Aula 11:

Ponteiros

Prof. Edvard

edvard@unifei.edu.br

Universidade Federal de Itajubá

Ponteiros

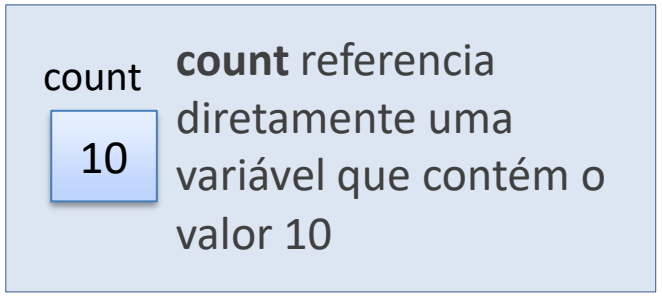
- Representam um dos recursos **mais poderosos** da linguagem C, mas estão, também, entre as capacidades **mais difíceis** de se compreender e dominar.
- São importantes especialmente em **duas situações**:
  - Permitem que um programa realize **chamadas de função por referência**, de maneira que o parâmetro possa ser acessado e modificado dentro da função.
  - Permitem a manipulação de **estruturas dinâmicas de dados**, ou seja, estruturas de dados que podem crescer ou encolher de acordo com a necessidade, em tempo de execução.
    - Listas ligadas, filas, pilhas, árvores, grafos, etc.
    - Este tópico se encontra um pouco além dos limites de nosso curso. Veremos em Estruturas de Dados!

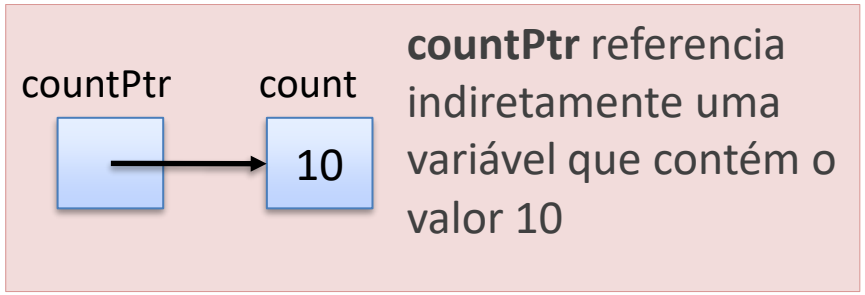
Ponteiros

- Primeiramente, é necessário reforçar um conceito: toda informação que manipulamos dentro de um programa (variável, array, estrutura, etc.) está, obrigatoriamente, armazenada na memória do computador.
  - Quando criamos uma variável, o computador reserva um **espaço na memória** onde podemos guardar o valor associado a ela. Ao nome que damos à variável, o computador associa internamente o endereço de memória do espaço que ele reservou para guardá-la.
    - Para o programador em C, somente interessa saber o nome da variável. Para o computador, é necessário saber **onde elas se encontram na memória** (seus endereços)

Ponteiros

- Mas o que, afinal, são **ponteiros**?
  - Eles são variáveis cujos **valores são endereços de memória**.
  - Às vezes (raramente) são chamados de apontadores.
  - Uma variável comum possui, normalmente, um **valor** específico. Um ponteiro, por outro lado, contém um **endereço** de uma variável que contém um valor específico.
    - Uma variável referencia o valor diretamente
    - Um ponteiro referencia o valor indiretamente.





Ponteiros

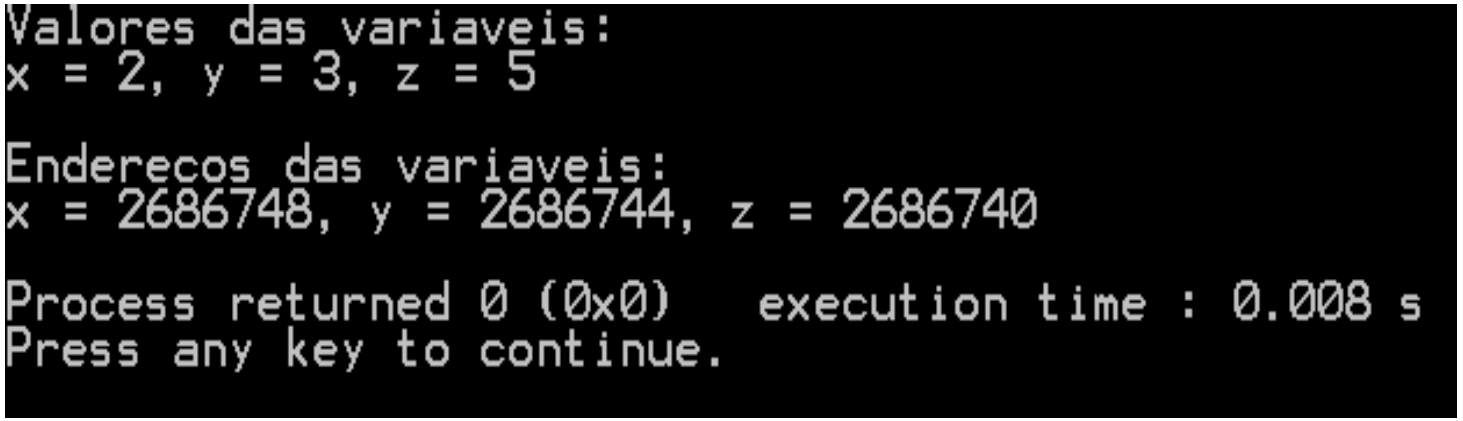
- Vamos imaginar um programa que aloca três variáveis do tipo **int**: x, y e z.
  - O compilador irá alocar **4 bytes** para cada inteiro como tamanho padrão.

```c
#include <stdio.h>

int main()
{
    int x, y, z;

    x = 2;
    y = 3;
    z = x + y;

    printf("Valores das variaveis: \n");
    printf("x = %d, y = %d, z = %d \n\n", x, y, z);
    printf("Enderecos das variaveis: \n");
    printf("x = %d, y = %d, z = %d \n", &x, &y, &z);

    return 0;
}
```

Como ficaria a saída do programa?

Ponteiros

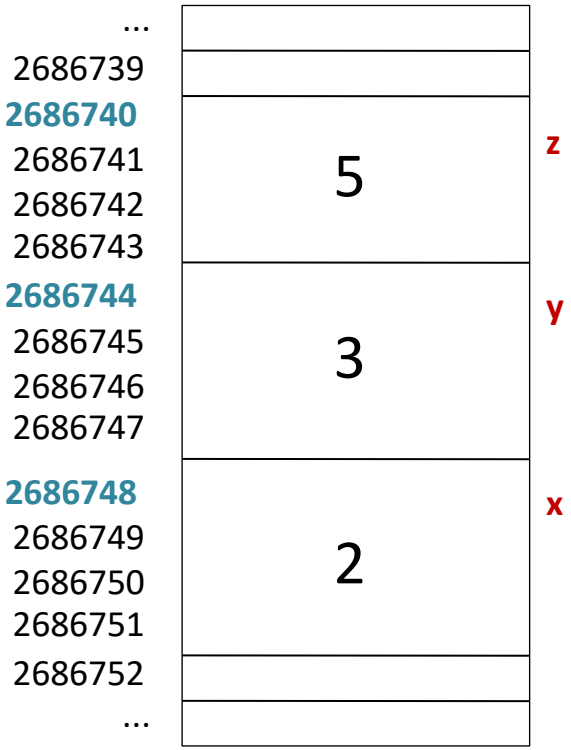
- Repare que o programa alocou as três variáveis de 4 bytes em sequência na memória, nos endereços 2686740, 2686744 e 2686748.
  - Os computadores possuem muitos endereços de memória. Imagine um computador com 4 GB de memória RAM: Cada um dos seus aproximadamente 4 bilhões de bytes possui um endereço único, diferente dos demais.



Um computador de 32 bits geralmente trata os endereços de memória como valores inteiros de 32 bits (ou 4 bytes), tornando o espaço de endereçamento igual a 2^{32} = 4.294.967.296 bytes de memória, ou 4 GB.

Ponteiros

- Na memória, os dados ficariam da seguinte maneira:



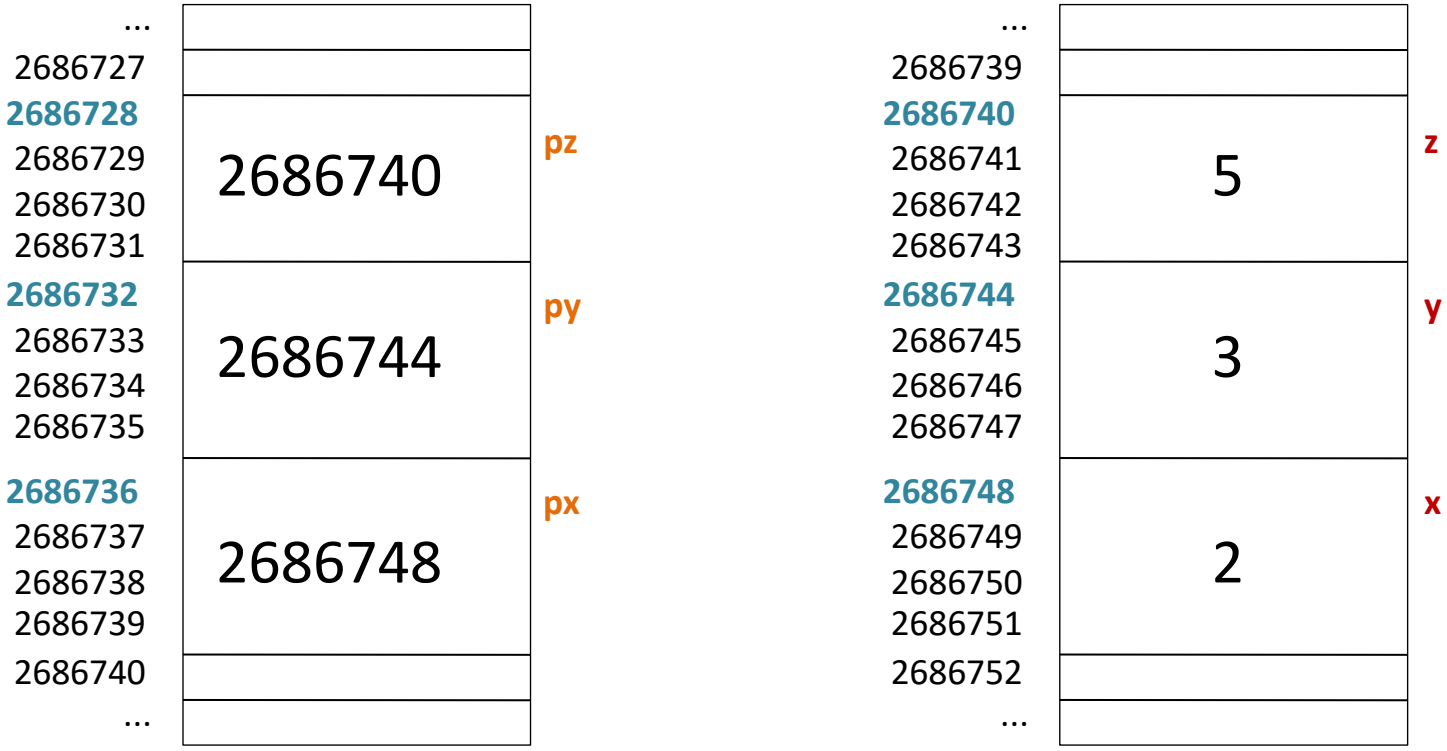
É bastante fácil, para nós programadores, alocar a memória, atribuir valores e povoar a memória do computador com nossos próprios dados. A linguagem C nos ajuda e muito, **abstraindo uma série de detalhes** e trabalhando internamente com endereços de memórias e valores.

Precisamos de 4 bytes para representar o valor 5? Como ele está representado verdadeiramente na memória?

00000000 00000000 00000000 **00000101**

Ponteiros

- Vamos agora criar, em nosso programa, três variáveis que serão ponteiros para x, y e z (px, py e pz), e, portanto, irão conter os endereços dessas variáveis e ficarão armazenados em outro local na memória, ocupando seus próprios 4 bytes.



```c
int main()
{
    int *px;
    int *py, *pz;
    int x, y, z;

    x = 2;
    y = 3;
    z = x + y;

    px = &x;
    py = &y;
    pz = &z;

    printf("Valores das variaveis: \n");
    printf("x = %d, y = %d, z = %d \n", x, y, z);
    printf("x = %d, y = %d, z = %d \n\n", *px, *py, *pz);

    printf("Enderecos das variaveis: \n");
    printf("x = %d, y = %d, z = %d \n", &x, &y, &z);
    printf("x = %d, y = %d, z = %d \n\n", px, py, pz);

    printf("Enderecos dos ponteiros: \n");
    printf("px = %d, py = %d, pz = %d \n", &px, &py, &pz);

    return 0;
}
```

```
Valores das variaveis:
x = 2, y = 3, z = 5
x = 2, y = 3, z = 5

Enderecos das variaveis:
x = 2686748, y = 2686744, z = 2686740
x = 2686748, y = 2686744, z = 2686740

Enderecos dos ponteiros:
px = 2686736, py = 2686732, pz = 2686728
```

Ponteiros

- Realizamos uma série de operações em ponteiros no último programa. Vamos, agora, formalizar sua sintaxe.
- **Declaração:**
  - Segue a seguinte forma geral:

    tipo_ponteiro *nome_ponteiro;
  - É o operador asterisco (*) que informa ao compilador que aquela variável não irá guardar um valor, mas um endereço de memória para o tipo especificado.
    - Seu significado depende de como é utilizado no programa. Pode também indicar uma operação de multiplicação ou de-referência (como vimos na aula 2 de funções).

Ponteiros

- **Declaração:**
  - Na linguagem C, quando declaramos um ponteiro, informamos para que tipo de variável poderemos apontá-lo. Um ponteiro do tipo int* **somente pode** guardar o endereço de uma variável inteira (tipo int).
    - **Observação:** Incluir as letras `ptr` antes do nome de um ponteiro contribui para a clareza do programa.
  - Outros **exemplos** de declaração:

```
int *c_ptr;
char *nome;
double *ptr_a, *ptr_b; // Asterisco precisa estar em todas as variaveis tipo ptr
float *ptr_c, d; // ptr_c é ponteiro, d não.
```

Ponteiros

- **Inicialização e atribuição:**
  - Os ponteiros devem ser inicializados quando são **declarados** ou, posteriormente, através de uma **operação de atribuição**.
  - Podem ser inicializados com **NULL**, **0**, ou um **endereço** de uma variável.
    - Um ponteiro NULL não aponta para nada. NULL é uma constante definida na biblioteca <stddef.h> e em vários outras, como a própria <stdio.h> e <stdlib.h>. Seu valor é 0 (zero).
    - Inicializá-lo com 0 é equivalente à inicialização com NULL.
  - Sempre inicialize um ponteiro, para evitar resultados inesperados no seu programa. Um ponteiro não inicializado pode possuir qualquer valor (lixo de memória) e, portanto, aponta para qualquer lugar.

```
int *ptr_x = NULL;
double *ptr_abc, *ptr_valor;

ptr_abc = 0; // NULL == 0
ptr_valor = NULL; // NULL == 0
```

Ponteiros

- **Inicialização e atribuição:**
  - A constante NULL (e 0), portanto, permite apontar o ponteiro para uma **posição de memória inexistente**.
  - Para apontá-lo para uma **posição de memória válida**, podemos utilizar o endereço de uma variável que já faça parte do programa, ou seja, que **já esteja declarada**.
    - Lembre-se: quando criamos uma variável, o computador reserva um espaço na memória para ela.
  - Para saber o endereço de uma variável, utilizamos o **operador de endereçamento &** na frente de seu nome. Portanto, se quisermos atribuir o endereço de `contador` para `ptr_contador`, faríamos o seguinte:

```
int contador, *ptr_contador;

// atribuindo endereço
ptr_contador = &contador;
```

Ponteiros

- **Acessando valores e endereços:**
  - Como um ponteiro armazena apenas um endereço de memória, como fazemos para acessar o valor para onde ele aponta?
  - Como acessar o conteúdo da posição de memória para qual o ponteiro aponta?
  - Utilizamos o operador de de-referência: o asterisco (*).
    - Na prática, através deste operador, o ponteiro se torna uma variável comum. Quando de-referenciado, o valor da posição para qual o ponteiro aponta pode ser modificado, acessado e utilizado em expressões aritméticas.
  - **Exemplo:**
    - Vamos escrever um programa que realiza o cálculo da lei de Ohm, que define Tensão Elétrica = Resistência x Intensidade da Corrente (V = R x I), calculando o valor de uma tensão através da de-referência de ponteiros que apontam para valores de resistência e corrente.

```c
int main()
{
    float v1, v2, r, i;
    float *ptr_r, *ptr_i;

    // inicializando ponteiros
    ptr_r = &r;
    ptr_i = &i;

    // atribui valores para as variáveis
    printf("Entre com o valor de resistencia (ohms) e corrente (A): ");
    scanf("%f %f", &r, &i);

    v1 = r * i;
    v2 = (*ptr_r) * (*ptr_i); // a utilização de parenteses evita confusao e
                              // garante a precedencia

    printf("Tensao calculada por variaveis comuns: %.2f V\n", v1);
    printf("Tensao calculada por variaveis ponteiros: %.2f V\n", v2);

    return 0;
}
```

```
Entre com o valor de resistencia (ohms) e corrente (A): 10 50
Tensao calculada por variaveis comuns: 500.00 V
Tensao calculada por variaveis ponteiros: 500.00 V

Process returned 0 (0x0)   execution time : 1.744 s
Press any key to continue.
```

Ponteiros

- **Acessando valores e endereços:**
  - A seguinte tabela é fundamental para o completo domínio da utilização de ponteiros, e torna a passagem de parâmetros por referência uma tarefa bastante simples:

	Variáveis comuns	Variáveis ponteiro
Valor	x nenhum operador	*ptr_x operador asterisco de-referência
Endereço	&x operador e-comercial Endereçamento	ptr_x nenhum operador

Ao Trabalho!

Escreva um programa para o cálculo da **lei de ohm** que utiliza uma função com passagem por referência.

– A função de cálculo deve receber os três parâmetros do tipo float por referência, e não retornar nada (tipo void).

– O programa principal deve receber os dados de entrada, chamar a função corretamente, e imprimir o valor calculado de tensão.

```c
#include <stdio.h>

void ohm(float *tensao, float *resist, float *corrente)
{
    // modificamos o valor da tensao
    // atraves do calculo com valores de resistencia e corrente
    *tensao = (*resist) * (*corrente);
}

int main()
{
    float v, r, i;

    // atribui valores para as variáveis
    printf("Entre com o valor de resistencia (ohms) e corrente (A): ");
    scanf("%f %f", &r, &i);

    ohm(&v, &r, &i);

    printf("Tensao calculada pela funcao: %.2f V\n", v);

    return 0;
}
```

```
Entre com o valor de resistencia (ohms) e corrente (A): 10 50
Tensao calculada pela funcao: 500.00 V

Process returned 0 (0x0)   execution time : 1.894 s
Press any key to continue.
```

Ao Trabalho!

1) Escreva um programa para adicionar dois números com passagem por referência.
2) Escreva um programa para trocar os valores entre três variáveis, usando passagem por referência.
3) Escreva um programa para encontrar o maior entre dois números, usando ponteiros.

```c
#include <stdio.h>
#include <stdlib.h>
void main()
{
 int fno,sno,*ptr1=&fno,*ptr2=&sno;

   printf("\n\n Encontrar o valor máximo entre dois numeros usando ponteiros :\n");
   printf("-------------------------------------------------------------\n");

   printf("Informe o primeiro numero : ");
   scanf("%d", ptr1);
   printf("Informe o segundo numero : ");
   scanf("%d", ptr2);


 if(*ptr1>*ptr2)
 {
  printf("\n\n %d eh o numero maximo\n\n",*ptr1);
 }
 else
 {
  printf("\n\n %d eh o numero maximo.\n\n",*ptr2);
 }

}

```

```
 Encontrar o valor mßximo entre dois numeros usando ponteiros :
-------------------------------------------------------------
Informe o primeiro numero : 5
Informe o segundo numero : 15


 15 eh o numero maximo.


Process returned 27 (0x1B)   execution time : 2.666 s
Press any key to continue.
```

```c
#include <stdio.h>
long somarDoisNumeros(long *, long *);

int main()
{
    long pnum, snum, soma;

    printf("\n\n Somar dois numeros com passagem por referencia:\n");
    printf("-----------------------------------------------------\n");

    printf(" Informe o primeiro numero: ");
    scanf("%ld", &pnum);
    printf(" Informe o segundo numero: ");
    scanf("%ld", &snum);
    soma = somarDoisNumeros(&pnum, &snum);
    printf(" A soma de %ld e %ld  eh %ld\n\n", pnum, snum, soma);
    return 0;
}
long somarDoisNumeros(long *n1, long *n2)
{
    long soma;
    soma = *n1 + *n2;
    return soma;
}

```

```
Somar dois numeros com passagem por referencia:
-----------------------------------------------------
Informe o primeiro numero: 13
Informe o segundo numero: 22
A soma de 13 e 22  eh 35


Process returned 0 (0x0)   execution time : 2.990 s
Press any key to continue.
```

```c
#include <stdio.h>
void trocaNumeros(int *x,int *y,int *z);
int main()
{
    int e1,e2,e3;
    printf("\n\n Troca de elementos usando passagem por referencia :\n");
    printf("-------------------------------------------------------\n");
    printf(" Informe o valor do primeiro elemento : ");
    scanf("%d",&e1);
    printf(" Informe o valor do segundo elemento : ");
    scanf("%d",&e2);
    printf(" Informe o valor do terceiro elemento : ");
    scanf("%d",&e3);


    printf("\n O valor antes da troca eh :\n");
    printf(" elemento 1 = %d\n elemento 2 = %d\n elemento 3 = %d\n",e1,e2,e3);
    trocaNumeros(&e1,&e2,&e3);
    printf("\n O valor depois da troca eh :\n");
    printf(" elemento 1 = %d\n elemento 2 = %d\n elemento 3 = %d\n\n",e1,e2,e3);
    return 0;
}
void trocaNumeros(int *x,int *y,int *z)
{
    int tmp;
    tmp=*y;
    *y=*x;
    *x=*z;
    *z=tmp;
}
```

```
 Troca de elementos usando passagem por referencia :
-------------------------------------------------------
 Informe o valor do primeiro elemento : 5
 Informe o valor do segundo elemento : 8
 Informe o valor do terceiro elemento : 11

 O valor antes da troca eh :
 elemento 1 = 5
 elemento 2 = 8
 elemento 3 = 11

 O valor depois da troca eh :
 elemento 1 = 11
 elemento 2 = 5
 elemento 3 = 8


Process returned 0 (0x0)   execution time : 6.533 s
Press any key to continue.
```

Ponteiros

- **Atribuição entre ponteiros**
  - Um ponteiro pode, ainda, receber o endereço apontado por outro ponteiro (seu valor).
  - Apenas é válido se ambos os ponteiros forem do **mesmo tipo**. O compilador sempre assume que o endereço apontado contenha um valor do tipo especificado.
    - O compilador não irá acusar erro algum caso você tente atribuir um int* para um float*. No entanto, seu programa poderá apresentar um comportamento inesperado, que resultará em erros de lógica.
  - Veja o **exemplo**:

```
int *p, *p1;
int x = 10;

p1 = &x; // p1 recebe o endereço de x
p = p1; // p recebe o valor de p1 (endereço de x)

printf("p  = %d \np1 = %d \n", p, p1);
printf("Valor apontado: %d e %d\n\n", *p, *p1);
```

```
p  = 2686740
p1 = 2686740
Valor apontado: 10 e 10
```

Ponteiros

- **Operações Aritméticas**
  - Basicamente, apenas duas posições aritméticas podem ser utilizadas nos endereços armazenados pelos ponteiros. **Adição** e **subtração**.
  - Mas o que vocês acham que ocorre quando subtraímos ou adicionamos um valor ao ponteiro?

```
int *ptr_valor;
int valor = 100;

ptr_valor = &valor;
ptr_valor += 4; // acrescenta 4 ao valor do ponteiro
```

Exatamente!
Ele aponta para outro lugar!
No exemplo, ele se move **16 bytes** para frente: o que corresponde a **4 inteiros** (com 4 bytes cada). Se apontava para a posição de memória 100, agora aponta para a posição 116.

- Operações dependem do tipo de dado para o qual o ponteiro aponta;
- São bastante úteis quando trabalhamos com arrays, como veremos a seguir.

Ponteiros

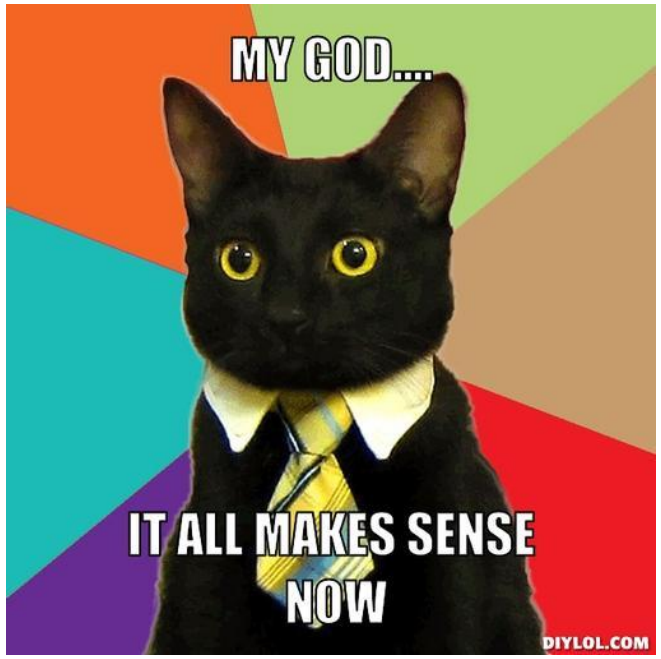
- **Ponteiros e Arrays:** Qual a relação?
  - Possuem uma **relação muito forte** dentro da linguagem C.
  - Arrays são agrupamentos de dados de um mesmo tipo na memória. Quando declaramos um array, informamos ao computador para que reserve uma quantidade de memória suficiente para armazenar os elementos do array de **forma sequencia**l.
  - Como resultado dessa operação, o computador nos **devolve um ponteiro** que aponta para o começo dessa sequência de bytes.
  - Portanto, prepare-se para uma revelação:

Ponteiros

- **Ponteiros e Arrays:** Qual a relação?
  - O NOME do array é, na verdade, **apenas um ponteiro** que aponta para o primeiro elemento do array.

```c
// Função recebe ponteiro como parametro
void imprime(int *vet, int tam)
{
    int i;

    // e - voilá - utiliza como vetor!!
    for(i = 0; i < tam; i++)
        printf("%d ", vet[i]);
    pritnf("\n");
}
```

Isso explica porque vetores são sempre passados por referência, e porque podem ser utilizados de forma equivalente na assinatura de funções.



Ao Trabalho!

- Escreva uma função que imprima os dados de um vetor de dez posições, composto por valores de 1 a 10, sem utilizar em momento algum o operador colchete do vetor.
  - Como poderíamos fazer?
  - Lembre-se que, na linguagem C, o nome de um array **sem índice** guarda o endereço para o começo do array na memória.
    - Aponta para seu primeiro elemento.

```c
#include <stdio.h>

// Função recebe ponteiro como parametro
void imprime(int *vet, int tam)
{
    int i;
    for(i = 0; i < tam; i++)
        printf("%d ", vet[i]);
    printf("\n");
}

// Função recebe ponteiro como parametro
void imprime_aritm(int *vet, int tam)
{
    int i;
    for(i = 0; i < tam; i++)
    {
        printf("%d ", *vet);
        vet++;
    }
    printf("\n");
}

// Função Principal
int main()
{
    int vetor[] = {1 ,2, 3, 4, 5, 6, 7, 8, 9, 10};

    printf("Impressão Comum: ");
    imprime(vetor, 10);
    printf("Impressão com operacao aritmetica: ");
    imprime_aritm(vetor, 10);

    return 0;
}
```

- Imprimimos sempre o elemento para o qual o ponteiro **vet** está apontando
- Incrementamos o ponteiro **vet** para a próxima posição do vetor

```
Impressao Comum: 1 2 3 4 5 6 7 8 9 10
Impressao com operacao aritmetica: 1 2 3 4 5 6 7 8 9 10

Process returned 0 (0x0)   execution time : 0.009 s
Press any key to continue.
```

O operador **colchetes[]** substitui o uso de operações aritméticas e de de-referência (*) no acesso ao conteúdo de uma posição do array. **Simplifica** o acesso ao array.

Ponteiros

- Em C, utilizamos ponteiros para **simular** **chamadas por referência**.
  - Os endereços de variáveis são passados como parâmetros da função.
  - Dentro dela, acessamos seu valor através do operador de de-referência (*)
  - Para chamá-la podemos tanto passar uma variável do tipo ponteiro previamente inicializada quanto utilizar o operador de endereçamento (&).

Ponteiros

- **Alocação dinâmica:**
  Além disso, a linguagem C permite alocar dinamicamente (em tempo de execução) blocos de memória utilizando ponteiros.
  - Permite que novos blocos de memória sejam reservados durante a execução, de maneira que não precisem ser sempre pré-definidos durante a escrita do código.
  - A função **malloc**, por exemplo, reserva a quantidade de memória especificada como parâmetro e retorna um ponteiro com o endereço do início do bloco alocado. Veja:

```c
#include <stdio.h>

int main()
{
    int *din, i;

    // aloca um espaço de 5*4 = 20 bytes
    // e retorna endereço do inicio do bloco
    din = (int*) malloc(5*sizeof(int));

    // acesso aos elementos identico ao acesso em vetores estaticos
    for(i = 0; i < 5; i++)
    {
        printf("Entre com o valor do %do. elemento: ", i+1);
        scanf("%d", &din[i]);
    }

    // Imprime valores
    printf("Vetor Inserido: ");
    for(i = 0; i < 5; i++)
        printf("%d ", din[i]);
    printf("\n\n");

    // Vetores alocados dinamicamente nao sao desalocados
    // automaticamente. É necessario liberar a memoria ao final
    // do programa.
    free(din);

    return 0;
}
```

```
Entre com o valor do 1o. elemento: 1
Entre com o valor do 2o. elemento: 2
Entre com o valor do 3o. elemento: 3
Entre com o valor do 4o. elemento: 4
Entre com o valor do 5o. elemento: 5
Vetor Inserido: 1 2 3 4 5
```

Ao Trabalho!

1) Imprima os elementos de um vetor na ordem inversa usando ponteiros.
2) Imprima todos o alfabeto usando ponteiros.
3) Encontre o maior elemento usando a alocação dinâmica de memória.
4) Faça um programa para ler um vetor, que imprimir todos os elementos e também a soma dos mesmos, usando a alocação dinâmica de memória.