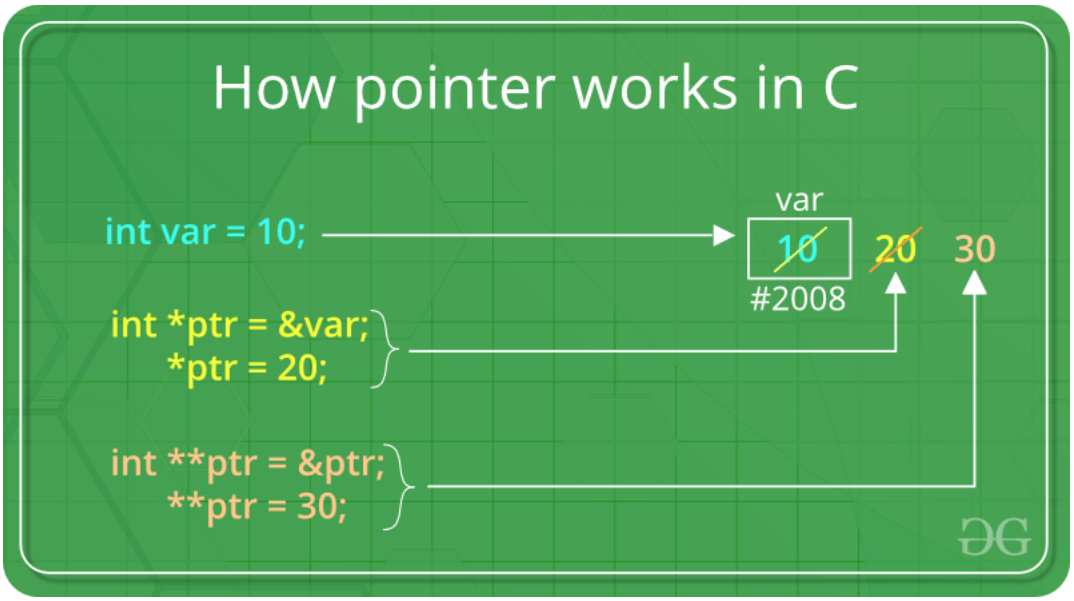
Ponteiros

Objetos (*i.e.* variáveis, funções, etc.) podem ser referidos por seus nomes, contudo em C/C++ eles contam com um identificador: seu endereço de memória. Todo objeto declarado é alocado em uma determinada posição de memória.



O que são?

São **tipos de variáveis** que têm como objetivo armazenar o endereço de outras variáveis.

Por exemplo, dado um tipo **int**, **int*** é um tipo "ponteiro para **int**". Isto é, uma variável do tipo **int*** pode armazenar um endereço de uma outra variável do tipo **int**.

Veja:

```
char c = 'a';
char* p = &c;       // & é o operador de endereço, ou seja, retorna o endereço da variável 'c', neste caso
```

> Tenha em mente que a declaração pode ser `char* c;` ou `char *c;`, o resultado é o mesmo.

Dereferenciação ou indireção

Uma operação essencial quando tratamos de ponteiros é a dereferenciação, que é se referir a um objeto que está sendo apontado pelo ponteiro. Em outras palavras, é recuperar o valor que uma variável, apontada pelo ponteiro, armazena.

Veja:

```
char c = 'a';
char* p = &c;           // & é o operador de endereço, ou seja, retorna o endereço da variável 'c', neste caso
char c2 = *p            // * é o operador de dereferenciação e retorna o valor da variável que o ponteiro 'p' aponta, ou sej
```

> Note que o ***** possui duas funções diferentes em ponteiros. A **primeira** é sua utilização na declaração de uma variável do tipo ponteiro. A **segunda** é como operador de *dereferenciação* para acessarmos o valor de uma variável apontada.

Quando declaramos um array (*vetor*):

```
int num[10];
```

é como se `num` recebesse um ponteiro para o endereço da primeira posição do vetor!

> Na realidade é feita uma conversão implícita do nome do array para o ponteiro que aponta para a primeira posição

Por que existem os ponteiros?

- Facilidade ao programador (não precisa decorar o mapeamento da memória).
- Fornece um melhor uso da memória (alocação de memória sobre demanda)...

Ponteiros de ponteiros

Um ponteiro é uma variável, armazenada em memória. Sendo assim, ele possui também um endereço que pode ser armazenado em outro ponteiro.

Não existe um limite para a composição de ponteiros de ponteiros. Para um ponteiro para uma variável "normal" se usa um *****, para um ponteiro de ponteiro que aponta para uma variável usa-se ****** , para um ponteiro de ponteiro de ponteiro que aponta para uma variável se usa ******* e por aí vai ...

Veja:

```
#include <iostream>

using namespace std;

int main() {
    int numero1 = 1;
    int numero2 = 2;
    int *p1 = &numero1;                     // ponteiro normal
    int *p2 = &numero2;                     // ponteiro normal
    int **p3 = &p1;                         // ponteiro de ponteiro
    int **pointer_array[2] = {&p1, &p2};    // vetor de ponteiros de ponteiros

    cout << **p3 << endl;
    cout << **pointer_array[0] << endl;     // imprime mesmo resultado da linha de baixo.
    cout << ***pointer_array << endl;       // lembre-se que um array age como um ponteiro
                                            // que aponta para a primeira posição.
}
```

Parâmetro por referência x Parâmetro por cópia

A passagem de parâmetros para funções utilizando ponteiros pode parecer confusa, mas não é.

Utilizando passagem de parâmetro por cópia, garantimos que o objeto passado só será modificado no escopo da função de destino e não afetará sua condição de fora.

Já utilizando passagem por referência, passamos o endereço do objeto e o mesmo será modificado em todos seus escopos.

Passagem por valor (ou cópia)

```
#include <iostream>

using namespace std;

void foo(int bar) {
    bar += 1;
    cout << bar << endl;
}

int main() {
    int sample = 10;
    foo(sample);
    cout << sample << endl;
}
```

Passagem por referência

```
#include <iostream>

using namespace std;

void foo(int *bar) {
    *bar += 1;
    cout << *bar << endl;
}

int main() {
    int sample = 10;
    foo(&sample);
    cout << sample << endl;
}
```

Alocação dinâmica

Documentação oficial

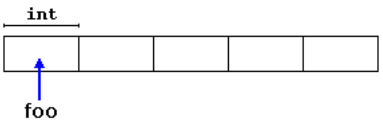
Em muitos programas, a quantidade (e a continuidade) de memória é dependente de alguma entrada do usuário. Desta forma, é necessária que esta alocação de memória seja dinâmica e ocorra por demanda (só alocar o que for necessário).

Operador *new*

Para realizar uma alocação dinâmica, utilizamos o operador **new** seguido de um tipo de dados (e se for um vetor, a quantidade de elementos desejados). Isso retornará um **ponteiro** para a primeira posição da memória alocada.

Veja:

```
int *foo;
foo = new int [5];
```



> Uma importante diferença em se alocar um vetor estaticamente e dinâmicamente é que, utilizando a primeira, o tamanho do vetor deve ser definido em tempo de design do programa, enquando na abordagem dinâmica, pode ser feita em tempo de execução.

Operador *delete*

Em muitos momentos, a memória alocada dinâmicamente não é mais necessária e pode ser desalocada. É possível realizar isso através do operador **delete**.

Veja:

```
delete foo;         // desaloca um único elemento alocado usando 'new'
delete[] bar;           // desaloca um array inteiro alocado usando 'new'
```