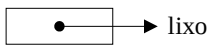
Ponteiros

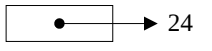
O ponteiro nada mais é do que uma variável que guarda o endereço de uma outra variável. A declaração de ponteiros é feita da seguinte forma:

```
int *pa;
```



A instrução acima indica que pa é um ponteiro do tipo int. Agora veremos como atribuir valor ao ponteiro declarado. Para isto é necessário saber que existem dois operadores unitários que são utilizados com os ponteiros. O primeiro é o operador (*), por meio dele é possível retornar o valor da variável que está localizada no ponteiro. E o segundo é o operador (&) que retorna o endereço de memória que está localizado o valor da variável contida no ponteiro. Portanto, para atribuirmos um valor para o ponteiro é necessário referenciar o valor da variável que se encontra no ponteiro utilizando o operador (*), como será definido abaixo.

```
*pa = 24;
```



Desta forma estamos atribuindo o valor 24 para a variável que está contida no ponteiro. Para entender melhor quando e como utilizar os operadores (*) e (&), veja o programa mostrado abaixo.

```
#include<stdio.h>
#include<conio.h> /* para utilizarmos getch() no final do programa,
                     função vista nos dois últimos artigos. */

void main()
{
 int a, b;
 int *pa, *pb; /* declarando ponteiros para as variáveis a e b */

 /* atribuindo valores para as variáveis a e b: */
 a=24;
 b=12;

 /* Fazendo os ponteiros pa e pb apontarem para os endereços de memória das v
 variaveis a e b respectivamente. */
 pa=&a;
 pb=&b;

 /* Imprimindo os dados nas telas, referentes aos ponteiros pa e pb declarados: */
 printf("\nImprimindo o endereço do ponteiro pa: %u", pa);
 printf("\nImprimindo o endereço da variavel contida no ponteiro pa: %u", &pa);
 printf("\nImprimindo o valor da variavel contida no ponteiro pa: %d", *pa);

 printf("\n\nImprimindo o endereço do ponteiro pb: %u", pb);
 printf("\nImprimindo o endereço da variavel contida no ponteiro pb: %u", &pb);
 printf("\nImprimindo o valor da variavel contida no ponteiro pb: %d", *pb);

 getch();
}
```

Quando os ponteiros são declarados, eles são inicializados com um endereço não válido, portanto antes de usá-los é necessário atribuir um endereço e isso é feito por meio do operador (&) como demonstra a instrução pa=&a e pb=&b que atribui aos ponteiros pa e pb o endereço das variáveis a e b.

Para um ponteiro pode ser atribuído também um endereço nulo (NULL), por exemplo:

```
pa = NULL;
```

Caso haja a necessidade de imprimir o endereço do próprio ponteiro isto é feito referenciando pa normalmente. Porém para imprimir o endereço contido no ponteiro é usado &pa e por último para imprimir o valor do endereço contido no ponteiro utiliza-se *pa.

Alocação Dinâmica

Alocação Dinâmica é o processo de solicitar e utilizar memória durante a execução de um programa. Ela é utilizada para que um programa em C/C++ utilize apenas a memória necessária pra sua execução, sem desperdícios de memória.

Um exemplo de desperdício de memória é quando um vetor de 1000 posições é declarado quando não se sabe, de fato, se as 1000 posições serão necessárias. Neste caso, por meio da alocação dinâmica, os espaços necessários para armazenar os valores seriam alocados dinamicamente, durante a execução do programa, e conforme a necessidade de armazenamento.

Sendo assim, a alocação dinâmica de memória deve ser utilizada quando não se sabe, por algum motivo ou aplicação, quanto espaço de memória será necessário pra o armazenamento de algum ou alguns valores.

Existem funções em C/C++ próprias para fazer o "pedido" de memória. Isso porque a memória pode estar cheia, e não haver mais espaço disponível. Neste caso, um pedido de alocação será recusado.

No padrão C ANSI existem 4 funções para alocações dinâmica pertencentes a biblioteca stdlib.h. São elas `malloc()`, `calloc()`, `realloc()` e `free()`. Sendo que as mais utilizadas são as funções `malloc()` e `free()`. Além das funções mencionadas acima existem outras que não são funções padrões, por exemplo, as funções `new()` e `delete()` que podem ser utilizadas no padrão C++.

A alocação dinâmica é muito utilizada em problemas de estruturas de dados, por exemplo, listas encadeadas, pilhas, filas, deques, árvores binárias e grafos. As funções `malloc()`, `calloc()` e `new()` são responsáveis por alocar memória, a

`realloc()` por realocar a memória e por último a `free()` ou `delete()` fica responsável por liberar a memória alocada.

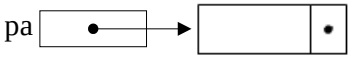
Observações importantes:

- Sempre que for feito um pedido de memória dinamicamente, deve-se verificar se o pedido foi aceito através da variável que recebeu o endereço que o `malloc()` ou `new()` retornou;
- A memória alocada dinamicamente NÃO tem nome, e deve sempre ser referenciada por uma outra variável. Ou seja, se você tem dúvida se uma variável foi alocada dinamicamente ou não, faça a simulação do programa (manual) e verifique se o local onde você está guardando uma informação possui ou não nome;
- `free()` ou `delete()` só libera memória alocada dinamicamente (memória sem nome).

Formas de uso:

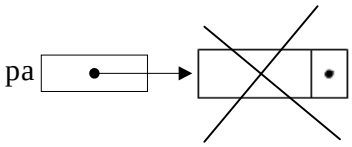
Alocando:

pa = new tipo;



Liberando:

free(pa);



A organização dos elementos alocados dinamicamente é definida por meio de ligação entre os mesmos. Esse tipo de ligação define as Listas Dinamicamente Encadeadas. Isso é possível a partir da definição de struct's vinculadas a ponteiros. Por exemplo:

```
struct TipoPonteiro
{
   char Elemento;
   TipoPonteiro *encadeamento;
};
```

Esse tipo de estrutura possibilita a formação de uma **Lista Dinâmica Simplesmente Encadeada**, que pode ser representada de forma gráfica conforme abaixo:

Cabeça da Lista