];
    return m/tamanho ;
}
```

---

# Exemplo – pnotas1.c

---

```
/* PNotas1.C */
/* Mostra passagem de matrizes para funções usando ponteiros */
#include <stdio.h>
#include <stdlib.h>
#define TAMANHO 50

float media(float *, int); /* protótipo */

int main()
{
    float notas[TAMANHO] , m;
    int i=0;

    do
    {
        printf("Digite a nota do aluno %d: ", i+1);
        scanf("%f", notas + i);
    } while( *(notas + i++) >= 0.0);

    i--; /* remove o item de término */

    m = media( notas, i );

    printf("Media das notas: %.2f\n", m);

    system("PAUSE");
    return 0;
}

/* Calcula a média dos valores da matriz */
float media(float *lista, int tamanho)
{
    int i;
    float m=0.0;
    for(i=0; i < tamanho ; i++) m += *(lista++);
    return m/tamanho ;
}
```

---

# Ponteiros e strings

---

- ▶ Strings são matrizes do tipo **char**
  - ▶ Dessa forma, a notação ponteiro pode ser aplicada
- ▶ Como primeiro exemplo, vamos escrever uma função que procure um caractere em uma cadeia de caracteres
  - ▶ Essa função retorna o endereço da primeira ocorrência do caractere, se este existir, ou o endereço zero, caso o caractere não seja encontrado

# Exemplo – strprocura.c

---

```
/* StrProcura.C */
/* Procura um caractere numa cadeia de caracteres */
#include <stdio.h>
#include <stdlib.h>

char * procura(char *, char );/* protótipo */

int main()
{
    char str[81], *ptr;

    printf("Digite uma frase:\n");
    gets(str);

    ptr = procura(str, 'h');

    printf("\nA frase começa no endereço %p\n", ptr);

    if(ptr)
    {
        printf("\nPrimeira ocorrência do caractere 'h': %p\n", ptr);
        printf( "\nA sua posicao e: %d\n", ptr-str);
    } else
        printf( "O caractere 'h' nao existe nesta frase.\n");

    system("PAUSE");
    return 0;
}

/* Procura um caractere numa frase */
char *procura(char *s, char ch)
{
    while( *s != ch && *s != '\0') s++;
    if(*s != '\0') return s;
    return (char *)0;
}
```

---

# Ponteiros para uma cadeia de caracteres constante

---

- ▶ Vamos analisar duas maneiras de inicializar cadeias de caracteres constante
  - ▶ Usando um ponteiro constante e usando um ponteiro variável

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

int main()
{
    char s1[] = "Saudacoes!";
    char *s2 = "Saudacoes!";

    printf("%p\n", s1);
    printf("%p\n", s2);

    /* *s1++; ERRO. Não podemos incrementar uma constante */
    s2++; /* OK */

    printf("%s\n", s2-1); /* Imprime: Saudacoes! */

    system("PAUSE");
    return 0;
}
```

---

# Matrizes de ponteiros

---

- ▶ Matrizes de ponteiros são essencialmente utilizadas para substituir matrizes de duas dimensões, em que cada elemento é uma cadeia de caracteres
- ▶ Esse uso permite uma grande economia de memória
  - ▶ Pois não teremos a desvantagem de dimensionar todos os elementos com o tamanho da maior cadeia
- ▶ Vamos modificar o programa **diadasemana.c**, da aula de matrizes, para que utilize uma matriz de ponteiros no lugar de uma de strings



# Exemplo – pdiadasemana.c

---

```
/* PDiaSemana.C */
/* Imprime o dia da semana a partir de uma data */
/* Mostra o uso de uma matriz de ponteiros */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>      /* para getch() */

int dsemana(int, int, int); /*Protótipo */

int main()
{
    static char *diasemana[7]=
    {
        "Domingo",
        "Segunda-feira",
        "Terça-feira",
        "Quarta-feira",
        "Quinta-feira",
        "Sexta-feira",
        "Sabado"
    };

    int dia, mes, ano;
    const char ESC = 27;
    do
    {
        printf("Digite a data na forma dd mm aaaa: ");
        scanf("%d%d%d", &dia, &mes, &ano);
        printf("%s\n", diasemana [ dsemana(dia,mes,ano) ] );
        printf("ESC para terminar ou ENTER para recomecar\n");
    } while (getch() != ESC);

    system("PAUSE");
    return 0;
}

/* Encontra o dia da semana a partir de uma data
 * Retorna 0 para domingo, 1 para segunda-feira etc.
 */
int dsemana(int dia, int mes, int ano)
{
    int dSemana = ano + dia + 3 * (mes - 1) - 1;
    if( mes < 3)
        ano--;
    else
        dSemana -= (int)(0.4*mes+2.3);
    dSemana += (int)(ano/4) - (int)((ano/100 + 1)*0.75);
    dSemana %= 7;
    return dSemana;
}
```

---

# Exemplo – `pdiadasemana.c`

---

- ▶ Na versão matriz, as cadeias de caracteres são guardadas na memória em sete posições de 14 bytes cada uma, portanto, ocupando 98 bytes
  - ▶ Na nova versão, as cadeias são guardadas de forma a ocupar somente o número de bytes necessários para seu armazenamento
  - ▶ Cada elemento da matriz é um ponteiro e não mais outra matriz
-

# Matriz de strings e a memória alocada

---

- ▶ A versão matriz aloca 98 bytes de memória da seguinte forma

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	D	o	m	i	n	g	o	\0						
1	S	e	g	u	n	d	a	-	f	e	i	r	a	\0
2	T	e	r	ç	a	-	f	e	i	r	a	\0		
3	Q	u	a	r	t	a	-	f	e	i	r	a	\0	
4	Q	u	i	n	t	a	-	f	e	i	r	a	\0	
5	S	e	x	t	a	-	f	e	i	r	a	\0		
6	S	á	b	a	d	o	\0							

► A versão ponteiros aloca 79 bytes de memória da seguinte forma

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	D	o	m	i	n	g	o	\0	S	e	g	u	n	d
1	a	-	f	e	i	r	a	\0	T	e	r	ç	a	-
2	f	e	i	r	a	\0	Q	u	a	r	t	a	-	f
3	e	i	r	a	\0	Q	u	i	n	t	a	-	f	e
4	i	r	a	\0	S	e	x	t	a	-	f	e	i	r
5	a	\0	S	á	b	a	d	o	\0					

# Ponteiros para ponteiros

---

- ▶ No próximo programa, mostraremos a ordenação de nomes por meio de uma matriz de ponteiros
- ▶ No lugar de ordenar os próprios nomes, ordenaremos a matriz de ponteiros em que cada elemento aponta para um dos nomes
- ▶ Os elementos da matriz de ponteiros são ponteiros que apontam para outros ponteiros

# Exemplo – psrtsort.c

```
/* PStrSort.C */
/* Mostra o uso de ponteiros para ponteiros */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define NOME_MAX 30
#define TAM_MAX 100

void ordena(char **p,int n)
{
    char *temp;
    int i,j;

    for(i=0;i<n-1;i++)
    {
        for(j=i+1; j<n;j++)
            if( strcmp(p[i],p[j])>0)
            {
                temp = *(p+i);
                *(p+i) = *(p+j);
                *(p+j) = temp;
            }
    }
}

int main()
{
    char nomes[NOME_MAX][TAM_MAX];
    char *p[30]; /* matriz de ponteiros */
    int n, i;

    for(n=0;;n++)
    {
        printf("Digite nome ou [ENTER] para fim: ");
        gets(nomes[n]);
        if(strlen(nomes[n])==0) break;
        p[n] = nomes[n];
    }

    printf("\n\nLista original:\n");
    for(i=0;i<n;i++)
        printf("%s\n", p[i]);

    ordena(p,n);

    printf("\n\nLista ordenada:\n");
    for(i=0;i<n;i++)
        printf("%s\n", p[i]);

    system("PAUSE");
    return 0;
}
```

# Ordenando ponteiros

---

- ▶ A novidade desse último programa está em como os nomes são ordenados
- ▶ Na realidade, a função **ordena( )** não ordena os nomes, mas sim os ponteiros para os nomes
- ▶ Essa ordenação é muito mais rápida que a da própria matriz **nomes[ ] [ ]**
  - ▶ pois estamos movimentando ponteiros pela memória, e não re-arranjando cada letra de cada nome
- ▶ Os ponteiros são variáveis que ocupam pouco lugar de memória, entretanto os nomes podem ocupar muita memória

# Ordenando ponteiros

---

- ▶ Observe a declaração do primeiro argumentos de **ordena( )**

```
void ordena(char **p, int n)
```

- ▶ O tipo da variável **p** é **char\*\***
  - ▶ Essa notação indica que **p** é um ponteiro duplamente indireto
  - ▶ Quando o endereço de um nome é passado para uma função como argumento, você já sabe que o seu tipo é **char\***
  - ▶ Um único asterisco é usado para indicar o endereço de um único nome
-



# Ordenando ponteiros

---

- ▶ Observe a declaração do primeiro argumentos de **ordena( )**

```
void ordena(char **p, int n)
```

- ▶ A função **ordena( )** não recebe o endereço de um nome, mas o de uma matriz de ponteiros para nomes
  - ▶ O nome de uma matriz é um ponteiro para um elemento dela
  - ▶ No caso da nossa matriz, um elemento é um ponteiro
  - ▶ Portanto, o nome da matriz é um ponteiro que aponta para outro ponteiro
  - ▶ Dois asteriscos são usados para indicar um ponteiro para ponteiro
-

# Notação ponteiro para matrizes de ponteiros

---

- ▶ Cada elemento da matriz **p** é um ponteiro para um nome
- ▶ Assim, **p[i]** é o endereço de um nome
- ▶ Você já sabe que um elemento de uma matriz pode ser escrito em notação ponteiro
- ▶ Dessa forma, podemos escrever a expressão **p[i]** como

$$*(p+i)$$

# Ponteiros para funções

---

- ▶ Apresentaremos um tipo de ponteiro especial: um ponteiro que aponta para uma função, ou seja, uma variável que irá conter o endereço de uma função
- ▶ O nosso primeiro exemplo mostra um ponteiro para a função **doisbeep( )**

```
/* PtrFunc.C */
/* Mostra o uso de ponteiro para função */
#include <stdio.h>
#include <stdlib.h>

void doisbeep(void); /* protótipo */

int main()
{
    void (*pf) (void); /* ponteiro p/ função void que recebe void */

    pf = doisbeep; /* nome da função sem os parênteses */

    (*pf) (); /* chama a função */

    system("PAUSE");
    return 0;
}

/* doisbeep() */
/* toca o alto-falante duas vezes */
void doisbeep(void)
{
    unsigned i;
    printf("\a");
    for(i=0; i < 800000 ; i++); /* dar um tempo */
    printf("\a");
}
```

---

# Análise do exemplo

---

## ► Declarando o ponteiro para função

- A função **main( )** começa declarando **pf** como um ponteiro para uma função **void**
- É claro que o tipo **void** é uma das possibilidades
- Se a função a ser apontada é do tipo **float**, por exemplo, o ponteiro deve ser declarado como tal

```
void (*pf) (void) ;
```

## ► Observe os parênteses envolvendo **\*pf**

- Esses parênteses são realmente necessários, pois, se omitidos

```
void *pf(void) ; /* ERRO: é um protótipo de função */
```

- Estaríamos declarando **pf** como sendo uma função que retorna um ponteiro **void**
-

# Análise do exemplo

---

## ► Endereços de funções

- O nome de uma função desacompanhado de parênteses é o seu endereço
- A instrução

```
pf = doisbeep; /* Nome da função sem os parênteses */
```

atribui o endereço da função **doisbeep( )** a **pf**

- Observe que não colocamos parênteses junto ao nome da função
- Se eles estivessem presentes, como em

```
pf = doisbeep( ); /* ERRO */
```

estariamos atribuindo a **pf** o valor de retorno da função, e não o seu endereço

---

# Análise do exemplo

---

## ▶ Executando a função por meio do ponteiro

- ▶ Da mesma forma que podemos substituir o nome de uma variável usando um ponteiro acompanhado do operador indireto \*
- ▶ Podemos substituir o nome da função usando o mesmo mecanismo

```
(*pf)( ); /* Chama a função */
```

É equivalente a

```
doisbeep( );
```

e indica uma chamada à função **doisbeep( )**

---

# Ponteiros para funções como argumentos

---

- ▶ O exemplo a seguir cria um ponteiro para armazenar o endereço da função de biblioteca **gets( )**

- ▶ Essa função tem o seguinte protótipo

```
char *gets(char *);
```

definido no arquivo **stdio.h**

- ▶ A função retorna um ponteiro para a cadeia de caracteres lida do teclado e armazenada no endereço recebido por ela como argumento

# Exemplo – ptrgets.c

---

```
/* PtrGets.C */
/* Mostra o uso de ponteiro como argumento de função */
#include <stdio.h>
#include <stdlib.h>

void func( char * (*)(char *));

int main()
{
    char * (*p)(char *);
    p = gets;
    func(p);

    system("PAUSE");
    return 0;
}

void func(char * (*p)(char *))
{
    char nome[80];

    printf("Digite seu nome: ");

    (*p)(nome); /* chama a função gets() */

    printf("Seu nome e: %s\n", nome);
}
```

---



# Matrizes de ponteiros para funções

---

- ▶ Os ponteiros para funções oferecem uma maneira eficiente de executar uma função
  - ▶ Com base em alguma escolha dependente de parâmetros conhecidos somente em tempo de execução
  - ▶ Por exemplo, suponhamos que você queira escrever um programa em que é apresentado um “menu” de opções ao usuário
    - ▶ Para cada escolha, o programa deve executar uma chamada a uma função particular
    - ▶ Você poderia utilizar estruturas tradicionais de programação, como **switch** ou **if-else**, ou qualquer outra estrutura de controle para decidir qual função deve ser chamada
    - ▶ Você também pode, simplesmente, criar uma matriz de ponteiros para funções e executa a função correta por meio de seu ponteiro
-

# Exemplo – fptrmatriz.c

```
/* FPtrMatriz.C */
/* Mostra uma matriz de ponteiros para função */
#include <stdio.h>
#include <stdlib.h>

const int TRUE=1;

void func0(void), func1(void), func2(void); /* Protótipos */

int main()
{
    void (*ptrf[3])(void); /* Matriz de ponteiros para funções */

    ptrf[0] = func0;
    ptrf[1] = func1;
    ptrf[2] = func2;

    do
    {
        int i;
        printf("0 - ABRIR\n");
        printf("1 - FECHAR\n");
        printf("2 - SALVAR\n");
        printf("\nEscolha um item: ");
        scanf("%d", &i);
        if(i < 0 || i > 2) break;

        (*ptrf[i])(); /* Chama função */

    } while(TRUE);
    system("PAUSE");
    return 0;
}

void func0()
{
    printf("\n*** Estou em func0() ***\n");
}

void func1()
{
    printf("\n*** Estou em func1() ***\n");
}

void func2()
{
    printf("\n*** Estou em func2() ***\n");
}
```

# Inicializando uma matriz de ponteiros para funções

---

- ▶ O programa anterior poderia ter inicializado a matriz **ptrf** na mesma instrução de sua declaração
- ▶ Eis a modificação

```
/* FPtrMatriz1.C */
/* Mostra uma matriz de ponteiros para função */
#include <stdio.h>
#include <stdlib.h>

-----

const int TRUE=1;

void func0(void), func1(void), func2(void); /* Protótipos */

int main()
{
    void (*ptrf[3])(void)={ func0,func1,func2}; /*Matriz inicializada*/

    do
    {
        int i;
        printf("0 - ABRIR\n");
        printf("1 - FECHAR\n");
        printf("2 - SALVAR\n");
        printf("\nEscolha um item: ");
        scanf("%d", &i);
        if(i < 0 || i > 2) break;

        (*ptrf[i])(); /* Chama função */

    } while(TRUE);
    system("PAUSE");
    return 0;
}

void func0()
{
    printf("\n*** Estou em func0() ***\n");
}

void func1()
{
    printf("\n*** Estou em func1() ***\n");
}

void func2()
{
    printf("\n*** Estou em func2() ***\n");
}
-----
```

# Usando **typedef** para declarar um ponteiro para função

---

- ▶ É comum definir um nome para um tipo de dado que seja ponteiro para função
- ▶ Fazemos isso por meio de **typedef**

```

/* FPTrMatriz2.C */
/* Mostra uma matriz de ponteiros para função */
#include <stdio.h>
#include <stdlib.h>
void func0(void), func1(void), func2(void); /* Protótipos */

-----

const int TRUE=1;

typedef void (*PFunc) (void); /* PFunc é ponteiro p/ função void */

int main()
{
    PFunc ptrf[3] = { func0, func1, func2}; /* Matriz inicializada */

    do
    {
        int i;
        printf("0 - ABRIR\n");
        printf("1 - FECHAR\n");
        printf("2 - SALVAR\n");
        printf("\nEscolha um item: ");
        scanf("%d", &i);
        if(i < 0 || i > 2) break;

        (*ptrf[i]) (); /* Chama função */

    } while(TRUE);
    system("PAUSE");
    return 0;
}

void func0()
{
    printf("\n*** Estou em func0() ***\n");
}

void func1()
{
    printf("\n*** Estou em func1() ***\n");
}

void func2()
{
    printf("\n*** Estou em func2() ***\n");
}
-----

```

# Ponteiros **void**

---

- ▶ O ponteiro do tipo **void** pode apontar para qualquer tipo de dado
- ▶ Pode ser declarado por meio da seguinte instrução

```
void *p; /* p aponta para qualquer tipo de dado */
```

- ▶ Ponteiros do tipo **void** são usados em situações em que seja necessário que uma função receba ou retorne um ponteiro genérico
  - ▶ E opere independentemente do tipo de dado apontado
- ▶ Qualquer endereço pode ser atribuído a um ponteiro **void**

# Exemplo – ptrvoid.c

---

```
/* PtrVoid.C */
/* Mostra ponteiros void */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int i=5;
    float f=3.2;

    void *pv;    /* Ponteiro genérico */
    pv = &i;     /* Endereço de um int */

    /* não podemos usar o operador indireto com ponteiros void */
    printf("%d\n", *pv); /* ERRO de compilação */

    pv = &f;     /* Endereço de um float */

    /* não podemos usar o operador indireto com ponteiros void */
    printf("%f\n", *pv); /* ERRO de compilação */

    system("PAUSE");
    return 0;
}
```

---



# Exemplo – ptrvoid.c

---

```
/* PtrVoid.C */
/* Mostra ponteiros void */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int i=5;
    float f=3.2;

    void *pv;    /* Ponteiro genérico */
    pv = &i;     /* Endereço de um int */

    /* não podemos usar o operador indireto com ponteiros void */
    printf("%d\n", *pv); /* ERRO de compilação */

    pv = &f;     /* Endereço de um float */

    /* não podemos usar o operador indireto com ponteiros void */
    printf("%f\n", *pv); /* ERRO de compilação */

    system("PAUSE");
    return 0;
}
```

- ▶ O conteúdo da variável apontada por um ponteiro **void** não pode ser acessado por meio desse ponteiro
  - ▶ É necessário criar outro ponteiro e fazer a conversão de tipo na atribuição
-

# Exemplo – ptrvoid1.c

---

```
/* PtrVoid1.C */
/* Mostra ponteiros void */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int i=5, *pi;
    float f=3.2, *pf;

    void *pv;    /* Ponteiro genérico */
    pv = &i;     /* Endereço de um int */

    pi = (int *)pv; /* Convertendo o tipo do ponteiro */
    printf("%d\n", *pi); /* Correto */

    pv = &f;     /* Endereço de um float */

    pf = (float *) pv; /* Convertendo o tipo do ponteiro */
    printf("%f\n", *pf); /* Correto */

    system("PAUSE");
    return 0;
}
```

---

# A função **qsort( )**

- ▶ A função de biblioteca-padrão **qsort( )** executa a ordenação de uma matriz por meio da implementação do algoritmo “**quick-sort**”
- ▶ A função é do tipo **void** e recebe quatro argumentos
  1. O endereço da matriz a ser ordenada;
  2. Um inteiro com o número de elementos da matriz;
  3. O tamanho em bytes de um elemento da matriz
  4. O endereço de uma função com o seguinte protótipo

```
int compara(const void *, const void *);
```

- ▶ A função **compara( )** deve ser escrita pelo usuário de **qsort( )** e compara dois elementos da matriz
  - ▶ Retorna um valor que informa o resultado da comparação

Valor retornado pela função <b>compara( )</b>	Descrição
<0	elem1 menor que elem2
0	elem1 igual ao elem2
>0	elem1 maior que elem2

# Ordenando números inteiros

---

- O exemplo mostra a ordenação de uma matriz de inteiros

```
/* QsortI.C */
/* Algoritmo qsort com inteiros */
#include <stdio.h>
#include <stdlib.h>

/* qsort - função para comparar inteiros */
int compara(const void *a, const void *b)
{
    const int *pa = (const int *)a; /* modifica o tipo do ponteiro */
    const int *pb = (const int *)b; /* modifica o tipo do ponteiro */
    return *pa - *pb; /* Retorna negativo se a<b e positivo se a > b */
}

int main()
{
    unsigned int tamanho, i;

    int tab[]={ 234, 760, 162, 890, -23, 914, 567, 888, 398, -45};

    tamanho = sizeof(tab)/sizeof(int);

    qsort(tab, tamanho, sizeof(int), compara);

    for(i=0; i< 8; i++) printf("%d\n", tab[i]);
    system("pause");
    return 0;
}
```

---

# Algoritmo de procura binária

---

- ▶ O algoritmo de procura binária é, de modo geral, o mais eficiente, mas requer que a lista de itens esteja ordenada
    - ▶ Demora menos para encontrar o valor
  - 1. Seleciona um valor no meio da lista e o compara ao valor que está sendo procurando
  - 2. Se o valor procurado for maior que o valor selecionado, repete-se o processo para a metade da lista posterior ao valor selecionado
  - 3. Se o valor procurado for menor que o valor selecionado, repete-se o processo para a metade da lista anterior ao valor selecionado
  - 4. O processo é repetido até um valor ser encontrado, ou até que a metade em que a procura deverá ser feita seja esvaziada
    - ▶ Neste último caso, o valor não está na lista
-

# Procura binária com números inteiros

---

- ▶ O próximo exemplo implementa o algoritmo de procura binária com inteiros

```

/* BQsortI.C */
/* Algoritmo qsort e binarySearch com inteiros */
#include <stdio.h>
#include <stdlib.h>

/* Função procura binária para inteiros
 * Procura entre MatrizOrdenada[inicio]..MatrizOrdenada[fim] pela chave.
 * Retorna o índice do elemento encontrado ou -1 se não foi encontrado */
-----
int binarySearchInt(int MatrizOrdenada[], int inicio, int fim, int chave)
{
    while (inicio <= fim)
    {
        int meio = (inicio + fim) / 2; /* divide ao meio */
        if (chave > MatrizOrdenada[meio])
            inicio = meio + 1; /* repete a procura a partir do meio */
        else if (chave < MatrizOrdenada[meio])
            fim = meio - 1; /* repete a procura até o meio */
        else
            return meio; /* encontrado, retorna posição */
    }
    return -1; /* não foi encontrado */
}

/* qsort - função para comparar inteiros */
int compara(const void *a, const void *b)
{
    const int *pa = (const int *)a; /* modifica o tipo do ponteiro */
    const int *pb = (const int *)b; /* modifica o tipo do ponteiro */
    return *pa - *pb; /* Retorna negativo se a<b e positivo se a > b */
}

int main()
{
    unsigned int tamanho, i, procura;

    int tab[]={ 234, 760, 162, 890, -23, 914, 567, 888, 398, -45};
    printf("\nMatriz Original\n");
    for(i=0; i< 8; i++) printf("%d\n", tab[i]);
    tamanho = sizeof(tab)/sizeof(int);

    qsort(tab, tamanho, sizeof(int), compara);
    printf("\nMatriz Ordenada\n");
    for(i=0; i< 8; i++) printf("%d\n", tab[i]);

    procura = binarySearchInt(tab, 0, tamanho - 1, 567);
    printf("\n\nÍndice de 567 = %d\n",procura);

    system("pause");
    return 0;
}
-----

```

# Procura binária com c-string

```
/* BQsortStr.C */
/* Algoritmo qsort e binarySearch com C-string */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Função procura binária para C-string
 * Procura entre MatrizOrdenada[inicio]..MatrizOrdenada[fim] pela chave.
 * Retorna o índice do elemento encontrado ou -1 se não foi encontrado */
int binarySearchStr(char *MatrizOrdenada[], int inicio, int fim, char*chave)
{
    while (inicio <= fim)
    {
        int meio = (inicio + fim) / 2; /* divide ao meio */
        int cmp = strcmp(chave, MatrizOrdenada[meio]);

        if (cmp > 0)
            inicio = meio + 1; /* repete a procura a partir do meio */
        else if (cmp < 0)
            fim = meio - 1; /* repete a procura até o meio */
        else
            return meio; /* encontrado, retorna posição */
    }
    return -1; /* não foi encontrado */
}

/* qsort - função para comparar C-string (matriz char) */
int cstring_cmp(const void *a, const void *b)
{
    const char **pa = (const char **)a;
    const char **pb = (const char **)b;
    return strcmp(*pa, *pb);
    /* strcmp -1 se a < b, 1 se a>b e 0 se a==b */
}

int main()
{
    unsigned int tamanho, i, procura;
    char *strings[] = { "Zuleima", "Andre", "Carolina", "Beto", "Fabio", "Denise" };
    tamanho = sizeof(strings) / sizeof(char *);
    puts("Matriz Original");
    for(i=0; i< tamanho; i++) printf("%s\n", strings[i]);

    qsort(strings, tamanho, sizeof(char *), cstring_cmp);

    puts("\nMatriz Ordenada");
    for(i=0; i< tamanho; i++) printf("%s\n", strings[i]);

    procura = binarySearchStr(strings, 0, 5, "Carolina");
    printf("\nÍndice de Carolina = %d\n",procura);

    system("pause");
    return 0;
}
```



# Ponteiros para estruturas

---

- ▶ Existem diversos motivos para se usar ponteiros para estruturas
- ▶ Por exemplo, se você quiser usar a função **qsort( )** para ordenar os dados de uma matriz de estruturas
  - ▶ Você deverá escrever a função de comparação que recebe ponteiros

```

/* BQsortStruct.C */
/* Algoritmo qsort e binarySearch com estruturas */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/* Exemplo com estrutura */
struct est_pop
{
    char estado[16];
    int pop; /* populacao */
};

/* qsort - função para comparar struct por populacao (membro int) */
int struct_cmp_por_pop(const void *a, const void *b)
{
    struct est_pop *pa = (struct est_pop *)a;
    struct est_pop *pb = (struct est_pop *)b;
    return pa->pop - pb->pop; /* Retorna neg. se a<b e pos. se a>b */
}

/* qsort - função para comparar struct por estado (membro C-string) */
int struct_cmp_por_estado(const void *a, const void *b)
{
    struct est_pop *pa = (struct est_pop *)a;
    struct est_pop *pb = (struct est_pop *)b;
    return strcmp(pa->estado, pb->estado);
}

int main()
{
    unsigned int tamanho, i;
    struct est_pop structs[] = /* populacao dividido por 1000 */
        {{ "Sergipe", 1968 }, { "Bahia", 13815 },
          { "Piauí", 3007 }, { "Acre", 670 },
          { "Rondônia", 1535 }, { "Tocantins", 1306 } };

    tamanho = sizeof(structs) / sizeof(struct est_pop);
    /* imprime matriz de estruturas original */
    puts("Estrutura Original");
    for(i=0; i<tamanho; i++)
        printf("[ estado: %s \t populacao: %6d000 ]\n",
               structs[i].estado, structs[i].pop);
    puts("=====");
    /* ordena usando a função qsort */
    qsort(structs, tamanho, sizeof(struct est_pop),
          struct_cmp_por_pop);
    /* imprime matriz de estruturas ordenada */
    puts("Estrutura ordenada por populacao");
    for(i=0; i<tamanho; i++)
        printf("[ estado: %s \t populacao: %6d000 ]\n",
               structs[i].estado, structs[i].pop);
    puts("=====");
    puts("Estrutura ordenada por estado");
    /* reordena usando a função qsort */
    qsort(structs, tamanho, sizeof(struct est_pop),
          struct_cmp_por_estado);
    /* imprime matriz de estruturas ordenada */
    for(i=0; i<tamanho; i++)
        printf("[ estado: %s \t populacao: %6d000 ]\n",
               structs[i].estado, structs[i].pop);
    puts("=====");
    system("pause");
    return 0;
}

```

# Declarando um ponteiro para estrutura

---

- ▶ As novidades são as duas funções de comparação

```
/* qsort - função para comparar struct por populacao (membro int) */
int struct_cmp_por_pop(const void *a, const void *b)
{
    struct est_pop *pa = (struct est_pop *)a;
    struct est_pop *pb = (struct est_pop *)b;
    return pa->pop - pb->pop; /* Retorna neg. se a<b e pos. se a>b */
}
```

```
/* qsort - função para comparar struct por estado (membro C-string) */
int struct_cmp_por_estado(const void *a, const void *b)
{
    struct est_pop *pa = (struct est_pop *)a;
    struct est_pop *pb = (struct est_pop *)b;
    return strcmp(pa->estado, pb->estado);
}
```

- ▶ Na instrução

```
    struct est_pop *pa = (struct est_pop *)a;
```

declaramos o ponteiro para estrutura **pa**

- ▶ Primeiro a palavra *struct* seguida da etiqueta *est\_pop*, então o operador indireto \* seguido do nome do ponteiro
  - ▶ A sintaxe é a mesma de qualquer outra declaração de ponteiro que já vimos
-

# Acessando membros por meio de ponteiros

---

- ▶ Você já aprendeu que, se o nome de uma estrutura for conhecido, podemos acessar seus membros usando o operador ponto (.)
- ▶ Será que uma construção análoga, usando um ponteiro em vez do nome da variável, poderia ser escrita?

```
pa.estado /* ERRO */
```

- ▶ A resposta é não, pois **pa** não é uma variável estrutura e sim um ponteiro para uma variável estrutura
  - ▶ Além disso, o operador ponto (.) trabalha somente sobre o nome de uma estrutura

# Acessando membros por meio de ponteiros

---

► C oferece dois métodos para resolver esta questão

1. Obter o nome da variável apontada por **pa** por meio do operador indireto (\*)

```
(*pa).estado /* OK */
```

Entretanto, essa expressão é de visualização complexa por causa dos parênteses

Os parênteses são necessários, pois o operador (.) tem precedência sobre o operador (\*)

2. Por meio do operador de acesso a membros (→) que consiste no sinal de “menos” (-) seguido do sinal de “maior que” (>)

Esse operador trabalha sobre o endereço de uma variável estrutura e não sobre seu nome

```
*pa->estado /* OK. Mais usado. */
```

Um ponteiro para uma estrutura, seguido pelo operador (→) e pelo nome de um membro, trabalha da mesma maneira que o nome da estrutura seguido pelo operador (.) e pelo nome do membro.

---

# Área de alocação dinâmica: **heap**

---

- ▶ A área de alocação dinâmica – também chamada **heap** – consiste em toda memória disponível que não foi usada para 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 a liberação dinâmica de memória
    - ▶ **malloc()**
    - ▶ **calloc()**
    - ▶ **free()**
-

# Alocando e desalocando memória do heap

---

- ▶ Suponhamos que você vá descrever um programa interativo e não conheça de antemão quantas entradas de dados serão fornecidas
    - ▶ O cadastro de livros de uma biblioteca é um bom exemplo
    - ▶ Você pode reservar uma quantidade de memória que pensa ser razoável (declarar uma matriz para armazenar 50 estruturas do tipo Livro)
    - ▶ Nesse caso, o compilador aloca memória para armazenar toda a matriz, e isso se torna ineficiente caso não ocuparmos todo o espaço reservado
    - ▶ Por outro lado, poderia ser necessário armazenar mais de 50 livros, e a matriz não atenderá a esse requisito
  - ▶ A solução para esse tipo de problema é solicitar memória toda vez que se fizer necessário
  - ▶ O mecanismo para aquisição de memória em tempo de execução se dá por meio da função de biblioteca-padrão **malloc( )**
-

# A função **malloc( )**

---

- ▶ A função **malloc( )** recebe como argumento um número inteiro positivo que representa a quantidade de bytes de memória desejada
- ▶ Solicita memória ao sistema operacional e retorna um ponteiro **void** para o primeiro byte do novo bloco de memória que foi alocado
- ▶ Se não houver memória suficiente para satisfazer a exigência, **malloc( )** retornará um ponteiro com valor **NULL**
  - ▶ Os bytes alocados são inicializados com lixo
- ▶ O fragmento de programa a seguir aloca memória para uma variável estrutura do tipo data

```
struct Data *ptr;  
ptr = (struct Data *) (malloc(sizeof(struct Data)));
```

---



# A função **calloc( )**

---

- ▶ A função **calloc( )** aloca uma matriz de elementos inicializados com zero
  - ▶ Internamente, a função **calloc( )** chama a função **malloc( )**
- ▶ A nova função recebe dois números inteiros como argumentos
  1. O primeiro indica o número de itens desejados
  2. O segundo indica o tamanho de cada item
- ▶ Então, retorna um ponteiro **void** apontando para o primeiro byte de bloco solicitado
- ▶ O fragmento de programa a seguir aloca memória para uma matriz de 100 inteiros

```
int *memnova;  
memnova = (int *) calloc(100, sizeof(int));
```

---

# A função **free( )**

---

- ▶ Uma vez alocada memória dinamicamente, ela continuará ocupada até que seja desalocada explicitamente pela função **free( )**
    - ▶ Em outras palavras, uma variável criada com a função **malloc( )** ou **calloc( )** existirá e poderá ser acessada em qualquer parte do programa
      - ▶ Contudo, somente enquanto não for liberada por meio da função **free( )** e seu espaço de memória devolvido ao sistema operacional
  - ▶ A função **free( )** recebe como argumento um ponteiro para uma área de memória previamente alocada por **malloc( )** ou **calloc( )**
    - ▶ E então libera essa área para uma possível utilização futura
-

# Alocação de tipos básicos usando memória dinâmica

---

- ▶ O nosso primeiro exemplo mostra como criar uma variável dinamicamente

```
/* Malloc.C */
/* mostra o uso de malloc() */
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int *pi;

    pi = (int *) malloc(sizeof(int));

    puts("Digite um numero: ");
    scanf("%d", pi);
    printf("\nVoce digitou o numero %d\n", *pi);

    free(pi);

    system("PAUSE");
    return 0;
}
```

---

# Dimensionando matrizes em tempo de execução

---

- ▶ A determinação do tamanho de uma matriz pode ser feita em tempo de execução
- ▶ Vamos modificar o programa **pmedia.c** para que o usuário indique quantas notas serão inseridas

# pmedia.c

```
/* PMedia.C */
/* Mostra passagem de matrizes para funções usando ponteiros */
#include <stdio.h>
#include <stdlib.h>
#define TAMANHO 50

float media(float *, int); /* protótipo */

int main()
{
    float notas[TAMANHO] , m;
    int i=0;

    do
    {
        printf("Digite a nota do aluno %d ", i+1);
        scanf("%f", notas + i);
    } while( *(notas + i++) >= 0.0);

    i--; /* remove o item de término */

    m = media( notas, i );

    printf("Média das notas: %.2f\n", m);

    system("PAUSE");
    return 0;
}

/* Calcula a média dos valores da matriz */
float media(float *lista, int tamanho)
{
    int i;
    float m=0.0;
    for(i=0; i < tamanho ; i++) m += *(lista++);
    return m/tamanho ;
}
```

# pdmedia.c

```
/* PDMedia.C */
/* Alocação dinâmica da matriz com calloc() */
#include <stdio.h>
#include <stdlib.h>

float media(float *, int );/*protótipo */

int main()
{
    float * notas , m;
    int tamanho, i;

    puts("Qual e o numero de notas? ");
    scanf("%d", &tamanho);

    notas = (float *)calloc(tamanho,sizeof(float));

    for(i=0; i < tamanho; i++)
    {
        printf("Digite a nota do aluno %d: ", i + 1);
        scanf("%f", notas+i);
    }

    m = media( notas, tamanho );
    printf("Media das notas: %.2f\n", m);

    free(notas);

    system("PAUSE");
    return 0;
}

/* Calcula a média dos valores da matriz */
float media(float *lista, int tamanho)
{
    int i;
    float m=0.0;
    for(i=0; i < tamanho ; i++) m += *(lista++);
    return m/tamanho ;
}
```

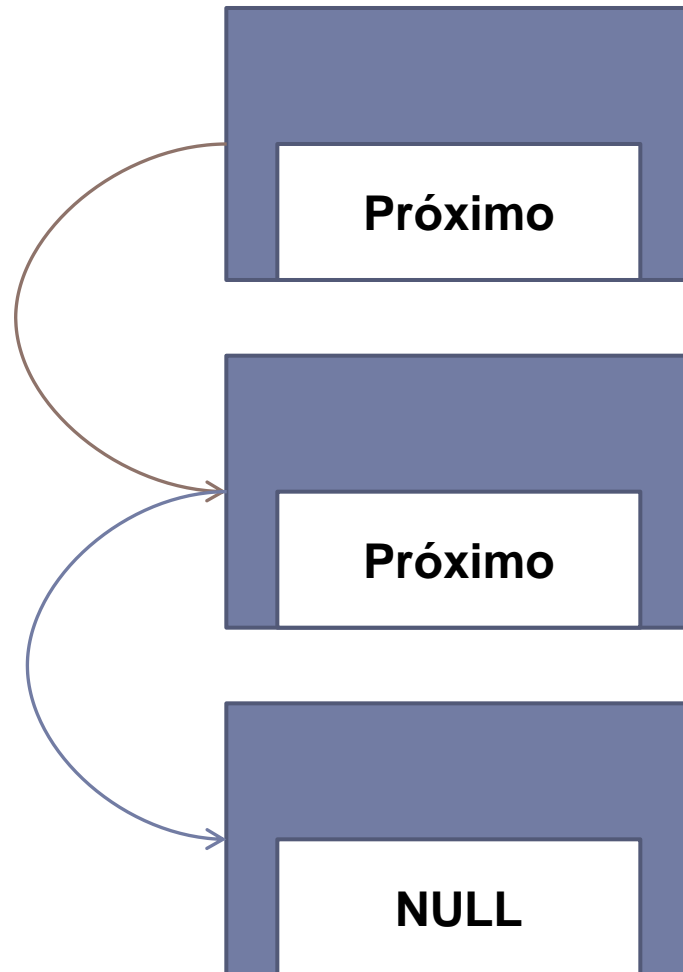
# Criando uma lista ligada

---

- ▶ *Lista ligada* é um algoritmo de armazenamento de dados que muitas vezes supera o uso de uma matriz ou de uma matriz de ponteiros
  - ▶ A *lista ligada* assemelha-se a uma corrente em que os registros de dados estão pendurados sequencialmente
    - ▶ O espaço de memória para cada registro é obtido pela função **malloc( )**, conforme surge a necessidade de adicionar itens à lista
    - ▶ Cada registro é conectado ao próximo por meio de um ponteiro
    - ▶ O último registro contém um ponteiro com o valor NULL, e cada registro anterior contém um ponteiro apontado para o próximo
  - ▶ Cada registro é representado por uma variável estrutura do tipo Livro
    - ▶ Cada estrutura contém um conjunto de membros para armazenar os dados de um livro: título, autor, número de registro e preço
    - ▶ Há um membro a mais, um ponteiro, para armazenar o endereço do próximo registro
  - ▶ A lista, como um todo, é acessada por meio de um ponteiro para a primeira estrutura, chamado cabeça
-

# Esquema de uma lista ligada

---





# O programa lista.c

---

```
/* Lista.C */
/* Mostra a implementação de uma lista ligada */
#include <stdio.h>
#include <stdlib.h>
#include <conio.h> /* para getch() */

typedef struct Livro
{
    char Titulo[30];
    char Autor[30];
    int NumReg;
    double Preco;
    struct Livro *Proximo;
}Livro;

Livro *primeiro, *atual, *NovoLivro;

void GetLivro()
{
    char temp[80];
    NovoLivro = (Livro *)malloc(sizeof(Livro));
    if( primeiro == (Livro *)NULL)
        primeiro = atual = NovoLivro;
    else
    {
        atual = primeiro;
        while( atual->Proximo != (Livro *)NULL)
            atual = atual->Proximo; /* procura novo item */
        atual->Proximo = NovoLivro;
        atual = NovoLivro;
    }
    printf("Digite titulo: ");
    gets(atual->Titulo);
    printf("Digite autor: ");
    gets(atual->Autor);
    printf("Digite o numero do registro: ");
    gets(temp);
    atual->NumReg = atoi(temp);
    printf("Digite o preco: ");
    gets(temp);
    atual->Preco = atof(temp);
    atual->Proximo=(Livro *)NULL;
}

/* continua na próxima página */
```

# O programa lista.c

---

```
void PrintLivro()
{
    if( primeiro == (Livro *)NULL)
    {
        puts("Lista vazia");
        return;
    }

    atual = primeiro;

    do
    {
        printf( "Titulo: %s\n", atual->Titulo);
        printf( "Autor  : %s\n" , atual->Autor);
        printf( "No.Reg: %d\n" , atual->NumReg);
        printf( "Preco  : %.2f\n\n" , atual->Preco);
        atual = atual->Proximo;
    }while(atual != NULL);
}

int main()
{
    char ch;
    primeiro = (Livro *) NULL; /* sem dados ainda */

    do
    {
        GetLivro();
        puts("\nInserir outro livro (s/n)? ");
        ch = getch();
    } while( (ch != 'n') && (ch != 'N'));

    puts("\nLISTA DOS LIVROS CADASTRADOS");
    puts("=====");

    PrintLivro();

    system("PAUSE");
    return 0;
}
```

---

# Considerações do exemplo **lista.c**

---

- ▶ A idéia básica é que cada variável estrutura contenha um ponteiro para a próxima estrutura da lista
    - ▶ e que o ponteiro da última estrutura contenha um ponteiro nulo (NULL)
  - ▶ O ponteiro **primeiro** será usado para armazenar o endereço da primeira estrutura da lista
    - ▶ Esse é um *endereço chave*, visto que indica onde encontrar a lista
    - ▶ O ponteiro é inicializado com NULL por ser externo
  - ▶ NULL é uma constante definida no arquivo **stdio.h** com o valor zero
  - ▶ A linguagem C garante que um ponteiro que aponta para algum endereço válido nunca terá o valor zero, de forma que esse valor pode ser retornado para sinalizar um evento anormal, nesse caso, o fim da lista
  - ▶ Escreveremos NULL em vez de 0 para indicar mais claramente que esse é um valor especial para um ponteiro
    - ▶ Como ele é inteiro e será atribuído a um ponteiro, devemos usar o operador de conversão de tipo (Livro \*)
-

# Considerações do exemplo **lista.c**

---

- ▶ A função **GetLivro( )** adiciona um livro à lista na instrução

```
NovoLivro = (Livro *)malloc(sizeof(Livro));
```

- ▶ Em seguida, a função verifica se esse é o primeiro item a ser colocado na lista

```
if(primeiro == (Livro *)NULL)
    primeiro = atual = NovoLivro;
```

em caso positivo, o novo endereço é atribuído aos ponteiros **primeiro** e **atual**

- ▶ Cada membro individual é acessado por meio do operador  $\rightarrow$  e, finalmente, é atribuído NULL ao membro **Proximo**

# Considerações do exemplo **lista.c**

---

- ▶ Entretanto, se esse não for o primeiro item da lista, o programa usa um laço **while** para encontrar o fim da lista

- ▶ Começa atribuindo primeiro a atual pela expressão

```
atual = primeiro
```

- ▶ E em seguida, o laço **while** verifica se o ponteiro da estrutura atual é NULL

- ▶ Se não for

```
while(atual->Proximo != (Livro *)NULL)
```

```
atual = atual->Proximo; /* Procura novo item */
```

o laço **while** atribui o endereço em **atual->Proximo** a **atual** e volta ao teste

- ▶ O ciclo termina quando é encontrada a última estrutura
-

# Considerações do exemplo **lista.c**

---

- ▶ Ao seu membro ponteiro, atribuímos o endereço da nova estrutura

```
atual->Proximo = NovoLivro;
```

```
atual = NovoLivro;
```

- ▶ Em seguida, os membros da nova estrutura são preenchidos com os dados digitados pelo usuário, com exceção do membro ponteiro

- ▶ A ele atribuímos NULL

```
atual->Proximo=(Livro *)NULL;
```

- ▶ O novo livro é inserido no final da lista
-

# Considerações do exemplo **lista.c**

---

- ▶ Ao seu membro ponteiro, atribuímos o endereço da nova estrutura

```
atual->Proximo = NovoLivro;
```

```
atual = NovoLivro;
```

- ▶ Em seguida, os membros da nova estrutura são preenchidos com os dados digitados pelo usuário, com exceção do membro ponteiro

- ▶ A ele atribuímos NULL

```
atual->Proximo=(Livro *)NULL;
```

- ▶ O novo livro é inserido no final da lista
-