

# **Introdução a Linguagem C (Parte II)**

**UFPA – Sistemas de Informação**

**Roberto Araujo  
2013**

# Programação Modular (Funções)

# Funções

- Fornecem um mecanismo para o desenvolvimento de programas que facilitam:
  - sua escrita, manutenção, depuração, modificação, além de facilitar seu entendimento.
  - Além disso → ***reaproveitamento de código***

Ex:

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

```
main( void ) {
```

```
    void printf("Meu segundo programa em C");
```

```
}
```

São um agrupamento de comandos em bloco. O bloco recebe um nome para execução dos comandos que estão dentro do bloco

# Funções



- Uma função possui:

Um nome

Um tipo para o valor de retorno

Um ou mais parâmetros

Ex: 

 **Tipo de retorno** `void teste ( void ) {`  **Parâmetros**  
`void printf("Minha primeira função");`  
`}`

# Exemplo

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

```
void teste (void) {
```

→ Declaração da função

```
    printf("Minha primeira função \n");
```

```
}
```

↙ Delimita o início e  
o final da função

```
main( ) {
```

```
    teste ( );
```

```
}
```

# Exercício

- Escreva um programa que contenha uma função para imprimir seu nome 5 vezes.

# Passagem de Parâmetros por Valor para a Função

- Uma função pode receber valores
- Esses valores são são variáveis que a função recebe
- Eles são chamados de **parâmetros da função**
- Um ou mais parâmetros podem ser estabelecidos
- Os parâmetros da função são estabelecidos dentro dos parênteses, após o nome da função
- Cada parâmetro possui: **tipo\_variável variável**
- Na **passagem de parâmetros por valor**, a função realiza apenas a leitura dos valores e os utiliza. Não é possível modificar os valores dos parâmetros passados dentro da função.
- Quando uma função é chamada os valores que são passados como argumento são copiados para os parâmetros

# Exemplo

```
/* Calcula o número triangular de um número 'n' */
#include <stdio.h>
void numero_triangular (int n) {
    int i, tri_num = 0;
    for (i = 1; i <= n; i++ )
        tri_num = tri_num + i;
    printf("O número triangular %i é %i", n, tri_num);
}
main() {
    numero_triangular (10);
    numero_triangular (50);
}
```

► **Parâmetro do tipo inteiro**



# Declaração do Protótipo da Função

**void numero\_triangular (int n)**

- Uma vez definido o nome do parâmetro na função, podemos se referir a ele em qualquer lugar no corpo da função.

Ex:

```
void numero_triangular (int n) {  
    int i, tri_num = 0;  
    for (i = 1; i <= n; i++)  
        tri_num = tri_num + i;  
    printf("O número triangular %i é %i", n, tri_num);  
}
```

- **Parâmetros atuais (Argumentos)** – são os valores passados através dos parâmetros. Eles são passados ao se chamar a função.

Ex: x = numero\_triangular(**50**);

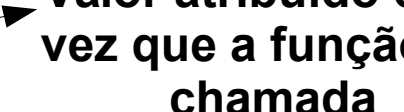
# Variáveis Locais

- São variáveis definidas dentro de cada função
- Essas variáveis existem apenas dentro da função
- O valor de uma variável local pode ser acessado somente pela função em que a variável foi definida
- Se um valor for inicial for atribuído uma variável local, esse valor será atribuído a variável cada vez que a função for chamada

Ex:

```
void numero_triangular (int n) {  
    int i, tri_num = 0;  
    for (i = 1; i <= n; i++)  
        - tri_num = tri_num + i;  
    printf("O número triangular %i é %i", n, tri_num);  
}
```

**Valor atribuído cada vez que a função for chamada**



- O escopo de uma variável local é a função em que ela é definida

# Exercício

- Considerando o exercício que calcula as bases binária, hexadecimal e octal a partir de um número base 10. Escreva funções para a conversão do número utilizando funções para cada uma dessas bases.

# Exercício

- Escreva uma função para calcular o MDC entre dois números não negativos.

```
/* Calcula o MDC*/  
  
#include <stdio.h>  
  
void mdc (int a, int b) {  
    int tmp;  
  
    printf("O mdc de %i e %i é: ", a, b);  
  
    while ( b != 0 ) {  
        tmp = a % b;  
        a = b;  
        b = tmp;  
    }  
  
    printf("%i \n", a);  
}
```

# Retornando valores

- Uma função também pode retornar valores para a rotina que a chamou

Ex: o valor do mdc calculado anteriormente

**int** mdc (int a, int b)

► Tipo do retorno da função

- Para retornar um valor para uma função que a chamou, utilizamos:

**return** ( expressão );

- Ela indica que a função deve retornar o valor da expressão para a rotina que a chamou
- **void** – Esse tipo informa que a função não possui um retorno

# Exercícios

- Adapte as funções **mdc** e **número triangular** para retornar valores para uma rotina que chama estas funções.
- Faça uma função que receba um valor inteiro e positivo e calcule o seu fatorial.
- Faça uma função que recebe a idade de uma pessoa em anos, meses e dias e retorna essa idade expressa em dias.
- Faça uma função que recebe, por parâmetro, a altura e o sexo de uma pessoa e retorna o seu peso ideal. Para homens, calcular o peso ideal usando a fórmula  $\text{peso ideal} = 72.7 \times \text{alt} - 58$  e, para mulheres,  $\text{peso ideal} = 62.1 \times \text{alt} - 44.7$

# Exercícios

- Escreva um programa que solicita o total gasto por um cliente em uma loja. O programa deve então imprimir as opções de pagamento, solicitar a opção desejada e imprimir o valor total das prestações (se houverem), baseado nas seguintes opções:

1) Opção: a vista com 10% de desconto

2) Opção: em duas vezes (preço da etiqueta)

3) Opção: de 3 até 10 vezes com 3% de juros ao mês (somente para compras acima de R\$ 100,00).

O programa deve então retornar ao usuário o valor da compra de acordo com a opção informada

# Vetores e Funções

- Além de variáveis tipos comuns, também podemos passar para uma função um elemento de um vetor ou o próprio vetor
- **Passando um elemento de um vetor como argumento da função**

Ex: `result = teste ( A[i] );`

O elemento do vetor é especificado como um argumento na função

- **Passando um vetor inteiro como argumento da função**

Ex: `int A[100];`

`x(A);`

- Observe que para receber um vetor, a função precisa ser preparada para isso

Ex: `int teste (int B[100]) {`

`...`

`return(valor);`

`}`



# Exemplo

```
/* Calcula o valor mínimo em um vetor */
#include <stdio.h>
int minimo (int valores[10]) {
    int valor_min, i;
    valor_min = valores[0];
    for (i = 1; i < 10; i++ )
        if ( valores[i] < valor_min )
            valor_min = valores[i];
    return (valor_min);
}
main() {
    int A[10], i, min;
    int minimo (int valores[10]);
    printf("Inf. 10 valores \n");
    for ( i=0; i < 10; i++)
        scanf("%i", &A[i]);
    min = minimo(A);
    printf("O valor mínimo no vetor é %i \n", min);
}
```

—————► Declaração da função

# Exercício

- Modifique o programa anterior para que a matriz seja informada diretamente no programa e para informar na função também o número de elementos do vetor.
- Escreva um programa para ler um conjunto de 10 valores inteiros e verificar se algum dos valores é igual a média dos mesmos.
- Escreva um programa onde o usuário informar um vetor de 10 elementos. Em uma função, o programa deve então copiar os elementos desse vetor para um segundo vetor. Cada elemento do segundo vetor deve ser multiplicado por dois e armazenado no próprio vetor. O programa deve imprimir os dois vetores.

# Observação

```
/* multiplica por 2*/
#include <stdio.h>

void mult_por_dois (float A[], int n) {
    int i;
    for ( i=0; i < n; ++i)
        A[i] *= 2;      —————> A[i] = A[i] * 2;
}

main() {
    float float_val[4] = { 1.2, -5.2, 7.1, 4.75 };
    int i;
    void mult_por_dois (float A[], int n);
    mult_por_dois (float_val, 4);
    for ( i=0; i < 4; ++i )
        printf("%.2", float_val[i]);
    printf("\n");
}
```

A função  
**mult\_por\_dois** muda  
os valores do vetor !!!!

**Se a função alterar o  
valor de um elemento  
do vetor, essa  
mudança implica  
diretamente no vetor  
que foi passado à  
função**

# Observações

- Na passagem de parâmetros por valor, os valores que são passados como argumento são copiados nos parâmetros
- No caso do vetor, não ocorre a cópia
- É passado para a função a informação sobre a localização de memória do vetor
- Assim, qualquer modificação no vetor é realizada no vetor original
- Isso ocorre apenas se todo o vetor for passado.
- Se apenas um elemento do vetor for passado, a cópia ainda ocorrerá

# Variáveis Globais

- **Variáveis Locais**

- Acessível somente dentro da função em que foram definidas

- **Variáveis Globais**

- Podem ser acessadas por qualquer função no programa
- Ela não pertence a uma função específica

- Ex. de definição de variáveis globais em C

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

```
int a;  
int b = 50;  
int vet[40];
```

# Exercício

- Escreva um programa em que o usuário informe um vetor de inteiro e um número. O usuário então deve informar um número qualquer e o programa deve realizar uma busca no vetor a fim de encontrar esse número.
- Faça um programa em que o usuário entre com um número positivo e uma base. O programa deve então converter o número para a base escolhida.

# Exercício

- Uma equação quadrada (2o grau) tem a forma:

$$ax^2 + bx + c = 0$$

os valores a,b,c são contantes. O valor de x que satisfaz uma equação quadrada, conhecido como raíz da equação, pode ser calculado substituindo os valores a,b,c na seguintes fórmulas:

$$X_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad X_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

Escreva um programa que resolva a equação. O usuário deve informar os valores a,b,c. Se o discriminante ( $D = b^2 - 4ac$ ) for menor que 0, deve ser informado ao usuário que não existe solução real. Caso contrário, deve ser apresentado as raízes ao usuário. Obs: Separar os cálculos utilizando funções.

# Registros (Structures)



# Estruturas

- **Vetores**

Agrupamento de elementos de um mesmo tipo

- ***Registros (Estruturas)***

Podem agrupar vários elementos de tipos diferentes

# Estruturas

- Como armazenar uma data em um programa ? Ex: 21/10/2013

Método simples - três variáveis: dia, mês, ano

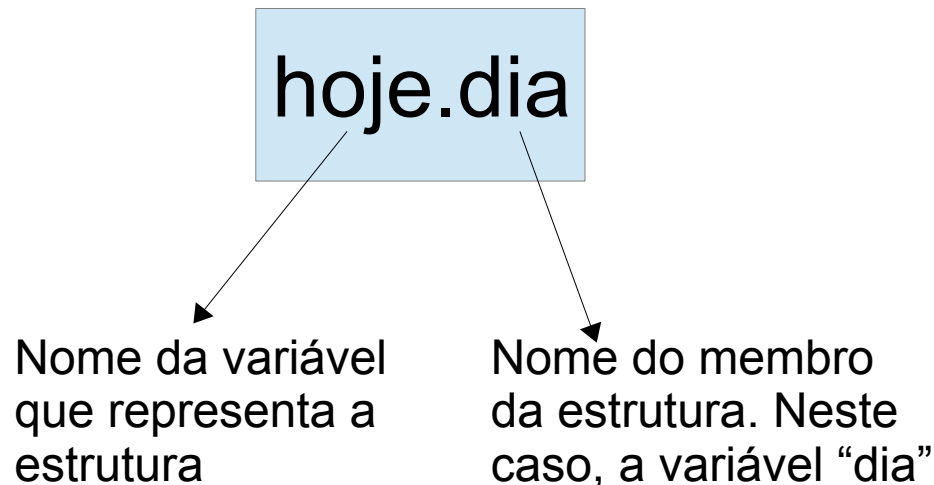
- Ao invés manter as três variáveis, poderíamos agrupá-las em uma estrutura contendo três inteiros:

```
struct data {  
    int dia;  
    int mes;  
    int ano;  
};
```

- Uma estrutura define um novo tipo, neste caso, chamado “data”
- Variáveis podem ser criadas utilizando esse novo tipo “data”.  
Ex: struct data hoje, amanha;

# Estruturas

- Uma estrutura utiliza uma sintax especial para acesso a suas variáveis
- Uma variável membro da estrutura é acessada através do nome da variável da estrutura, o ponto “.” e o nome da variável membro da estrutura. Ex:



```
hoje.dia = 15;  
hoje.mes = 10;  
hoje.ano = 2013;
```

# Funções e Estruturas

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

```
struct data {
```

```
    int dia;
```

```
    int mes;
```

```
    int ano;
```

```
};
```

```
int teste (struct data d ) {
```

```
    if ( d.dia == 2 )
```

```
        return(1);
```

```
}
```

```
main() {
```

```
    struct data hoje;
```

```
    int flag;
```

```
    hoje.dia = 2;
```

```
    hoje.mes = 3;
```

```
    hoje.ano = 2013;
```

```
    flag = teste(hoje);
```

```
    if ( flag != 1 )
```

```
        printf("Erro !\n");
```

```
}
```

## IMPORTANTE:

Qualquer alteração na função relativa aos valores da estrutura de seu argumento não altera os valores originais da estrutura. Ou seja, a função recebe uma cópia da estrutura.

d.dia = 5;

printf("%i\n\n", hoje.dia);

# Vetores de Estruturas

- **Definindo um vetor de registros**

struct data X[10]; ← define um vetor X contendo 10  
elementos referentes a estrutura

- **Atribuindo valores a um elemento do vetor**

X[1].dia = 2;

X[1].mes = 10;

X[1].ano = 2013;

# Definindo estruturas com o 'typedef'

- Podemos utilizar o comando typedef para atribuir um nome alternativo para estrutura durante sua definição. Ex:

```
typedef struct DATA1 {  
    int mes;  
    int dia;  
    int ano;  
} DATA;
```

- Dessa forma, podemos declarar a variável da estrutura da seguinte forma:

```
DATA hoje;
```

# Exercícios

- Escreva uma função que receba o tempo medido em minutos e retorne o valor correspondente em horas e em minutos. Utilize uma estrutura para armazenar as horas/minutos.
- Defina uma estrutura para armazenar os seguintes dados de um livro: nome, autor, isbn e preço. O programa deve solicitar ao usuário o cadastramento de 10 livros. Em seguida, o programa deve solicitar ao usuário um número de ISBN e o programa deve realizar uma pesquisa nos valores cadastrado. Caso o ISBN seja encontrado, o programa deve apresentar ao usuários o outros dados do livro.

# Ponteiros em C



# Ponteiros

- Um ponteiro é uma variável que mantém o endereço de memória de outra variável
- Ele fornece uma forma indireta de acesso ao valor de um dado particular
- Um ponteiro pode ser **dereferenciado**, ou seja, ele pode obter o valor que está armazenado na posição de memória para onde ele aponta.

- **Operadores importantes:**

**&** → **operador (unário) de endereço** – retorna o endereço de memória de uma variável.

Utilizado para construir um ponteiro para um objeto. Ele atribui para a 'variável de ponteiro' o endereço para a variável desejada.

**Ex:** Se **x** é uma variável do tipo **int** e **ptr\_int** é um ponteiro int, **&x** apresenta o endereço de memória da variável **x** para **ptr\_int**

**\*** → **Operador de indireção (valor no endereço)**. É utilizado para declarar a variável de ponteiro e para **dereferenciar o ponteiro**.

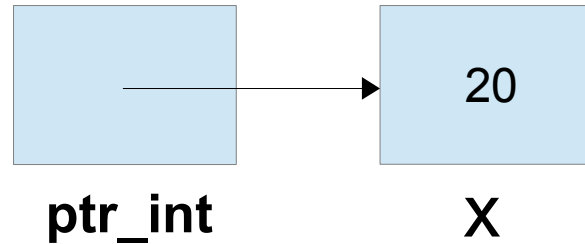
# Ponteiros

- **Ex:**

```
int x = 20, y;
```

```
int *ptr_int;
```

```
ptr_int = &x;
```



**&x** pode ser atribuído para qualquer variável de ponteiro do mesmo tipo

O ponteiro ( `*ptr_int` ) para um inteiro indica o acesso para o valor de uma ou mais variáveis inteiras.

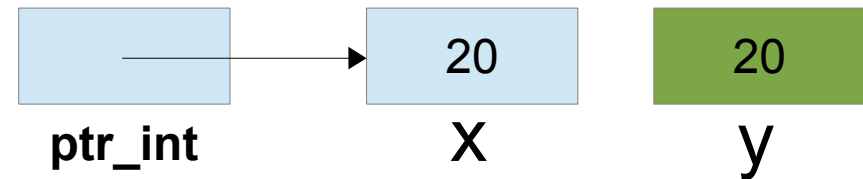
# Ponteiros

- Ponteiros não inicializados mantêm endereços de memória aleatórios !!!
  - **Boa prática**: iniciar o ponteiro com NULL ao declará-lo ←
- Ponteiros podem ser atribuídos ao: valor **0** (ou NULL), end.de memória, ou ao valor de um outro ptr. Ex.: `int *ptr1 = 0;`
- Dereferenciando o conteúdo com o **operador de indireção \***

## Exemplo 1:

```
#include <stdio.h>
main() {
    char c = 'U';
    char *pont_char = &c;
    printf("%c - %c \n", c , *pont_char);
    c = 'F';
    printf("%c - %c \n", c , *pont_char);
    *pont_char = 'P';
    printf("%c - %c \n", c , *pont_char);
}
```

Ex: `y = *ptr_int;`



Atribui a variável 'y' o valor  
que é indiretamente  
referenciado pelo ponteiro  
'ptr\_int'

O que o programa ao lado  
apresenta ao usuário ?

O ponteiro fica sem valor até ser  
apontado para algo.

# Ponteiros

- Ponteiros do tipo '**void**' podem manter ponteiros de qualquer tipo
- Apresentando o endereço de memória mantido por um ponteiro

Ex:

```
int *p1 = NULL;
```

```
int a = 70;
```

```
p1 = &a;
```

```
printf("O end. do ponteiro P1 = %p \n", (void *) p1);
```

**%p – identificador de ponteiro**

- **Apresentando o endereço de memória de uma variável**

Ex: `printf("O end. da variável 'a' = %p \n", (void *) &a);`

# Ponteiros

## Exemplo 2:

```
#include <stdio.h>
#include <stdlib.h>
main() {
    int i1=NULL, i2=NULL, *p1=NULL, *p2=NULL;
    i1 = 5;
    p1 = &i1;
    i2 = *p1 / 2 + 10; ←
    p2 = p1;

    printf("i1 = %i, i2 = %i, *p1 = %i, *p2 = %i \n",
           i1, i2, *p1, *p2);
}
```

O que esse programa apresenta ao usuário ?

# Funções e Ponteiros

(passando valores por referência para uma função)

- **Podemos passar ponteiros como argumentos de uma função**

- O ponteiro é incluído na lista de argumentos da função

Ex: calcular( lista\_de\_ponteiros )

- Dentro da função, o parâmetro deve ser declarado como ponteiro

Ex: void calcular(struct dados \*ptr)

- O parâmetro da função pode ser então utilizado como uma variável de ponteiro normal

- **IMPORTANTE:** Quando a função é chamada, o valor do ponteiro é copiado dentro parâmetro da função. Assim, qualquer mudança do parâmetro na função, não afetará o ponteiro passado para a função. Entretanto, embora o ponteiro não pode mudado pela função, os dados referenciados pelo ponteiro podem ser alterados.

- Podemos passar um ponteiro para uma variável com um argumento para a função se for necessário mudar o valor da variável de dentro da função.

## Exemplo:

```
#include <stdio.h>

void teste (int *ptr_int) {
    *ptr_int = 100;
}

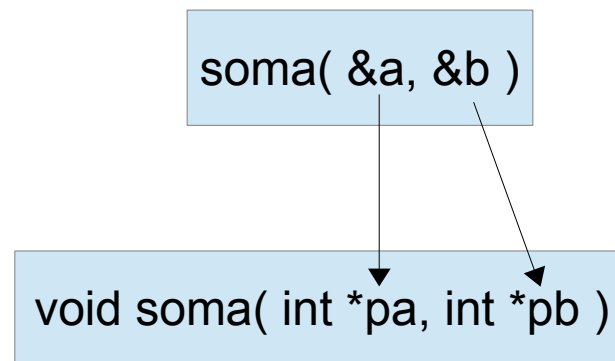
main() {
    int i = 50, *p = &i;
    printf("i antes = %i\n", i);
    teste(p);
    printf("i depois = %i\n", i);
}
```

Variável 'i' antes = 50

Variável 'i' depois = 100

# Funções e Ponteiros

- Passando valores por referência para uma função
  - Para isso, vamos utilizar o operador de endereço &



- Qualquer alteração de valores utilizando '\*pa' e '\*pb' dentro da função soma, modifica o valor das variáveis 'a' e 'b'

# Exemplo

```
#include <stdio.h>

void trocar( int *p1_int, int *p2_int ) {
    int tmp;
    tmp = *p1_int;
    *p1_int = *p2_int;
    *p2_int = tmp;
}

main() {
    int i1 = 10, i2 = 30, *p1 = &i1, *p2 = &i2;
    printf("i1 = %i, i2 = %i \n", i1, i2);

    troca( p1, p2 );
    printf("i1 = %i, i2 = %i \n", i1, i2);

    troca( &i1, &i2 );
    printf("i1 = %i, i2 = %i \n", i1, i2);
}
```



# Exercício

- Escreva uma função que recebe como argumentos duas variáveis inteiras por referência. A função deve inverter os valores das variáveis.

# Ponteiros e Estruturas

- **Definindo um ponteiro para uma estrutura**

*struct nome\_estrutura \*ponteiro\_estrutura;*

Ex: struct data \*pt\_data;

- Assim como 'ligamos' um ponteiro a uma variável, podemos 'ligar' um ponteiro de estrutura para uma estrutura

*ponteiro\_estrutura = &variavel\_estrutura*

Ex: pt\_data = &hoje;

- **Acessando a estrutura através do ponteiros**

*(\*ponteiro\_estrutura).variavel = dado;*

Ex: (\*pt\_data).dia = 5;

- **Obs 1:** Os parênteses são necessários devido o operador '.' ter uma precedência maior que o operador '\*'

**Obs 2:** O operador de ponteiro para estrutura '->' substitui os operadores '\*' e '.'

Ex: pt\_data->dia = 5;

# Estruturas contendo Ponteiros

- Muito utilizada nas estruturas de dados que veremos adiante
- Um ponteiro pode ser declarado dentro de uma estrutura.

Ex:

```
struct teste_ponteiro {  
    int *ptr1;  
    int *ptr2;  
}
```

- A variável para acesso a estrutura é definida normalmente.

Ex:

```
struct teste_ponteiro abc;
```

# Exemplo

```
#include <stdio.h>

main() {
    struct teste_ponteiro {
        int *ptr1;
        int *ptr2;
    };
    struct teste_ponteiro teste;

    int i1 = 500, i2;
    teste.ptr1 = &i1;
    teste.ptr2 = &i2;

    *teste.ptr2 = 100;

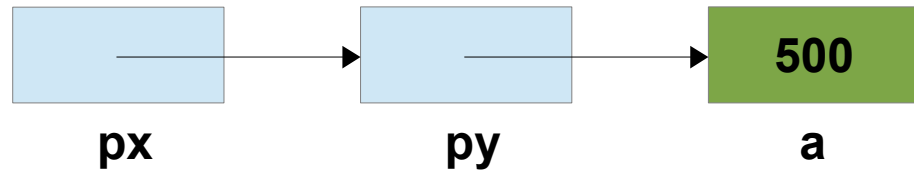
    printf("i1 = %i, *teste.p1 = %i \n", i1, *teste.ptr1);
    printf("i2 = %i, *teste.p2 = %i \n", i2, *teste.ptr2);
}
```

# Ponteiros para Ponteiros

- Um ponteiro pode manter o endereço de um outro ponteiro.

Ex:

```
int a = 500;  
int *py; int **px;  
py = &a;  
px = &py; // indireção dupla
```



- A variável **px** mantém o endereço do ponteiro **py**
- Obtém-se um endereço que, quando é dereferenciado, obtém-se um outro endereço.
- No exemplo, ao se dereferenciar o ponteiro **\*\*px**, obtém-se o valor 500
- Mais informações sobre ponteiros:**  
The 5-minute Guide to C Pointers  
<http://denniskubes.com/2012/08/16/the-5-minute-guide-to-c-pointers/>

# Alocação Dinâmica de Memória

- Diferente dos vetores que possuem um tamanho fixo e são definidos em tempo de compilação, a lista encadeada pode crescer, ou seja, novos nós podem ser adicionados a lista.
- A cada novo nó, precisamos alocar memória para armazená-lo e ligá-lo a lista. Tal alocação é realizada em tempo de execução
- Dessa forma, os NOs são criados dinamicamente, i.e. de acordo com a necessidade e enquanto exista memória.
- Através da alocação dinâmica podemos manter na memória somente o necessário, sem desperdiçar espaço.

# Alocação Estática x Dinâmica

- **Alocação Estática**

Possui tamanho fixo de memória alocada

Esse tamanho é alocado em tempo de compilação

Ex: Alocação de tipos de dados comuns como: int, float, vetor de 20 elementos inteiros

- **Alocação Dinâmica**

Iniciam vazias e crescem de acordo com a necessidade de memória

Ex: Estrutura de dados que armazena os componentes de uma expressão matemática que varia de acordo com a expressão

# Alocação Estática x dinâmica

- Exemplo de alocação estática

```
int adicao (int a, int b) {  
  
    int i, soma;  
  
    soma = 0;  
  
    for (i=a; i<b ; i++) {  
  
        soma = soma + i;  
  
    }  
  
    return(soma);  
  
}
```

- Alocação dinâmica

A alocação dinâmica em geral envolve a alocação de um bloco de armazenamento chamado aqui de **nodo**

```
typedef structure nodo  
{  
  
    char nome[20];  
    int idade;  
    float peso;  
    struct nodo *lk;  
  
} funcionario;
```

\*lk – é uma variável de ponteiro  
Utilizado para conectar nodos

Estrutura de dados dinâmicas em C  
em geral utilizam ponteiros



# Alocando Memória Dinamicamente em C

- Para alocar memória dinamicamente, utilizaremos a função **malloc()** contida na **biblioteca <stdlib.h>**
- Esta função recebe como argumento o número de bytes requeridos e retorna um ponteiro (endereço) para o novo bloco de bytes.
- **malloc()** retorna um ponteiro NULL caso a memória requerida não esteja disponível
- Para definir o número de bytes necessário a função **malloc()**, utilizaremos a função **sizeof()**
- Se precisamos alocar memória para um nó, precisamos passar o tamanho da estrutura para o **malloc()** e definir um ponteiro para estrutura para retorno do **malloc()**. Ex:

**NO \*ptr;**

**ptr = malloc( sizeof(NO) );**