PONTEIROS – Lista 9

Algumas tarefas em C/C++ se tornam mais fáceis com uso dos ponteiros e outras tarefas (como estruturas com alocação dinâmica) simplesmente não podem ser realizadas sem uso dos ponteiros.

Cada variável é alocada em memoria e essa memoria tem um endereço especifico que pode ser acessado com operador (**&**).

Esse operador, aplicado ao nome de uma variável, indica que se trata de um endereço na memoria daquela variável.

O exemplo a seguir mostra como podemos acessar o endereço de uma variável usando o operador **&**.

Os endereços da memoria de computador são representados em formato hexadecimal. Portanto, para imprimir o endereço de uma variável usando função **printf()** precisamos de especificador correspondente.

Alguns dos especificadores da função **printf()**

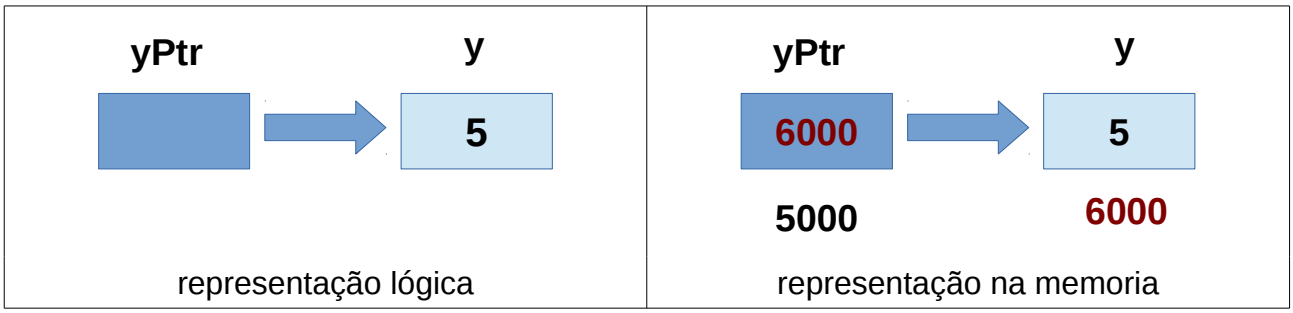
Especificador	Tipo de dados
o	Unsigned octal
x	Unsigned hexadecimal integer
e	Scientific notation (mantissa/exponent)
a	Hexadecimal floating point

Exemplo 1 (0,5p): operador &

```c
#include <stdio.h>

int main ()
{
   int  var1;
   char var2[10];

   printf("Address of var1 variable: %x \n", &var1 );
   printf("Address of var2 variable: %x \n", &var2 );

   return 0;
}
```

Address of var1 variable: 49298e7c
Address of var2 variable: 49298e80

Ponteiro é uma variável que armazena o endereço da memoria de uma outra variável.



Como uma variável comum, o ponteiro deve ser declarado e inicializado antes de ser usado em programa.

A forma geral de declaração da variável do tipo ponteiro:

```
type * varName;
```

onde

type – é um dos tipos de dados de C/C++ (int, float, double, char, ...)
varName – é o nome da variável

Na verdade, independentemente do tipo de dados declarado, uma variável do tipo ponteiro sempre vai armazenar um número hexadecimal, que representa um endereço de memoria.

O tipo de dados da declaração de um ponteiro especifica que tipo de variáveis esse ponteiro poderá referenciar.

Operações com ponteiros

As principais operações que podem ser efetuadas com ponteiros são:

- declaração de uma variável do tipo ponteiro
- atribuição de endereço de alguma variável para o ponteiro
- acesso a variável para qual o ponteiro esta apontando

O operador unário (*****) aplicado a um ponteiro retorna o valor da variável endereço da qual é armazenado naquele ponteiro.

Observação:
Os operadores **&** e ***** se complementam.

Exemplo 2 (0,5p): operadores * e &

```c
#include <stdio.h>

int main ()
{
   int  y = 5;    /* actual variable declaration */
   int  *yPtr;    /* pointer variable declaration */

   yPtr = &y;  /* store address of var in pointer variable*/

   printf("Address of y variable: %x \n", &y  );

   /* address stored in pointer variable */
   printf("Address stored in yPtr variable: %x \n", yPtr );

   /* y value */
   printf("Value of y: %d\n", y );
   /* access the value using the pointer */
   printf("Value of * yPtr variable: %d \n", *yPtr );

   /* *& and &*  */
   printf("Value of *& yPtr : %x \n", *&yPtr );
   printf("Value of &* yPtr : %x \n", &*yPtr );

   return 0;
}
```

```
Address of y variable: f6489004
Address stored in yPtr variable: f6489004
Value of y: 5
Value of * yPtr variable: 5
Value of *& yPtr : f6489004
Value of &* yPtr : f6489004
```

Ponteiros do tipo NULL

Boa pratica de programação:

Atribuir o valor NULL para as variavas do tipo ponteiro ajuda evitar os resultados inesperáveis do programa.

A palavra-chave NULL indica que o ponteiro não aponta para nenhum lugar especifico da memoria.

A maioria das bibliotecas interpreta NULL como uma constante com valor zero.

Na maioria dos sistemas operacionais os programas não tem permissão para acessar a memoria com endereço 0, porque essa parte é reservada para próprio sistema operacional.

Nesse sentido o valor zero serve para indicar que o ponteiro não aponta para nenhum lugar acessível da memoria (em outras palavras aponta para nada).

Exemplo 3 (0,5p): NULL

```c
#include <stdio.h>

int main ()
{
    int  *ptr = NULL;

    printf("The value of ptr is : %x\n", ptr  );

    return 0;
}
```

The value of ptr is : 0

Ponteiros e vetores

Em C/C++ nome de um vetor (uni o multidimensional) é na verdade é um ponteiro para primeiro elemento.

Então a declaração:

```
int v[10];
```

significa que **v** é um ponteiro para **&v[0]**.

Declaração a seguir atribui para o ponteiro **ptr** o endereço do primeiro elemento do vetor **v**.

```
int *ptr;
int v[10];

ptr = v;
```

Atribuições de valores entre os ponteiros e nomes dos vetores são válidas e permitidas pela linguagem.

Assim sendo o elemento v[4] da vetor pode ser acessado como

```
* ( v + 4)
```

No nosso exemplo, uma vez que o endereço do primeiro elemento do vetor **v** é armazenado em ponteiro **ptr** todos os elementos do vetor **v** podem ser acessados usando ***ptr, *(ptr+1), *(ptr + 2), ...** .

Operações aritméticas com ponteiros

O valor armazenado em um ponteiro é um endereço de memoria que é um número. Por isso é possível realizar as operações aritméticas com ponteiros: **+, - , ++** e **--**.

Essas operações serão executadas considerando o tipo de dados para qual um ponteiro especifico pode apontar.

Por exemplo se temos um ponteiro **iPtr** que aponta para variável do tipo **int**:

```
int * iPtr;
```

Vamos considerar a seguinte situação:

- o endereço armazenado no ponteiro **iPtr** é **1000**
- o tamanho de um **int** no nosso sistema é de **4** bytes

Depois de executar o comando:

```
iPtr++;
```

o endereço armazenado em **iPtr** será **1004**.

Agora vamos considerar uma situação diferente:

- o ponteiro **cPtr** aponta para variáveis do tipo **char**
- o endereço armazenado no ponteiro **cPtr** é **1000**
- o tamanho de um **char** é de **1** byte

```
char * cPtr;
```

Nesse caso, depois de executar o comando:

```
cPtr++;
```

o endereço armazenado em **cPtr** será **1001**.

Exemplo 4 (0,5p): Incremento de ponteiros

```c
#include <stdio.h>

int main ()
{
   int  v[10] = {10, 100, 200, -3, 1, 0 , 45, 67, 8, 23};
   int  i, *ptr;

   /* let us have array address in pointer */
   ptr = v;
   for ( i = 0; i < 10; i++)
   {

      printf("\n Address of v[%d] = %x \n", i, ptr );
      printf(" Value of v[%d] = %d \n", i, *ptr );

      ptr++;  /* move to the next element */
   }
   return 0;
}
```

```
Address of v[0] = 1b60480
Value of v[0] = 10

Address of v[1] = 1b60484
Value of v[1] = 100

Address of v[2] = 1b60488
Value of v[2] = 200

Address of v[3] = 1b6048c
Value of v[3] = -3

Address of v[4] = 1b60490
Value of v[4] = 1

Address of v[5] = 1b60494
Value of v[5] = 0

Address of v[6] = 1b60498
Value of v[6] = 45

Address of v[7] = 1b6049c
Value of v[7] = 67

Address of v[8] = 1b604a0
Value of v[8] = 8

Address of v[9] = 1b604a4
Value of v[9] = 23
```

Operações de comparação de ponteiros

Os ponteiros podem ser comparados usando operadores relacionais (<, > e ==). Essa comparação faz sentido se dois ponteiros apontam para dados relacionados entre si, como os elementos de um vetor por exemplo.

Outra comparação que pode ser feita com ponteiros é a verificação se um ponteiro é nulo, por exemplo:

if (ptr)	é considerado bem sucedido se o ponteiro **ptr** não é nulo
if (! ptr)	é considerado bem sucedido se o ponteiro **ptr** é nulo

Exemplo 5 (1p): Comparação entre ponteiros

```c
#include <stdio.h>

const int arraySize = 10;

int main ()
{
   int  v[] = {10, 100, 200, -3, 1, 0 , 45, 67, 8, 23};
   int  i, *ptr;

   i = 0;
   ptr = v;

   while ( ptr <= &v[arraySize - 1] )
   {
     printf("\nAddress of v[%d] = %x\n", i, ptr );
     printf("Value of v[%d] = %d\n", i, *ptr );

     ptr++;
     i++;
    }

   return 0;
}
```

```
Address of v[0] = 3bbbe240
Value of v[0] = 10

Address of v[1] = 3bbbe244
Value of v[1] = 100

...

Address of v[9] = 3bbbe264
Value of v[9] = 23
```

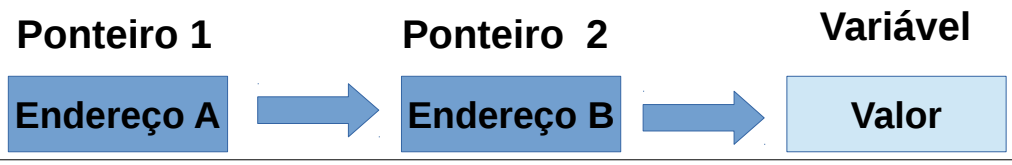
Exemplo 6 (1p): Vetor de ponteiros

```c
#include <stdio.h>

const int arraySize = 10;

int main ()
{
    int  v[] = {10, 100, 200, -3, 1, 0 , 45, 67, 8, 23};
    int  i, *ptr[arraySize];;

    for ( i = 0; i < arraySize; i++)
    {
       ptr[i] = &v[i]; /* assign the address of integer. */
     }

     for ( i = 0; i < arraySize; i++)
     {
       printf("Value of v[%d] = %d\n", i, *ptr[i] );
     }
    return 0;
}
```

```
Value of v[0] = 10
Value of v[1] = 100
Value of v[2] = 200
Value of v[3] = -3
Value of v[4] = 1
Value of v[5] = 0
Value of v[6] = 45
Value of v[7] = 67
Value of v[8] = 8
Value of v[9] = 23
```

Ponteiro para ponteiro

Geralmente um ponteiro contem o endereço de uma variável. C/C++permite declaração de ponteiro para ponteiro: nesse caso o primeiro ponteiro contem endereço do segundo ponteiro e esse segundo ponteiro contem o endereço da variável.



Ponteiro para ponteiro

Uma variável do tipo ponteiro para ponteiro deve ser declarada de forma correspondente.

Isso é feito colocando um simbolo de ***** adicional na declaração.

```
int ** pptr;
```

Para acessar a variável a partir do ponteiro para ponteiro o operador ***** deve ser usado duas vezes:

**** pptr**

Exemplo 7 (1p): Ponteiro para ponteiro

```
 5    int  var;
 6    int  *ptr;
 7    int  **pptr;
 8
 9    var = 5;
10
11    /* take the address of var */
12    ptr = &var;
13
14    /* take the address of ptr using address of operator & */
15    pptr = &ptr;
16
17    /* take the value using pptr */
18    printf(" Value available at **pptr = %d", **pptr);
19    printf("\n Address stored in pptr = %x \n", pptr );
20
21    printf("\n Value available at *ptr = %d", *ptr );
22    printf("\n Address of ptr = %x ", &ptr );
23    printf("\n Address stored in ptr = %x \n", ptr );
24
25    printf("\n Value of var = %d", var );
26    printf("\n Address of var = %x \n", &var );
```

```
Value available at **pptr = 5
Address stored in pptr = b2af0960

Value available at *ptr = 5
Address of ptr = b2af0960
Address stored in ptr = b2af095c

Value of var = 5
Address of var = b2af095c
```

Modos de passagem de argumentos para função

Exitem duas possibilidades de passagem de argumentos para função em linguagem C:

- passagem por valor
- passagem por referencia

Observação:

Em C++ existem duas possibilidades de passagem por referencia, enquanto em C existe somente uma forma.

Passagem por valor

Quando acontece a passagem por valor a função recebe uma copia de argumento.

A função pode executar qualquer tipo de operação com o valor do argumento, mas o valor original da variável permanecerá o mesmo depois da execução da função.

Esse modo de passagem de parâmetros evita uma eventual alteração de dados por uma das funções do programa e melhora a qualidade e segurança de software.

A desvantagem desse modo é que no caso de grandes volumes de dados em todas as chamadas de função uma copia desse dados será criada, que pode causar uma queda de desempenho de software.

Passagem por referencia

Nesse caso a função recebe o endereço do argumento e pode alterar o valor original do argumento.

No caso de grandes volumes de dados esse modo permite um aumento de desempenho significativo, porém diminui a segurança de software.

Exemplo 8 (1p): Passagem por valor e passagem por referencia

```
3   int  byValue(int a);// passagem de argumento por valor
4   void byPtr(int *ptr); // passagem de argumento por referencia
5   //---------------------------------------------------------------------
6   int main ()
7   {
8      int num;
9      int x = -5, y = -5;
10
11     printf("\n Passagem de argumento por ByValue ");
12     printf("\n x = %i", x);
13     printf("\n Chamada de função  num = byValue(x) ");
14     num = byValue(x);
15     printf("\n num = %i", num);
16     printf("\n x = %i", x);
17
18     printf("\n\n\n Passagem de argumento por byPtr ");
19     printf("\n y = %i", y);
20     printf("\n Chamada de função  byPtr(&y)");
21     byPtr(&y);
22     printf("\n y = %i \n", y);
23
24     return 0;
25  }
26  //=====================================================================
27  // passagem de argumento por valor
28  int  byValue(int a)
29  {
30    if( a < 0 )
31      return a * -1;
32    else
33      return a;
34  }
35  //---------------------------------------------------------------------
36   // passagem de argumento por referencia
37  void byPtr(int *ptr)
38  {
39    if( *ptr < 0 )
40      *ptr = *ptr * -1;
41    return;
42  }
```

Passagem de argumento por ByValue
x = -5
Chamada de função num = byValue(x)
num = 5
x = -5

Passagem de argumento por byPtr
y = -5
Chamada de função byPtr(&y)
y = 5

Uma função que recebe um ponteiro como argumento também pode receber um vetor, já que o nome do vetor na verdade é um ponteiro para primeiro elemento dele.

Exemplo 9 (1p): Passagem de vetor para função (por referencia)

```c
#include <stdio.h>

float getAverage(int *arr, int size);
//----------------------------------------
int main ()
{
   int v[5] = {20, 30, 10, 20, 20};
   double avg;

   avg = getAverage( v, 5 ) ;

   printf("Average value is: %.2f\n", avg );

   return 0;
}
//========================================
float getAverage(int *arr, int size)
{
  int i, sum = 0;
  float avg;

  for (i = 0; i < size; i++)
  {
    sum += arr[i];
  }

  avg = (float)sum / size;

  return avg;
}
```

Average value is: 20.00

Função que retorna um ponteiro

A linguagem C/C++ permite que a função retorne um ponteiro.

Por outro lado a linguagem não permite que uma função retorne um vetor de forma explicita.

Como um vetor pode ser interpretado como ponteiro para o primeiro elemento dele, retornar um ponteiro de uma função na verdade é uma forma de retornar um vetor.

Uma coisa que deve ser levada em consideração é que as variáveis locais declaradas dentro de uma função existem somente durante a execução daquela função.

Caso desejamos retornar uma variável local ela deve ser declarada como **static**.

Exemplo 10 (1p): Função que retorna um vetor

```c
#include <stdio.h>

/* function to generate and return 10 random numbers */
int * getRandom( );
//-------------------------------------------------------
int main ()
{
   /* a pointer to an int */
   int *p;
   int i;

   p = getRandom();
   printf("\n Acesso em main(): \n");
   for ( i = 0; i < 10; i++ )
   {
       printf( "*(p + %d) : %d\n", i, *(p + i));
   }

   return 0;
}
//=======================================================
int * getRandom( )
{
  static int  r[10];
  int i;

  printf("\n Números gerados dentro da função: \n");
  for ( i = 0; i < 10; ++i)
  {
     r[i] = 1 + rand() % 20;
     printf( "r[%d] = %d\n", i, r[i]);
  }

  return r;
}
```

Números gerados dentro da função:
r[0] = 4
r[1] = 7
r[2] = 18
r[3] = 16
r[4] = 14
r[5] = 16
r[6] = 7
r[7] = 13
r[8] = 10
r[9] = 2

Acesso em main():
*(p + 0) : 4
*(p + 1) : 7
*(p + 2) : 18

```
*(p + 3) : 16
*(p + 4) : 14
*(p + 5) : 16
*(p + 6) : 7
*(p + 7) : 13
*(p + 8) : 10
*(p + 9) : 2
```

Exercícios:

Exercício 11 (3p):

- criar
  - vetores **a** e **c** com 5 elementos
  - vetores **b** e **d** com 15 elementos.
- inicializar os vetores **a** e **b**
- receber um numero **n** do usuário.
- criar uma função que recebe dois vetores e um numero, multiplica os valores do primeiro vetor pelo número e armazena o resultado em segundo vetor.
- multiplicar os vetores **a** e **b** por número **n** e armazenar o resultado em vetores **c** e **d** respectivamente.

Exercício 12 (3p):

- criar o vetor **a** com 10 elementos e o vetor **b** com 20 elementos
- inicializar esses vetores com números aleatórios no intervalo [1,50]
- usando funções:
  - calcular o valor médio dos elementos.
  - calcular a soma dos elementos acima da média.
  - multiplicar por -1 todos os elementos com valores abaixo da media.

Exercício 13 (3p):

- criar o vetor **a** com 5 elementos e o vetor **b** com 10 elementos
- inicializar os vetores com dados fornecidos por usuário, sendo que todos os elementos devem ser positivos (usar uma função para controle de entrada de dados)
- calcular o valor médio dos elementos pares em cada um dos vetores.
- dividir todos os elementos do vetor pelo menor elemento dele.