

## EXPLICANDO A AULA: Exercício: Funções que retornam Arrays Invertidos

### Código da aula:

<https://github.com/professormarcosp/AprendaCParaGamesUE4/blob/master/Arrays/RetornandoArrays.cpp>

Na primeira linha do código declarei esta função

```
void DuplicaVetor(int *Array, int Tam);
```

Observe que ele não retorna nada(void) e será capaz de receber dois argumentos os quais serão enviados aos parâmetros desta função que são: int \*Array e int Tam

O Parâmetro int \*Array indica que Array é um ponteiro capaz de receber o endereço de memória de uma variável que contém nela um número inteiro.

int Tam – Este vai receber um valor inteiro(int) que será o tamanho do vetor

Veja que na função main() nós declaramos um vetor de nome Numeros e já iniciamos ele com dez elementos.

```
int Numeros[] = { 1,2,3,4,5,6,7,8,9,10 };
```

Lembre que podemos omitir o valor do colchete [] pois na hora da compilação o compilador irá contar quantidade de elementos em { 1,2,3,4,5,6,7,8,9,10 } e vai colocar int Numeros[10] = { 1,2,3,4,5,6,7,8,9,10 };

```
int Tam = sizeof(Numeros) / sizeof(int);
```

Já o tamanho(Tam) como já mencionado em aulas anteriores é calculado da seguinte forma

sizeof(Numeros) irá retornar o número de bytes total do vetor Numeros. E como são 10 inteiros e assumindo na arquitetura do computador cada inteiro tenha 4 bytes. sizeof(Numeros) irá retornar 10 elementos vezes 4 bytes sendo portando  $10 \times 4 = 40\text{bytes}$

já sizeof(int) irá retornar tamanho do tipo de dados int que neste caso é 4bytes

Portanto a expressão `int Tam = sizeof(Numeros) / sizeof(int)` ficará assim:

```
int Tam = 40 / 10;
```

logo `int Tam = 10`

assim a variável Tam da função main() terá valor 10(Dez) que é o número de elementos do vetor Numeros

E ()Então ao chamarmos a função DuplicaVetor na função main

```
int main()
{
    int Numeros[] = { 1,2,3,4,5,6,7,8,9,10 };
    int Tam = sizeof(Numeros) / sizeof(int);

    std::cout << "sizeof(Numeros)" << sizeof(Numeros) << " Bytes\n";
    std::cout << "sizeof(&Numeros[0])" << sizeof(&Numeros[0]) << " Bytes\n";

    //chama a função e passa o vetor Numeros para ela
    //O endereço do primeiro elemento do vetor será enviado para Array
    MostraVetor(Numeros, Tam);
}
```

```

    DuplicaVetor(Numeros, Tam);
    MostraVetor(Numeros, Tam);
    //Aqui Mostrar vetor recebe o retorno da função Inverte Vetor
    //E este retorno será o endereço do primeiro elemento do array invertido
    MostraVetor(InverteVetor(Numeros, Tam), Tam);
    system("PAUSE");
    return 0;
}

```

Ao chamarmos a função DuplicaVetor, estaremos na realidade enviado estes argumentos:

```
void DuplicaVetor(&Numero[0], 10);
```

Lembre-se e como já comentei em outras aulas que nas linguagens C++ e C o nome de um vetor é o mesmo que endereço de memória do primeiro elemento deste vetor ou &NomeDoVetor[0]

Logo ao chamar a função DuplicaVetor na função main você está na realidade enviando o argumentos desta forma.

```
DuplicaVetor(&Numeros[0], Tam);
```

Se você quiser inclusive você pode usar explicitamente &Numeros[0] ao invés de apenas o nome do vetor Numeros.

Agora após chamar a função DuplicaVetor(Numeros, Tam) os argumentos Numeros, Tam serão enviados respectivamente para os parâmetros int \*Array, int Tam desta função que tem assinatura:

```
void DuplicaVetor(int *Array, int Tam)
```

Logo a variável ponteiro de inteiro Array (int \*Array) irá receber o argumento Numeros que é na realidade &Numeros[0]. E o que isso significa?

Significa que a variável Array que é parâmetro int \*Array da função DuplicaVetor irá armazenar dentro dela o endereço de memória do primeiro elemento do vetor Numeros

Ou seja, quando você declara uma função e indica que ela terá parâmetros, quando ela for chamada estes parâmetros são variáveis locais a esta função e serão criados e armazenados na memória apenas quando a função estiver sendo executada. Depois que a função for finalizada estas variáveis Array e Tam que são os parâmetros da função DuplicaVetor, serão descartadas automaticamente da memória e por isso, chamados estes parâmetros de variáveis locais a função.

Uma observação importante é que apesar de Tam ter o mesmo nome da variável Tam declarada na função main() eles são diferentes, pois são recursos diferentes e estão em endereços de memória diferentes! Uma é uma variável local da função main e a outro é uma variável local da função DuplicaVetor

Após enviar os argumentos para os parâmetros é hora de executar o código da função

```

void DuplicaVetor(int* Array, int Tam)
{
    //função percorre o vetor e duplica os valores
    //do vetor
    for (int i = 0; i < Tam; i++)
    {
        //Array[i] = 2 * Array[i];
        *(Array + i) = 2 * *(Array + i);
    }
}

```

Então o vamos supor para fins de aprendizado e simplificação que o parâmetro Array ou a variável declarada na função `int* Array` recebe o valor do endereço de memória 100

Ou seja, estou simplificando e imaginando que o endereço de memória do primeiro elemento do vetor `Numeros` seja 100 ou seja `&Numeros[0]` é cem! 100

Como inteiro `int` tem 4 bytes na memória o vetor `Numeros` estaria assim:

Numeros										
Índice do vetor	0	1	2	3	4	5	6	7	8	9
Endereço	100-103	104-107	108-111	112-115	116-119	120-123	124-127	128-131	132-135	136-139
Valor	1	2	3	4	5	6	7	8	9	10

Ou seja, o endereço de memória do primeiro elemento do vetor é 100 e para ler cada elemento contamos 4 bytes pois este é um vetor de inteiros ou do tipo `int`

Logo como o compilador sabe que é um array de inteiros pois você declarou assim:

```
int Numeros[] = { 1,2,3,4,5,6,7,8,9,10 };
```

Ele saberá ler e percorrer estes array pois, a cada 4 bytes lidos você lê um elemento deste vetor

E aí temos a seguir no código o trecho do loop `for` ou repetição usando `for`

```
for (int i = 0; i < Tam; i++)
{
    //Array[i] = 2 * Array[i];
    *(Array + i) = 2 * *(Array + i);
}
```

Temos a variável `i` de controle do loop `for` e que é declarada e iniciada com o valor zero `int i = 0`

Além disso, temos a condição `i < Tam`, ou seja, como `Tam` tem valor 10 neste caso a condição de permanência no loop `for` será `i < 10`

Logo `i` vai de 0 até 9 e depois o loop é encerrado

Depois temos que incrementar `i` para ele ir para próximo elemento através de `i++`

Depois já no corpo do loop `for` entra o comando `Array[i] = 2 * Array[i];`

O que ele significa?

Lembre da aula de Aritmética de ponteiros!

Como `Array` é um ponteiro então `Array[i]` é o mesmo que `*(Array + i)`

Veja que você pode inclusive colocar desta forma no código e irá funcionar!

```

void DuplicaVetor(int* Array, int Tam)
{
    //função percorre o vetor e duplica os valores
    //do vetor
    for (int i = 0; i < Tam; i++)
    {
        //Array[i] = 2 * Array[i];
        *(Array + i) = 2 * *(Array + i);
    }
}

```

C:\Users\curso\source\repos\TesteAlunos\Debug\TesteAlunos.exe

[ 1 2 3 4 5 6 7 8 9 10 ]

[ 2 4 6 8 10 12 14 16 18 20 ]

[ 20 18 16 14 12 10 8 6 4 2 ]

Pressione qualquer tecla para continuar. . .

Logo **Array[i]** é uma simplificação, um atalho para o comando **\*(Array + i)**

C++ manteve, portanto, algumas simplificações para poupar código e digitação, o que é importante para diminuir tamanho do código para ambientes como poucos recursos de memória, etc como microcontroladores, por exemplo.

Logo **\*(Ponteiro + i)** pode ser simplificado para apenas **Ponteiro[i]**

Logo **\*(Array + i)** é o mesmo que **Array[i]**

Mas o que significa então **\*(Array + i) = 2 \* \*(Array + i);** ou **Array[i] = 2 \* Array[i];**

Vamos percorrer o loop for e ver o que está ocorrendo

Primeiro  $i = 0$  e Array recebe o valor do endereço de memória do primeiro elemento do vetor e Tam recebe o valor da Variável Tam de main enviada.

Então temos que na função DuplicaVetor a variável Array irá receber o valor do endereço do primeiro elemento do vetor Numeros (&Numeros[0]) e que assumimos que seja 100 por questões da explicação

```

//chama a função e passa o vetor Números para ela
//O endereço do primeiro elemento do vetor será enviado para
MostraVetor(Numeros, Tam);
DuplicaVetor(Numeros, Tam);
MostraVetor(Numeros, Tam);
//Aqui Mostrar vetor recebe o retorno da função Inverte Vetor
//E este retorno será o endereço do primeiro elemento do arr
MostraVetor(InverteVetor(Numeros, Tam), Tam);
system("PAUSE");
return 0;
}

void DuplicaVetor(int* Array, int Tam)
{
    //função percorre o vetor e duplica os valores
    //do vetor
    for (int i = 0; i < Tam; i++)

```

Logo Array, que é um ponteiro de inteiros, terá armazenado dentro dele o valor de memória 100 e Tam terá o valor 10. Ambos valores passados via argumentos na chamada da função `DuplicaVetor` como você observa acima na figura.

Temos na memória Ram carregado o vetor `Numeros`

Numeros										
Índice do vetor	0	1	2	3	4	5	6	7	8	9
Endereço	100-103	104-107	108-111	112-115	116-119	120-123	124-127	128-131	132-135	136-139
Valor	1	2	3	4	5	6	7	8	9	10

Então vamos executar o loop passo a passo

Para  $i = 0$ ; temos `Array[0] = 2 * Array[0]`; ou `*(Array + 0) = 2 * *(Array + 0)`;

Logo como `Array` recebeu o valor do endereço de memória de `Numeros` é isso que o compilador “enxergará”:

`*(100 + 0) = 2 * *(100 + 0)`;

ou `*(100) = 2 * *(100)`;

Lembre que o operador de desreferenciamento asterisco `*` indica que queremos o valor armazenado no endereço de memória apontado pelo ponteiro.

Logo `*(100)` irá retornar o valor 1 do vetor `Numeros`, pois 100 é o endereço de memória do primeiro elemento do vetor `Numeros`...

Mas observe que o primeiro elemento está armazenado nos endereços 100 à 103

Mas uma vez lembrando o que já comentei em aulas anteriores, o endereço de memória de uma variável é o primeiro endereço que ela está armazenada, não importando o tamanho dela. Inclusive uma variável do tipo ponteiro só armazena um único e somente um endereço de memória.

Mas como o operador de desreferenciamento `*` sabe ler variável se o ponteiro só tem armazenado dentro dele o primeiro endereço desta variável?

É aí que entra o tipo do ponteiro. Como é um ponteiro do tipo `int` o compilador sabe que são 4 bytes de tamanho. Logo o primeiro elemento será lido os endereços 100, 101, 102 e 103 totalizando 4 bytes!

Endereço	100-103	104-107	108-111
Valor	1	2	3

Então retomando a execução do loop `for` temos

Para  $i = 0$ ; temos `Array[0] = 2 * Array[0]`;

ou `*(Array + 0) = 2 * *(Array + 0)`;

Logo  $*(100 + 0) = 2 * *(100 + 0);$

ou  $*(100) = 2 * *(100);$

Assim podemos ler  $*(100) = 2 * *(100);$  da seguintes forma

$*(100) = 2 * *(100);$

Compilador atribua ao valor contido no endereço de memória 100 o resultado da multiplicação de 2 pelo valor contido no endereço de memória 100.

Assim temos

$*(100)$  deve receber  $2 * 1$  pois  $*(100)$  retorna o valor contido no endereço 100

Assim o vetor ficará assim quando  $i$  é zero

Endereço	100-103	104-107	108-111	112-115	116-119	120-123	124-127	128-131	132-135	136-139
Valor	2	2	3	4	5	6	7	8	9	10

**Seguindo para  $i = 1$  pois o for volta compara  $1 < 10$  e continua**

Para  $i = 1$ ; temos  $\text{Array}[1] = 2 * \text{Array}[1];$

ou  $*(\text{Array} + 1) = 2 * *(\text{Array} + 1);$

Logo  $*(100 + 1) = 2 * *(100 + 1);$

**IMPORTANTE DESTACAR AQUI O QUE JÁ COMENTEI NA AULA SOBRE ARITMÉTICA DE PONTEIROS**

**ESTE  $100 + 1$  NÃO DARÁ O VALOR 101!!!! ISSO É BEM IMPORTANTE!!!**

**ENDEREÇO DE MEMÓRIA  $100 + 1$  EM ARITMÉTICA DE PONTEIROS SERÁ 100 MAS O TAMANHO DO TIPO DO PONTEIRO**

**LOGO COMO INT TEM 4 BYTE ESTE  $100 + 1$  SERÁ NA REALIDADE  $100 + 4$  O QUE DARÁ ENDEREÇO 104!**

**E COM ISSO OBSERVE QUE CHAGAREMOS NO SEGUNDO ELEMENTO DO VETOR** Numeros que tem endereço 104

Lembre-se também que como o ponteiro Array tem o endereço do vetor Numeros ele consegue alterar o valor deste vetor. É isso que chamamos de passagem de argumentos para a função por referência.

Pois foi enviado a função DuplicaVetor o endereço de memória do vetor Numeros que apesar ter 40 bytes de tamanho total seu endereço é O PRIMEIRO ENDEREÇO DE MEMÓRIA DELE! E que neste caso assumimos como 100.

Índice do vetor	0	1	2	3	4	5	6	7	8	9
Endereço	100-103	104-107	108-111	112-115	116-119	120-123	124-127	128-131	132-135	136-139
Valor	1	2	3	4	5	6	7	8	9	10

Assim  $*(100 + 1) = 2 * *(100 + 1)$ ; é na realidade Logo  $*(100 + 4) = 2 * *(100 + 4)$ ;

Ou Logo  $*(104) = 2 * *(104)$ ;

Ou Logo  $*(104) = 2 * 2$ ;

Ou  $*(104) = 4$ ;

Então podemos ler novamente que o valor contido no endereço de memória 104 deverá receber o valor 4

Assim o vetor fica assim

Índice do vetor	0	1	2	3	4	5	6	7	8	9
Endereço	100-103	104-107	108-111	112-115	116-119	120-123	124-127	128-131	132-135	136-139
Valor	2	4	3	4	5	6	7	8	9	10

Então se executarmos o for inteiro com i indo de zero até nove termos o seguinte

i	i < Tam	Array[i] = 2 * Array[i]								
i = 0	0 < 10 (V)	Array[0] = 2* Array[0] ou *(Array + 0) = 2 * *(Array + 0); *(100 + 0) = 2 * *(100 + 0); *(100) = 2 * 1; aritmética desloca 4 bytes * i pois é ponteiro int *(100) = 2 // vá no endereço 100 e coloque o valor 2								
Índice do vetor	0	1	2	3	4	5	6	7	8	9
Endereço	100-103	104-107	108-111	112-115	116-119	120-123	124-127	128-131	132-135	136-139
Valor	2	2	3	4	5	6	7	8	9	10

i	i < Tam	Array[i] = 2 * Array[i]								
i = 1	1 < 10 (V)	Array[1] = 2* Array[1] ou *(Array + 1) = 2 * *(Array + 1); *(100 + 1) = 2 * *(100 + 1); // aritmética de ponteiros!!! *(104) = 2 * 2; aritmética desloca 4 bytes vezes(*) i pois é ponteiro int *(104) = 4 // vá no endereço 104 e coloque o valor 4								
Índice do vetor	0	1	2	3	4	5	6	7	8	9
Endereço	100-103	104-107	108-111	112-115	116-119	120-123	124-127	128-131	132-135	136-139
Valor	2	4	3	4	5	6	7	8	9	10

i	i < Tam	Array[i] = 2 * Array[i]								
i = 2	2 < 10 (V)	Array[2] = 2* Array[2] ou *(Array + 2) = 2 * *(Array + 2); *(100 + 2) = 2 * *(100 + 2); // aritmética de ponteiros!!! *(108) = 2 * *(108); // este 2 é duas vezes 4 bytes logo 8 bytes *(108) = 2 * 3 *(108) = 6; // vá no endereço 108 e coloque o valor 6								
Índice do vetor	0	1	2	3	4	5	6	7	8	9
Endereço	100-103	104-107	108-111	112-115	116-119	120-123	124-127	128-131	132-135	136-139
Valor	2	4	6	4	5	6	7	8	9	10

i	i < Tam	Array[i] = 2 * Array[i]								
i = 3	3 < 10 (V)	Array[3] = 2* Array[3] ou *(Array + 3) = 2 * *(Array + 3); *(100 + 3) = 2 * *(100 + 3); // aritmética de ponteiros!!! *(112) = 2 * *(112); // este 3 é três vezes 4 bytes logo 12 bytes *(112) = 2 * 4 // aritmética desloca 4 bytes * i pois é ponteiro int *(112) = 8; // vá no endereço 112 e coloque o valor 8								
Índice do vetor	0	1	2	3	4	5	6	7	8	9
Endereço	100-103	104-107	108-111	112-115	116-119	120-123	124-127	128-131	132-135	136-139
Valor	2	4	6	8	5	6	7	8	9	10



i	i < Tam	Array[i] = 2 * Array[i]								
i = 4	4 < 10 (V)	Array[4] = 2* Array[4] ou *(Array + 4) = 2 * *(Array + 4); *(100 + 4) = 2 * *(100 + 4); // aritmética de ponteiros!!! *(116) = 2 * *(116); // este 4 é quatro vezes 4 bytes logo 16 bytes *(116) = 2 * 5 // aritmética desloca 4 bytes * i pois é ponteiro int *(116) = 10; // vá no endereço 116 e coloque o valor 10								
Índice do vetor	0	1	2	3	4	5	6	7	8	9
Endereço	100-103	104-107	108-111	112-115	116-119	120-123	124-127	128-131	132-135	136-139
Valor	2	4	6	8	10	6	7	8	9	10

i	i < Tam	Array[i] = 2 * Array[i]								
i = 5	5 < 10 (V)	Array[5] = 2* Array[5] ou *(Array + 5) = 2 * *(Array + 5); *(100 + 5) = 2 * *(100 + 5); // aritmética de ponteiros!!! *(120) = 2 * *(120); // este 5 é cinco vezes 4 bytes logo 20 bytes *(120) = 2 * 6 // aritmética desloca 4 bytes * i pois é ponteiro int *(120) = 12; // vá no endereço 120 e coloque o valor 12								
Índice do vetor	0	1	2	3	4	5	6	7	8	9
Endereço	100-103	104-107	108-111	112-115	116-119	120-123	124-127	128-131	132-135	136-139
Valor	2	4	6	8	10	12	7	8	9	10

i	i < Tam	Array[i] = 2 * Array[i]								
i = 6	6 < 10 (V)	Array[6] = 2* Array[6] ou *(Array + 6) = 2 * *(Array + 6); *(100 + 6) = 2 * *(100 + 6); // aritmética de ponteiros!!! *(124) = 2 * *(124); // este 6 é seis vezes 4 bytes logo 24 bytes *(124) = 2 * 7 // aritmética desloca 4 bytes * i pois é ponteiro int *(124) = 14; // vá no endereço 124 e coloque o valor 14								
Índice do vetor	0	1	2	3	4	5	6	7	8	9
Endereço	100-103	104-107	108-111	112-115	116-119	120-123	124-127	128-131	132-135	136-139
Valor	2	4	6	8	10	12	14	8	9	10

i	i < Tam	Array[i] = 2 * Array[i]								
i = 7	7 < 10 (V)	Array[7] = 2* Array[7] ou *(Array + 7) = 2 * *(Array + 7); *(100 + 7) = 2 * *(100 + 7); // aritmética de ponteiros!!! *(128) = 2 * *(128); // este 7 é sete vezes 4 bytes logo 28 bytes *(128) = 2 * 8 // aritmética desloca 4 bytes * i pois é ponteiro int *(128) = 16; // vá no endereço 128 e coloque o valor 16								
Índice do vetor	0	1	2	3	4	5	6	7	8	9
Endereço	100-103	104-107	108-111	112-115	116-119	120-123	124-127	128-131	132-135	136-139
Valor	2	4	6	8	10	12	14	16	9	10

i	i < Tam	Array[i] = 2 * Array[i]								
i = 8	8 < 10 (V)	Array[8] = 2* Array[8] ou *(Array + 8) = 2 * *(Array + 8); *(100 + 8) = 2 * *(100 + 8); // aritmética de ponteiros!!! *(132) = 2 * *(132); // este 8 é oito vezes 4 bytes logo 32 bytes *(132) = 2 * 9 // aritmética desloca 4 bytes * i pois é ponteiro int *(132) = 18; // vá no endereço 132 e coloque o valor 18								
Índice do vetor	0	1	2	3	4	5	6	7	8	9
Endereço	100-103	104-107	108-111	112-115	116-119	120-123	124-127	128-131	132-135	136-139
Valor	2	4	6	8	10	12	14	16	18	20

i	i < Tam	Array[i] = 2 * Array[i]
i = 9	9 < 10 (V)	Array[9] = 2* Array[9] ou *(Array + 9) = 2 * *(Array + 9); *(100 + 9) = 2 * *(100 + 9); // aritmética de ponteiros!!! *(136) = 2 * *(136); // este 9 é nove vezes 4 bytes logo 36 bytes *(136) = 2 * 10 // aritmética desloca 4 bytes * i pois é ponteiro int *(136) = 20; // vá no endereço 136 e coloque o valor =20



Índice do vetor	0	1	2	3	4	5	6	7	8	9
Endereço	100-103	104-107	108-111	112-115	116-119	120-123	124-127	128-131	132-135	136-139
Valor	2	4	6	8	10	12	14	16	18	20
i = 10	10 < 10 (F)		SAI DO LOOP FOR!							

Agora vem a chamada da função MostraVetor(Numeros, Tam);

```
void MostraVetor(int* Array, int Tam)
{
    std::cout << "[ ";
    for (int i = 0; i < Tam; i++)
    {
        std::cout << Array[i] << " ";
    }
    std::cout << "]" \n\n";
}
```

Observe que a função possui os mesmos parâmetros int\* Array, int Tam

```
void MostraVetor(int* Array, int Tam)
```

Logo a variável ponteiro irá receber o endereço de memória do primeiro elemento do vetor Numeros

E novamente vamos assumir que este endereço seja 100

Então primeiro vai para tela um colchete apenas para deixar mais bonita a saída

```
std::cout << "[ ";
```

Depois começa o loop for que vai usar a aritmética de ponteiros

```
for (int i = 0; i < Tam; i++)
{
    std::cout << Array[i] << " ";
}
```

Veja que mais uma vez temos Array[i] que é o mesmo que \*(Array + i)

Lembrando que o Vetor Numeros foi alterado pela função DuplicaVetor e ficou assim:

Índice do vetor	0	1	2	3	4	5	6	7	8	9
Endereço	100-103	104-107	108-111	112-115	116-119	120-123	124-127	128-131	132-135	136-139
Valor	2	4	6	8	10	12	14	16	18	20

Assim teremos para cada valor de i o seguinte

i = 0	std::cout << Array[0] << " ";	O mesmo que std::cout << *(Array + 0) << " "; ou std::cout << *(Array) << " "; ou std::cout << *(100) << " "; logo vai para tela o valor contido no endereço 100 que é 2
i = 1	std::cout << Array[1] << " ";	O mesmo que std::cout << *(Array + 1) << " "; ou std::cout << *(Array + 1) << " "; ou std::cout << *(100 + 1) << " "; // 1 é 4 bytes aritmética de ponteiros! é como se fosse std::cout << *(100 + 4) << " "; ou std::cout << *(104) << " "; logo vai para tela o valor contido no endereço 104 que é 4
i = 2	std::cout << Array[2] << " ";	std::cout << *(Array + 2) << " "; ou std::cout << *(100 + 8) << " "; ou std::cout << *(108) << " "; logo vai para tela o valor contido no endereço 108 que é 6

E então a aritmética de ponteiros percorre todo Array colocando na tela seus 10 elementos

```
[ 2  4  6  8 10 12 14 16 18 20 ]
```

Temos então a chamada de uma função MostraVetor(InverteVetor(Numeros, Tam), Tam);

```
MostraVetor(Numeros, Tam);
DuplicaVetor(Numeros, Tam);
MostraVetor(Numeros, Tam);
//Aqui Mostrar vetor recebe o retorno da função Inverte Vetor
//E este retorno será o endereço do primeiro elemento do array invertido
MostraVetor(InverteVetor(Numeros, Tam), Tam);
system("PAUSE");
return 0;
```

Observe que uma função pode retornar valores ou não. Além disso, você pode ter chamadas de funções como argumentos na chamada de funções...

Como assim?

Veja que estamos chamando a função InverteVetor(Numeros, Tam) dentro da região destinada aos argumentos da função MostraVetor

Lembre que a função MostraVetor pode receber os seguintes argumentos:

Um ponteiro do tipo int que será colocado no ponteiro Array e um valor inteiro que será colocado em Tam.  
void MostraVetor(int\* Array, int Tam);

logo a linha de código abaixo primeiro vai chamar a função InverteVetor(Numeros, Tam) para somente depois chamar a função MostraVetor. Pois é primeiro necessário saber o valor do primeiro argumento

```
MostraVetor(InverteVetor(Numeros, Tam), Tam);
```

```
MostraVetor(int* Array, int Tam);
```

Portanto o compilador vai primeiro executar: `InverteVetor(Numeros, Tam)`

```
int* InverteVetor(int* Array, int Tam)
{
    int j = 0;
    static int ArrayInvertido[10];
    for (int i = Tam - 1; i >= 0; i--)
    {
        ArrayInvertido[j] = Array[i];
        j++;
    }
    return ArrayInvertido;
}
```

O que já chama atenção e confunde alunos pelo mundo todo é o tipo de retorno desta função

Veja que o tipo de retorno da função `InverteVetor` é do tipo `int*`

Logo podemos ler `int*` como sendo um ponteiro do tipo `int`. Então esta função `InverteVetor` deverá retornar um ponteiro do tipo `int`. Como ponteiros são variáveis especiais que armazenam dentro dele um endereço de memória esta função vai retornar um endereço de memória!

A função começa declarando uma variável auxiliar `j` que é iniciada com zero

```
int j = 0;
```

Depois declaramos um Vetor estático.

Mas por que este `static`?

Lembre das aulas que as variáveis de uma função são locais a ela e são destruídas da memória quando a função parar sua execução. Mas nós precisaremos manter o array invertido na memória para poder colocar ele na tela. Senão tivesse `static` este `ArrayInvertido[10]` seria destruído na saída da função e não é isso que queremos. Logo indicamos que ele é uma variável estática da função `InverteVetor`. Isto é, uma variável que é preservada na memória mesmo após a função ser encerrada.

Depois disso iniciamos o loop `for`.

```
for (int i = Tam - 1; i >= 0; i--)
```

```
{
```

```
    ArrayInvertido[j] = Array[i];
```

```
    j++;
```

```
}
```

A ideia da inversão é simples.

J vai de zero até 9, enquanto i vai de 9 até 0

assim teremos dentro do for todas estas execuções

```
ArrayInvertido[0] = Array[9];
```

```
ArrayInvertido[1] = Array[8];
```

```
ArrayInvertido[2] = Array[7];
```

```
ArrayInvertido[3] = Array[6];
```

```
ArrayInvertido[4] = Array[5];
```

```
ArrayInvertido[5] = Array[4];
```

```
ArrayInvertido[6] = Array[3];
```

```
ArrayInvertido[7] = Array[2];
```

```
ArrayInvertido[8] = Array[1];
```

```
ArrayInvertido[9] = Array[0];
```

E isso vai inverter o Array.

E ai vem algo importante! O retorno da função. Este array acima é o array estático **ArrayInvertido** declarado e preenchido na função com valores invertidos que vieram do vetor **Numeros**

Veja que temos o comando **return ArrayInvertido;**

E o que este comando quer dizer?

Lembre-se pois já comentei isso, o nome de um vetor(também chamado de array) é o mesmo que o endereço de memória do primeiro elemento dele. Supondo que seja 200 o primeiro endereço desta variável vetor de nome **ArrayInvertido**

Logo este comando return está na realidade retornando o seguinte: **return &ArrayInvertido[0];**

Assim temos agora o nosso retorno e o compilador poderá executar a função **MostraVetor**

A chamada fica assim:

```
MostraVetor(InverteVetor(Numeros, Tam), Tam); // Função chamada, mas tem que resolver primeiro outra chamada de função InverteVetor(Numeros, Tam) para poder compor seus argumentos
```

Logo outra função é chamada passando como argumentos o endereço de memória do vetor **Numeros** e o Tamanho deste vetor - **InverteVetor(Numeros, Tam)**

Esta função **InverteVetor** constrói outro vetor de nome **ArrayInvertido** com os valores invertidos do vetor **Numeros**

ArrayInvertido										
Índice do vetor	0	1	2	3	4	5	6	7	8	9
Endereço	200-203	204-207	208-211	212-215	216-219	220-223	224-227	228-231	232-235	236-239
Valor	20	18	16	14	12	10	8	6	4	2

E ai ela retorna o valor do endereço de memória do primeiro elemento do vetor

**return ArrayInvertido; que significa return &ArrayInvertido[0];**

este valor retornado vai compor o argumento da função MostraVetor

MostraVetor(InverteVetor(Numeros, Tam), Tam);


Vai ficar assim

MostraVetor(&ArrayInvertido[0], Tam);

Com isso a função MostraVetor irá ser chamada passando estes argumentos acima:

**&ArrayInvertido[0], Tam**

MostraVetor(&ArrayInvertido[0], Tam);



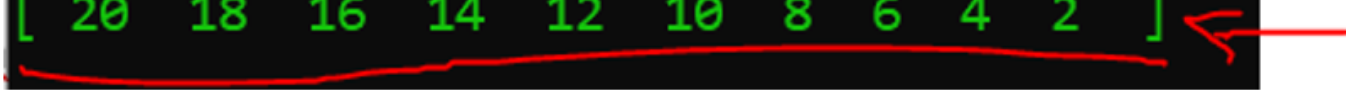
```

void MostraVetor(int* Array, int Tam)
{
    //coloca colchete antes do for
    std::cout << "[ ";
    //Coloca os elementos e finaliza o loop
    for (int i = 0; i < Tam; i++)
    {
        std::cout << Array[i] << " ";
    }
    //e depois coloca ] para fechar o vetor
    std::cout << "]" \n\n";
}

```

Desta forma a função MostraVetor será capaz de percorrer via aritmética de ponteiros todos os elementos vetor **ArrayInvertido** pois seu parâmetro **int\* Array** recebeu o endereço de memória de **ArrayInvertido**. Com isso, ele irá colocar os elementos de **ArrayInvertido** na tela seguindo a aritmética de ponteiros que já expliquei amplamente acima

```
[ 1  2  3  4  5  6  7  8  9 10 ]  
[ 2  4  6  8 10 12 14 16 18 20 ]  
[ 20 18 16 14 12 10 8  6  4  2 ]
```



Ufa! Foram 12 páginas, mas acredito que esclareceu muita coisa sobre a aulas e os conceitos abordados nela!

Grande Abraço!

Professor Marcos Pacheco