

Antes de mais nada, é importante entender a diferença entre uma variável comum, e um ponteiro. Na verdade, um ponteiro nada mais é do que um tipo especial de variável que não armazena exatamente um valor, mas sim um endereço de memória.

#### Variável:

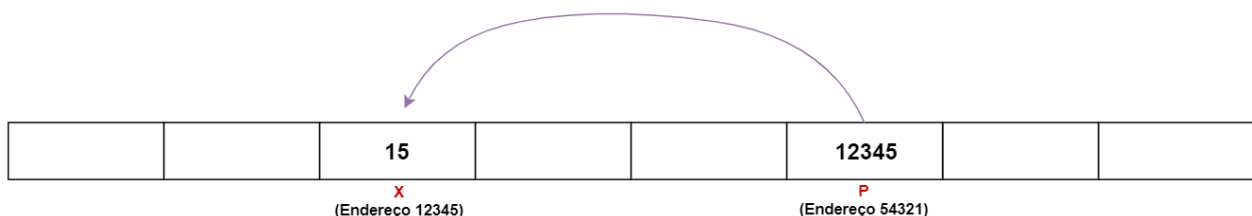
- Armazena um valor qualquer.
- Possui uma posição na memória.

#### Ponteiro:

- Armazena o endereço na memória que está "separado" para a variável
- Também possui uma posição na memória.

Quando um ponteiro **P** recebe e armazena o endereço de uma variável **X**, podemos dizer que “**P**” aponta para **X**, que **P** é o endereço de **X**. Sendo assim, **P** se referencia a **X**.

Observando a imagem abaixo, podemos ter uma noção melhor da relação entre ponteiro-variável:



Quando falamos de ponteiro, utilizamos dois operadores:

1. "\*" utilizado na declaração de um ponteiro ou para acessar o valor que ele está apontando
2. "&" utilizado para obter o endereço de uma variável.

## Pegando o endereço de uma variável

Quando utilizamos o **&**, estamos pegando o endereço de memória que está reservado para uma variável e o armazenando em outra variável.

Então se **X** é uma variável, **&X** é o seu endereço. Por isso, ao utilizarmos o **scanf**, sempre colocamos o **&**, para que o computador saiba onde armazenar o conteúdo que está lendo.

É como se disséssemos: “Vá no endereço dessa variável e armazene o conteúdo lá”

Por isso, se não colocarmos o **&**, o programa simplesmente encerrará, pois ele tentará armazenar o conteúdo na memória, porém não saberá onde.

```
1 #include <stdio.h>
2
3 int main(){
4     int var;
5
6     scanf("%d", var);
7     printf("-> %d", var);
8
9     return 0;
10 }
```

## Declaração de ponteiro

Na declaração de ponteiros, utilizamos o asterisco “\*”, e o nome que queremos dar ao nosso ponteiro; como na imagem a seguir:

```
1 #include <stdio.h>
2
3 int main(){
4     int x = 10;
5     int *p;
6
7     return 0;
8 }
```

Para passar o endereço da memória de uma variável para um ponteiro, ambos devem ser do mesmo tipo, **int**, **char**, etc.

```
1 #include <stdio.h>
2
3 int main(){
4     int x = 10;
5     int *p;
6     p = &x;
7
8     return 0;
9 }
```

Na hora da atribuição do endereço da memória de **X** para **P**, se não inserirmos o operador **&**, o programa irá apresentar um erro e não compilará.

```
1 #include <stdio.h>
2
3 int main(){
4     int x = 10;
5     int *p;
6     p = x;
7
8     return 0;
9 }
```

Line	Col	File	Message
		C:\Users\Dany\Desktop\Untitled1.cpp	In function 'int main()':
5	14	C:\Users\Dany\Desktop\Untitled1.cpp	[Error] invalid conversion from 'int' to 'int*' [-fpermissive]

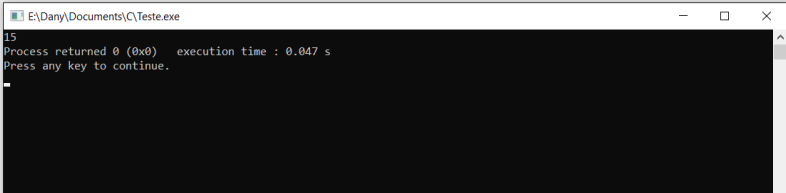
No **CodeBlocks** o programa até compilará, mas também “reclama” da mesma coisa:

File	Line	Message
		=== Build file: "no target" in "no project" (compiler: unknown) ===
E:\Dany\Docume...		In function 'main':
E:\Dany\Docume... 5		warning: initialization makes pointer from integer without a cast [-Wint-conversion]
		=== Build finished: 0 error(s), 1 warning(s) (0 minute(s), 0 second(s)) ===

Isso acontece pois, um ponteiro pode receber apenas um endereço de memória, e não um conteúdo. Quando colocamos **P = X**, estamos tentando atribuir o valor de X (que é 10) ao ponteiro.

Agora, tendo o ponteiro armazenando o endereço da variável **X**, é possível alterar o valor de **X** utilizando **P**:

```
1 #include <stdio.h>
2
3 int main(){
4     int x = 10;
5     int *p;
6
7     p = &x;
8     *p = 15;
9
10    printf("%d", *p);
11
12    return 0;
13 }
```

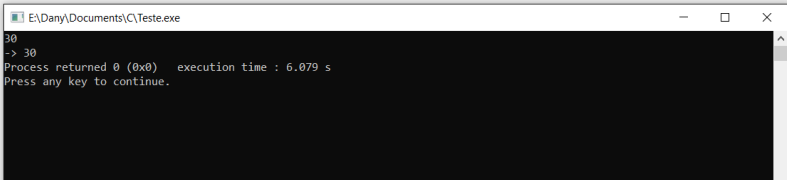


Observe que, sempre ao mexer com o conteúdo da variável **X**, utiliza-se o **\***. Por isso, para um melhor entendimento, sempre leio o **\*** como “**conteúdo apontado por**”. Nesse caso, é o conteúdo apontado por **P**.

Feito isso, é como se dissesse: “**pegue o conteúdo que o ponteiro está referenciando, e mude para 15**”

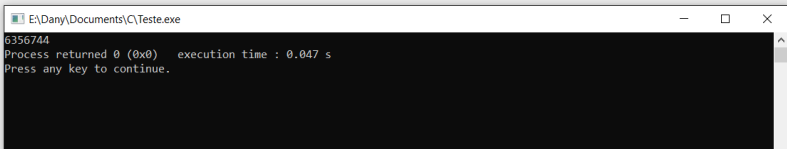
Também é possível alterar o valor que **P** está apontando ao utilizar o *scanf*. Mas nesse caso, é desnecessário o uso do **&** como de costume, pois o ponteiro já possui o endereço da memória de **X**.

```
1 #include <stdio.h>
2
3 int main(){
4     int x = 10;
5     int *p;
6
7     p = &x;
8
9     scanf("%d", p);
10    printf("-> %d", *p);
11
12    return 0;
13 }
```



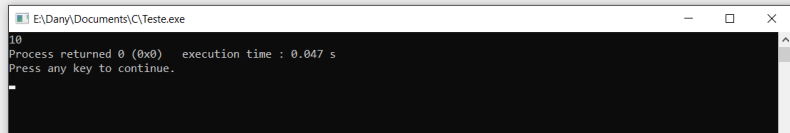
Então, como **P** guarda um endereço e não um valor, ao colocarmos para imprimir **P**, ele imprimirá o endereço da variável **X**:

```
1 #include <stdio.h>
2
3 int main(){
4     int x = 10;
5     int *p;
6
7     p = &x;
8
9     printf("%d", p);
10
11    return 0;
12 }
```



Para imprimir o conteúdo apontado por P, utilizamos o “\*”:

```
1 #include <stdio.h>
2
3 int main(){
4     int x = 10;
5     int *p;
6
7     p = &x;
8
9     printf("%d", *p);
10
11     return 0;
12 }
```

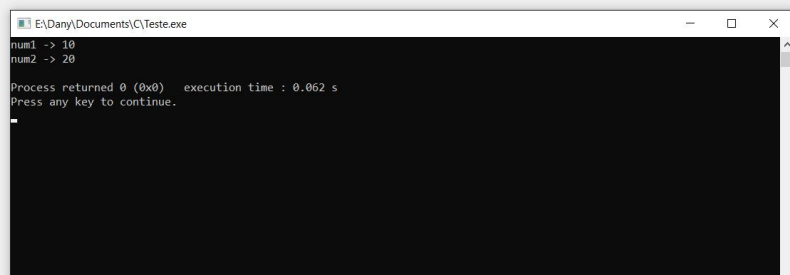


A terminal window titled "E:\Dany\Documents\C\teste.exe" showing the output of the first program. It displays the number 10, followed by "Process returned 0 (0x0) execution time : 0.047 s" and "Press any key to continue."

## Utilizando ponteiro em funções

Suponhamos que é preciso fazer uma função que troque os valores de duas variáveis:

```
1 #include <stdio.h>
2
3 void trocar (int n1, int n2) {
4     int aux;
5
6     aux = n1;
7     n1 = n2;
8     n2 = aux;
9 }
10
11 int main(){
12     int num1 = 10, num2 = 20;
13
14     trocar(num1, num2);
15
16     printf("num1 -> %d\n", num1);
17     printf("num2 -> %d\n", num2);
18
19     return 0;
20 }
```



A terminal window titled "E:\Dany\Documents\C\teste.exe" showing the output of the second program. It displays "num1 -> 10" and "num2 -> 20", followed by "Process returned 0 (0x0) execution time : 0.062 s" and "Press any key to continue."

Como observado no código acima, não alcançamos o resultado desejado, pois a função recebe apenas os valores das variáveis, e não as variáveis em si. Então, mesmo chamando essa função, as variáveis não serão alteradas.

Utilizando ponteiros, a função receberá o endereço da memória das variáveis, podendo assim, alterá-las:

```

1 #include <stdio.h>
2
3 void trocar(int *n1, int *n2) {
4     int aux;
5
6     aux = *n1;
7     *n1 = *n2;
8     *n2 = aux;
9 }
10
11 int main(){
12     int num1 = 10, num2 = 20;
13
14     trocar(&num1, &num2);
15
16     printf("num1 -> %d\n", num1);
17     printf("num2 -> %d\n", num2);
18
19     return 0;
20 }

```

Observe que, a função agora espera receber um ponteiro como parâmetro. Logo, ao chamar a função, devemos utilizar o &, para que o endereço da variável seja enviado, e não apenas seu conteúdo.

```

14 trocar(&num1, &num2);

```

Vale observar também, que a função, onde uso N1 e N2 (que são ponteiros) têm um “\*”:

```

6     aux = *n1;
7     *n1 = *n2;
8     *n2 = aux;

```

Basta lembrar o que foi exemplificado a pouco::

1. A variável AUX, recebe o conteúdo apontado por N1, que neste caso é 10.
2. O conteúdo apontado por N1 recebe o Conteúdo apontado por N2, que neste é 20.
3. O conteúdo apontado por N2 recebe o que está armazenado na variável AUX, que neste é 10.

## Ponteiro para arquivos

Quando vamos manipular arquivos, também utilizamos ponteiros, além de um tipo especial para arquivos:

```

1 #include <stdio.h>
2
3 int main(){
4     FILE *P_arq;
5 }

```

Assim, surgirá uma variável chamada “P\_arq”, que é um ponteiro para um arquivo a ser manipulado.

Caso você tente manipular um arquivo sem ponteiro, ocorrerá um erro:

```

1 #include <stdio.h>
2
3 int main(){
4     FILE P_arq;
5
6     P_arq = fopen ("P_arq.txt", "w");
7 }
8

```

File	Line	Message
=== Build file: "no target" in "no project" (compiler: unknown) ===		
E:\Dany\Docume...		In function 'main':
E:\Dany\Docume...	6	error: incompatible types when assigning to type 'FILE {aka struct _iobuf}' from type ...
=== Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===		

Na verdade, esse ponteiro é um identificador de fluxo. É através dele que vamos ler e escrever em arquivos.

## Aritmética de ponteiros

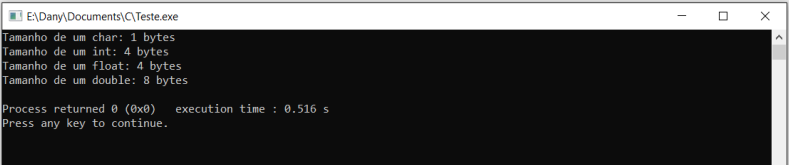
Antes de entrarmos no assunto, devemos entender mais ou menos como funciona a memória.

Cada tipo de dado possui um tamanho específico. Isso não é algo que não precisa ser decorado, podemos utilizar a função `sizeof()`.

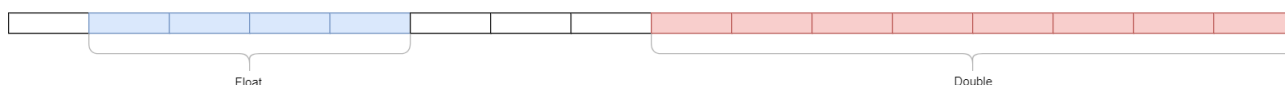
```

1 #include <stdio.h>
2
3 int main(){
4     printf("Tamanho de um char: %d bytes\n", sizeof(char));
5     printf("Tamanho de um int: %d bytes\n", sizeof(int));
6     printf("Tamanho de um float: %d bytes\n", sizeof(float));
7     printf("Tamanho de um double: %d bytes\n", sizeof(double));
8
9     return 0;
10 }
11

```



Então quando declaramos uma variável do tipo `float`, e outra do tipo `double`, na memória fica mais ou menos assim:

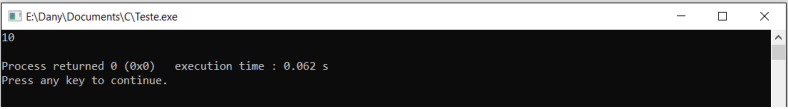


Quando falamos de aritmética de ponteiros, podemos utilizar apenas os operadores de adição e subtração. Agora criaremos um ponteiro **P**, e um vetor **VET** de inteiros, com três posições:

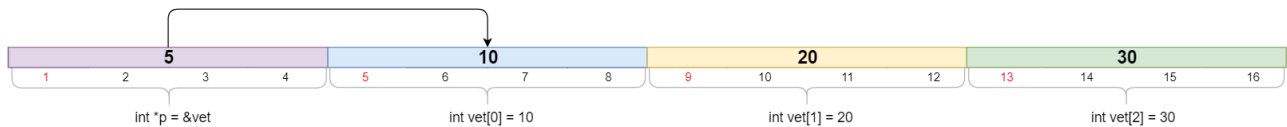
```

1 #include <stdio.h>
2
3 int main(){
4     int vet[] = {10, 20, 30};
5     int *p = &vet;
6
7     printf("%d\n", *p);
8 }

```



Na memória, de uma forma abstrata, é algo parecido com o exemplo a seguir:



O ponteiro está apontando para o endereço do primeiro item do nosso vetor (no exemplo, o endereço é o número 5).

Caso seja necessário acessar o segundo item do vetor, existem dois modos. O mais prático (e mais utilizado) é utilizando o índice. Deste modo, não é necessário declarar o ponteiro.

```

1 #include <stdio.h>
2
3 int main(){
4     int vet[] = {10, 20, 30};
5
6     printf("%d\n", vet[1]);
7
8     return 0;
9 }

```

Agora, utilizando a técnica da aritmética de ponteiros, basta apenas incrementarmos o ponteiro:

```

1 #include <stdio.h>
2
3 int main(){
4     int vet[] = {10, 20, 30};
5     int *p = &vet;
6
7     p++;
8     printf("%d\n", *p);
9
10    return 0;
11 }

```

E para ir avançando no vetor, basta ir incrementando P.

É **MUITO** importante lembrar que, uma vez incrementado, o ponteiro para de apontar o primeiro elemento, passando a apontar para o próximo.

Para voltar a apontar para o primeiro, basta fazer o contrário:

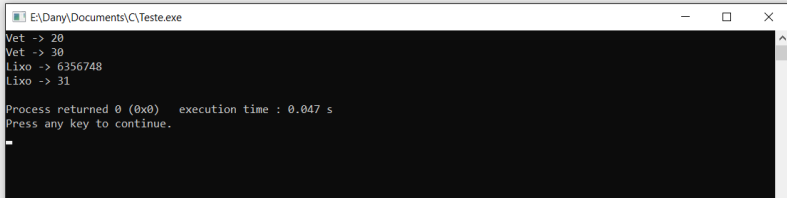
```

1 #include <stdio.h>
2
3 int main(){
4     int vet[] = {10, 20, 30};
5     int *p = &vet;
6
7     p++;
8     printf("%d\n", *p);
9
10    p--;
11    printf("-> %d\n", *p);
12
13    return 0;
14 }

```

Se continuarmos incrementando, para “além do vetor”, ele apontará para endereços que contêm lixo:

```
1 #include <stdio.h>
2
3 int main(){
4     int vet[] = {10, 20, 30};
5     int *p = &vet;
6
7     p++;
8     printf("Vet -> %d\n", *p);
9
10    p++;
11    printf("Vet -> %d\n", *p);
12
13    p++;
14    printf("Lixo -> %d\n", *p);
15
16    p++;
17    printf("Lixo -> %d\n", *p);
18
19    return 0;
20 }
```



Quando criamos um vetor, os dados ficam alinhados sequencialmente na memória, por isso, apenas incrementando o ponteiro conseguimos acessar o próximo valor. Entretanto, tendo duas variáveis **A** e **B**, e um ponteiro **P** recebendo o endereço de **A**, e **P** é incrementado, a chance de ele pegar o endereço de **B** é mínima, sendo assim, há uma pequena chance de estarem “uma do lado da outra”, tornando assim, quase impossível chegar em **B** desse modo.

```
1 #include <stdio.h>
2
3 int main(){
4     int a = 10, b = 20;
5     int *p = &a;
6
7     printf("A -> %d\n", *p);
8     p++;
9     printf("Lixo -> %d\n", *p);
10
11    return 0;
12 }
```

