Ponteiros

PE-06 – v1.2

Prof. Paulo Joia Filho

Objetivos

Os principais objetivos desta aula são:

- Apresentar o conceito de ponteiro.
- Explorar a passagem de parâmetros e valores de retorno de funções.
- Compreender a relação entre ponteiro e arranjo.
- Entender a aritmética de ponteiros.

Por que estudar ponteiros?

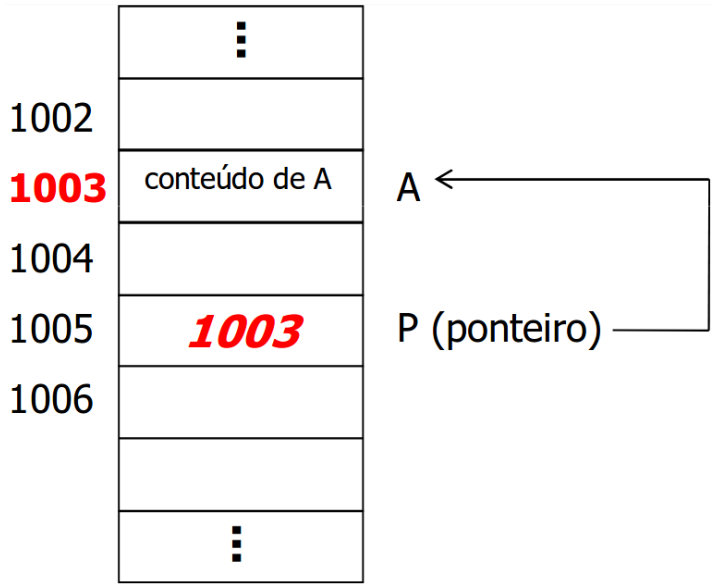
Entre outras funcionalidades, ponteiros permitem...

- Passar parâmetros para funções **"por referência"**.
- **Alocar e liberar memória dinamicamente** (em tempo de execução).
- Implementação eficiente de certas **estruturas de dados**.

O que são ponteiros?

- Ponteiro é uma variável que armazena um endereço de memória;
- Esse endereço, normalmente, é a posição de outra variável na memória;
- Se uma variável contém o endereço de outra, dizemos que a primeira variável aponta para a segunda, daí a origem do termo “ponteiro”.

Exemplo:



Declaração de um ponteiro

- Para declarar uma variável do tipo ponteiro, deve-se utilizar o operador * (asterisco). A forma geral para declaração é:

```
tipo *nome;
```

☞ Note que o tipo do ponteiro está associado ao tipo da variável apontada

- Quando um ponteiro é declarado, seu valor inicial é desconhecido (pode conter qualquer valor). Para evitar problemas futuros, é comum criarmos um ponteiro apontando-o para nulo:

```
int *p = NULL;
```

- Para imprimir o endereço de memória de um ponteiro, utilize o formatador **%p**. Exemplo:

```
printf("O ponteiro p aponta para %p", p);
```

Operadores de ponteiros [ponteiros_operadores.c]

- Os principais operadores para ponteiros são: `*` e `&`.
- O operador `&` (*address of*) é um operador unário que devolve o endereço de memória do seu operando.

Exemplo 1

```
1  #include<stdio.h>

3  int main() {
       int count = 100;
5      int *p;

7      p = &count;

9      printf("O endereço apontado por p é %p\n", p);

11     return 0;
   }
```

Operadores de ponteiros

- É colocado em `p` o endereço da memória que contém a variável `count`.

- O operador `&` pode ser imaginado como “o endereço de”. Assim, o comando de atribuição do exemplo anterior significa “`p` recebe **o endereço de** `count`”.

Operadores de ponteiros

- O operador `*` é o complemento de `&`. É um operador unário que devolve o valor da variável localizada no endereço que o segue

```
int q = *p;
```

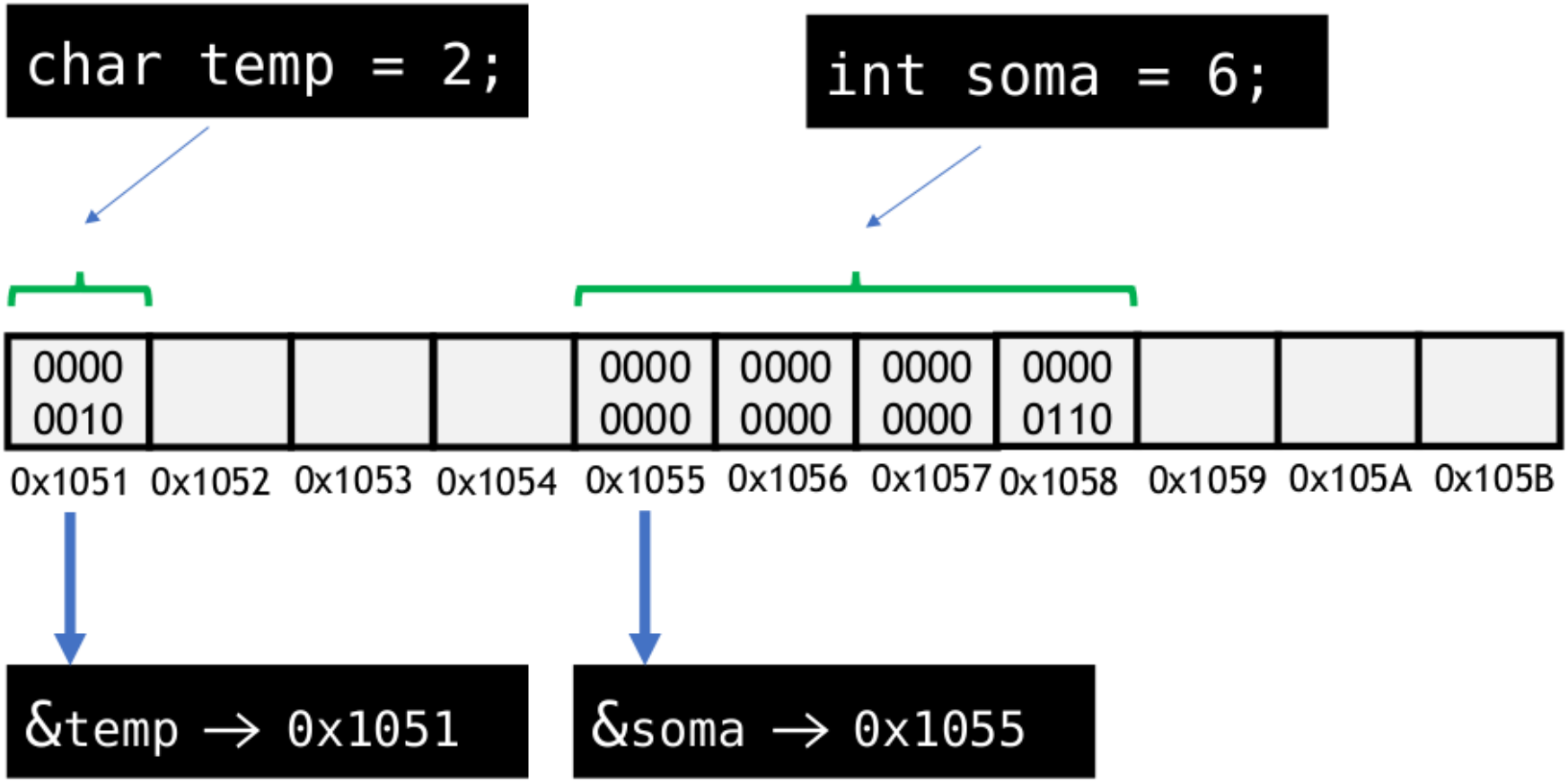
- Sabendo que `p` contém o endereço da variável `count`, i.e., `p` aponta para `count` podemos pensar no resultado de duas maneiras:
  - O valor de `count` é colocado em `q`.
  - O valor da variável apontada por `p` é colocado em `q`.

- O operador `*` pode ser imaginado como "o valor da variável apontada por", assim como o exemplo acima significa "`q` recebe **o valor da variável apontada por** `p`".

Memória

Para facilitar, podemos entender a memória como um grande vetor de bytes, devidamente endereçado.

Endereço de uma variável



Endereço de uma variável

[ponteiros_endereco.c]

Exemplo 2

```c
#include<stdio.h>

int main() {
    int num = 800;
    printf("Endereço de num = %d é %p\n", num, &num);

    return 0;
}
```

Saída

```
Endereço de num = 800 é 0x7fff6b9a3ca4
```

Exemplo 3 (Tamanho de um ponteiro)

[ponteiros_endereco2.c]

```
1  #include <stdio.h>

3  int main() {
       char var1;
5      int var2;
       double var3;
7      char *ponteiro1;
       int *ponteiro2;
9      double *ponteiro3;

11     printf("Tamanhos %ld %ld %ld\n",
              sizeof(var1),
13            sizeof(var2),
              sizeof(var3));
15     printf("Tamanhos %ld %ld %ld\n",
              sizeof(ponteiro1),
17            sizeof(ponteiro2),
              sizeof(ponteiro3));
19     return 0;
   }
```

O que será impresso?

Depende do compilador:

32 ou 64-bit?

Exemplo 3 (Tamanho de um ponteiro)

[ponteiros_endereco2.c]

```
1  #include <stdio.h>

3  int main() {
       char var1;
5      int var2;
       double var3;
7      char *ponteiro1;
       int *ponteiro2;
9      double *ponteiro3;

11     printf("Tamanhos %ld %ld %ld\n",
               sizeof(var1),
13             sizeof(var2),
               sizeof(var3));
15     printf("Tamanhos %ld %ld %ld\n",
               sizeof(ponteiro1),
17             sizeof(ponteiro2),
               sizeof(ponteiro3));
19     return 0;
   }
```

O que será impresso?

Depende do compilador:

32 ou 64-bit?

32-bit:

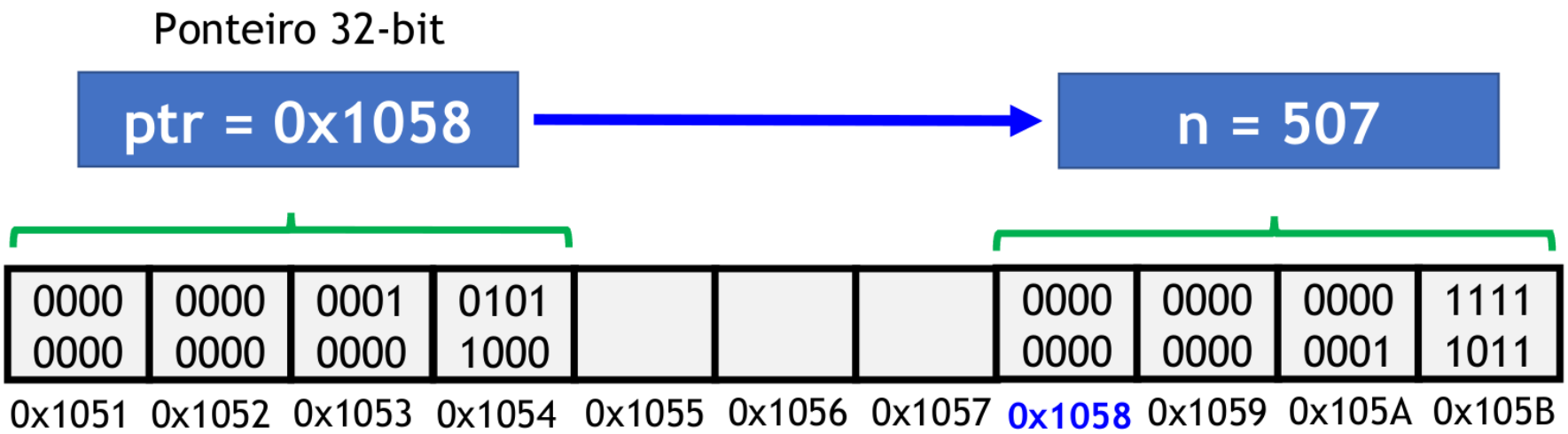
```
Tamanhos 1 4 8
Tamanhos 4 4 4
```

64-bit:

```
Tamanhos 1 4 8
Tamanhos 8 8 8
```

Ponteiro

```
int n = 507;
int *ptr;
ptr = &n;
```



Endereço do ponteiro

[ponteiros_endereco3.c]

Um ponteiro é uma variável, portanto, ele é armazenado em um endereço de memória também.

Exemplo 4

```
1  #include<stdio.h>
2
3  int main() {
4      int num = 507;
5      int *pt1 = &num;
6
7      printf("%d %p %p %p\n", num, &num, pt1, &pt1);
8
9      return 0;
10 }
```

O que será impresso?

Endereço do ponteiro

[ponteiros_endereco3.c]

Um ponteiro é uma variável, portanto, ele é armazenado em um endereço de memória também.

Exemplo 4

```
1  #include<stdio.h>

3  int main() {
       int num = 507;
5      int *pt1 = &num;

7      printf("%d %p %p %p\n", num, &num, pt1, &pt1);

9      return 0;
   }
```

O que será impresso?

```
507 0x7ffde2c316cc 0x7ffde2c316cc 0x7ffde2c316d0
```

Acessando o valor do endereço

```
int n = 507;
int *ptr = &n;
```

Declaração de uma variável do tipo **int**

Declaração de um ponteiro, que é inicializado apontando para **n**

```
*ptr = 25;
```

Altera o valor da variável que **ptr** aponta

Acessando o valor do endereço

[ponteiros_endereco4.c]

Exemplo 5

```
1  #include<stdio.h>

3  int main() {
       int n = 507;
5      int *ptr;
       ptr = &n;
7
       *ptr = *ptr + 1;
9
       printf("%d\n", n);
11     printf("%d\n", *ptr);

13     return 0;
   }
```

O que será impresso?

Acessando o valor do endereço

[ponteiros_endereco4.c]

Exemplo 5

```
1  #include<stdio.h>

3  int main() {
       int n = 507;
5      int *ptr;
       ptr = &n;
7
       *ptr = *ptr + 1;
9
       printf("%d\n", n);
11     printf("%d\n", *ptr);

13     return 0;
   }
```

O que será impresso?

```
508
508
```

[ponteiros1.c]

Exemplo 6 (Modificando o valor a partir de um ponteiro)

```
1  #include<stdio.h>
2
3  int main() {
4      int count = 100;
5      int *p;
6
7      p = &count;
8
9      printf("%d\n", *p);      /* Imprimira 100 */
10     *p = 34;                 /* Altera valor de count para 34 */
11     printf("%d\n", count);   /* Imprimira 34 */
12     printf("%d\n", *p);      /* Imprimira 34 */
13
14     return 0;
15 }
```

[ponteiros2.c]

Exemplo 7

```
1  #include<stdio.h>

3  int main() {
       int x = 2;
5      int *p;

7      p = &x;     /* Ponteiro recebe "o endereco de" x */
       *p = 4;     /* "O valor da variavel apontada por p" recebe 4 */
9
       printf("valor de x: %d\n", x);
11
       return 0;
13 }
```

O que será impresso?

[ponteiros2.c]

Exemplo 7

```c
#include<stdio.h>

int main() {
    int x = 2;
    int *p;

    p = &x;     /* Ponteiro recebe "o endereco de" x */
    *p = 4;     /* "O valor da variavel apontada por p" recebe 4 */

    printf("valor de x: %d\n", x);

    return 0;
}
```

O que será impresso?

```
valor de x: 4
```

[ponteiros3.c]

Exemplo 8

```
1  #include<stdio.h>

3  int main(){
       int x = 2;
5      int *p1, *p2;

7      /* Um ponteiro só recebe um endereço ou um outro ponteiro*/
       p1 = &x;
9      p2 = p1;

11     printf("%p = %p? Apontam para o mesmo endereço?\n", p1, p2);

13     printf("Valor apontado por p1: %d\n", *p1);   /* Imprimira 2 */
       printf("Valor apontado por p2: %d\n", *p2);   /* Imprimira 2 */
15
       return 0;
17 }
```

Passagem de Parâmetros

[ponteiros_valor.c]

A passagem de parâmetros em C é por valor, isto é, quando se passa o valor de uma variável para uma função, tem-se a certeza que seu valor não será alterado.

Exemplo 9 (Passagem por valor)

```
1  #include<stdio.h>
2
3  void incrementa (int x) {
4      x++;   /* Incrementa x */
5  }
6
7  int main(int argc, char *argv[]) {
8      int x = 10;
9      incrementa(x);   /* Passei o valor de x */
10     printf("valor de x: %d\n", x);   /* imprimira 10 */
11
12     return 0;
13 }
```

Simulando chamada por referência

- Embora a convenção de passagem de parâmetros em C seja por valor, é possível **simular** uma chamada por referência, passando o endereço de uma variável para um ponteiro.

- Desse modo, o endereço da variável é passado à função e, consequentemente, pode-se alterar o valor da variável fora da função.

Lembrando que...

Isso não se qualifica como passagem por referência, uma vez que o endereço da variável é **copiado** para o ponteiro, i.e., ainda é passagem por valor.

Simulando chamada por referência [ponteiros_referencia.c]

Para passar parâmetros por referência precisamos passar ponteiros por valor!

Exemplo 10 (Passagem por referência)

```
1  #include<stdio.h>
2
3  void incrementa (int *x) {
4      (*x)++;   /* Incrementa o valor da variavel apontada por x */
5  }
6
7  int main() {
8      int x = 10;
9      incrementa(&x);   /* Passei o endereco de x */
10     printf("valor de x: %d\n", x);   /* imprimira 11 */
11
12     return 0;
13 }
```

[ponteiros_troca.c]

Exemplo 11 (Função de troca de valores)

```
1  #include<stdio.h>

3  void troca(int *, int *); /* Protótipo */

5  int main() {
       int x = 2, y = 4;
7      printf("x = %d e y = %d\n", x, y);
       troca(&x, &y);
9      printf("x = %d e y = %d\n", x, y);

11     return 0;
   }
13
   void troca(int *px, int *py){ /* Observe o uso do asterisco (*) */
15     int temp = *px;
       *px = *py;
17     *py = temp;
   }
```

Caso de uso: função `scanf`

[ponteiros_scanf.c]

- O **`scanf`** é uma função que recebe argumentos passados por referência: o valor das variáveis é alterado pelo **`scanf`**;
- Por isso, usamos o operador **`&`** (*address of*), ou seja, temos que passar o endereço da variável no **`scanf`**;
- Se já temos um endereço de memória, podemos passá-lo diretamente, sem usar o operador **`&`**.

Exemplo 12

```
1  #include<stdio.h>
2
3  int main() {
4      int num;
5      int *pt1 = &num;
6
7      scanf("%d", pt1);
8      printf("%d\n", num);
9
10     return 0;
11 }
```

Retornando mais de um valor

Para retornar mais de um valor em C, passe os parâmetros por referência.

Exemplo:[1]

```
void divint(int dividendo, int divisor, int *quociente, int *resto){
   *quociente = dividendo / divisor;
   *resto = dividendo % divisor;
}
```

Erroneamente, algumas pessoas acreditam que a linguagem C é limitada por não admitir mais de um valor de retorno em suas funções. Isso demonstra a falta de conhecimento da linguagem, a qual supre essa limitação com o uso de ponteiros em seus argumentos.

[1] Note que as variáveis **quociente** e **resto** são ponteiros.

[multiplos_retornos.c]

Exemplo 13 (Função com múltiplos valores de retorno)

```
1  #include <stdio.h>
2
3  void divint(int dividendo,int divisor,int *quociente, int *resto){
4      *quociente = dividendo / divisor;
5      *resto = dividendo % divisor; /* divisão inteira */
6  }
7
8  int main() {
9      int a, b;
10     printf("Informe o dividendo: ");
11     scanf("%d", &a);
12     printf("Informe o divisor:   ");
13     scanf("%d", &b);
14
15     int q, r;
16     divint(a, b, &q, &r);
17     printf("Quociente = %d\nResto = %d\n", q, r);
18     return 0;
19 }
```

Qual a saída quando dividendo= 82 e divisor= 9?

[multiplos_retornos.c]

Exemplo 13 (Função com múltiplos valores de retorno)

```c
#include <stdio.h>

void divint(int dividendo,int divisor,int *quociente, int *resto){
    *quociente = dividendo / divisor;
    *resto = dividendo % divisor; /* divisão inteira */
}

int main() {
    int a, b;
    printf("Informe o dividendo: ");
    scanf("%d", &a);
    printf("Informe o divisor:   ");
    scanf("%d", &b);

    int q, r;
    divint(a, b, &q, &r);
    printf("Quociente = %d\nResto = %d\n", q, r);
    return 0;
}
```

Qual a saída quando dividendo= 82 e divisor= 9?

```
Informe o dividendo: 82
Informe o divisor:   9
Quociente = 9
Resto = 1
```

Arranjos e Ponteiros

Segundo Kernighan e Ritchie, 1988:

Em C, existe um forte relacionamento entre ponteiros e arranjos.

- *Forte o suficiente para que ponteiros e arranjos sejam discutidos simultaneamente.*
- *Qualquer operação que possa ser realizada com arranjos também pode ser realizada com ponteiros.*
- *A versão com ponteiro é, em geral, mais rápida. Mas, para os iniciantes, algumas vezes mais difícil de entender.*

Pergunta

Mais rápida quanto?

Arranjos e Ponteiros

- O **nome** de um **arranjo** é um ponteiro para o primeiro elemento do arranjo.

  Exemplo:

  ```
  double v[5];
  ```

- Logo, as seguintes sintaxes são equivalentes:
  - `*v` $\leftrightarrow$ `v[0]`
  - `v` $\leftrightarrow$ `&v[0]`

Equivalência

- Logo, são assinaturas equivalentes de funções:

```
void funcao(float v[], int sz)
```

ou

```
void funcao(float *v, int sz)
```

- Logo, são chamadas equivalentes às funções:

```
funcao(v, sz);
```

ou

```
funcao(&v[0], sz);
```

Possíveis operações

Considere os ponteiros `p`, `q` e o inteiro `i > 0`. Assim:

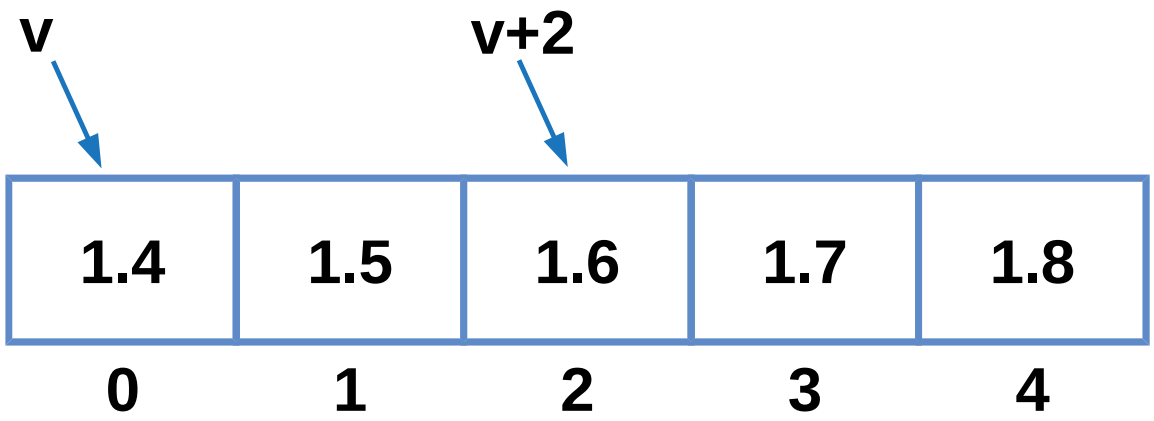
`p + i` → ponteiro `p` é deslocado `i` elementos para direita.

`p - i` → ponteiro `p` é deslocado `i` elementos para esquerda.

`|q - p|` → retorna a distância, em elementos, entre os ponteiros `p` e `q`.

Movendo um ponteiro

```
double v[5];
v[0] = 1.4; ↔ *v = 1.4;
v[2] = 1.6; ↔ *(v+2) = 1.6;
```



Exemplo 14 (Movendo Ponteiros)

[movendo_ponteiros.c]

```c
#include <stdio.h>

int main() {
    float v[] = {1.4, 1.5, 1.6, 1.7, 1.8};
    int sz = sizeof(v) / sizeof(float);

    // Posição inicial dos ponteiros
    float *p = v;
    float *q = &v[2];
    printf("*p = %g\t *q = %g\t q-p = %ld\n", *p, *q, q-p);

    // Movendo os ponteiros
    p += sz - 1;
    q--;
    printf("*p = %g\t *q = %g\t q-p = %ld\n", *p, *q, q-p);

    return 0;
}
```

O que será impresso?

Exemplo 14 (Movendo Ponteiros)

[movendo_ponteiros.c]

```c
#include <stdio.h>

int main() {
    float v[] = {1.4, 1.5, 1.6, 1.7, 1.8};
    int sz = sizeof(v) / sizeof(float);

    // Posição inicial dos ponteiros
    float *p = v;
    float *q = &v[2];
    printf("*p = %g\t *q = %g\t q-p = %ld\n", *p, *q, q-p);

    // Movendo os ponteiros
    p += sz - 1;
    q--;
    printf("*p = %g\t *q = %g\t q-p = %ld\n", *p, *q, q-p);

    return 0;
}
```

O que será impresso?

```
*p = 1.4    *q = 1.6    q-p = 2
*p = 1.8    *q = 1.5    q-p = -3
```

Aritmética de Ponteiros: + e -

- Cada vez que um ponteiro é incrementado, ele aponta para a posição de memória do próximo elemento do seu tipo base.
- Cada vez que é decrementado, ele aponta para a posição do elemento anterior.

Exemplos:

```
1  float v[5];
2  float *p;
3  p = v;          /* Ponteiro p está apontando para v[0] */
4  p = v + 2;      /* Ponteiro p está apontando para v[2] */
5  p--;            /* Ponteiro p está apontando para v[1] */
```

Percorrendo um arranjo

[percorrer_arranjo.c]

Exemplo 15 (Forma tradicional)

```c
#include <stdio.h>

int main() {
    int i, v[] = {1, 3, 5, 7, 9};
    int sz = sizeof(v) / sizeof(int);

    for (i = 0; i < sz; i++) {
        printf("%d\t", v[i]);
    }
    printf("\n");

    return 0;
}
```

Percorrendo um arranjo

[percorrer_arranjo2.c]

Exemplo 16 (Com o uso de ponteiros)

Se além de um ponteiro para o primeiro elemento tivermos um outro ponteiro para o último elemento, pode-se iterar de uma forma bem interessante, confira!

```
1  #include <stdio.h>
2
3  int main() {
4      int v[] = {1, 3, 5, 7, 9};
5      int sz = sizeof(v) / sizeof(int);
6      int *p, *q;
7
8      for (p = v, q = v + sz - 1; p <= q; p++) {
9          printf("%d\t", *p);
10     }
11     printf("\n");
12
13     return 0;
14 }
```

Quem é mais rápido:
Arranjo ou Ponteiro?

[func_arr.c; func_ptr.c]

Para responder esta questão, vamos analisar o código ***assembler*** gerado pelos programas abaixo. O programa da esquerda modifica o array diretamente usando seu índice, enquanto que o da direita usa a notação de ponteiros.

```c
#include <stdio.h>

int arr[20] = {0};

int func1(int i) {
   arr[i] = 10;
}

int main() {
   func1(9);
   for(int i = 0; i < 20; i++) {
      printf("%d ", arr[i]);
   }
   return 0;
}
```

```c
#include <stdio.h>

int arr[20] = {0};

int func2(int i) {
   *(arr+i) = 10;
}

int main() {
   func2(9);
   for(int i = 0; i < 20; i++) {
      printf("%d ", arr[i]);
   }
   return 0;
}
```

Quem é mais rápido:
Arranjo ou Ponteiro?

Vamos compilar os programas anteriores com a informação de *debug* e, em seguida, utilizar o *The GNU Project Debugger* (GDB)[2] para desmontar o código-objeto (*disassemble*) e comparar.

Exemplo:

```
$ gcc -g func_arr.c -o func_arr.out
```

```
$ gdb -q func_arr.out
```

Nota:

No **gdb**, pode-se utilizar comandos para executar, listar e desmontar o código-objeto[3].

[2] https://www.gnu.org/software/gdb

[3] https://www.gnu.org/software/gdb/documentation

Compilando e executando com *gdb disassemble*, `func1` retorna:

```
Dump of assembler code for function func1:
   0x0000000000400526 <+0>:    push   %rbp
   0x0000000000400527 <+1>:    mov    %rsp,%rbp
   0x000000000040052a <+4>:    mov    %edi,-0x4(%rbp)
   0x000000000040052d <+7>:    mov    -0x4(%rbp),%eax
   0x0000000000400530 <+10>:   cltq
   0x0000000000400532 <+12>:   movl   $0xa,0x601060(,%rax,4)
   0x000000000040053d <+23>:   nop
   0x000000000040053e <+24>:   pop    %rbp
   0x000000000040053f <+25>:   retq
End of assembler dump.
```

Desmontando `func2`, obtém-se:

```
Dump of assembler code for function func2:
   0x0000000000400526 <+0>:    push   %rbp
   0x0000000000400527 <+1>:    mov    %rsp,%rbp
   0x000000000040052a <+4>:    mov    %edi,-0x4(%rbp)
   0x000000000040052d <+7>:    mov    -0x4(%rbp),%eax
   0x0000000000400530 <+10>:   cltq
   0x0000000000400532 <+12>:   shl    $0x2,%rax
   0x0000000000400536 <+16>:   add    $0x601060,%rax
   0x000000000040053c <+22>:   movl   $0xa,(%rax)
   0x0000000000400542 <+28>:   nop
   0x0000000000400543 <+29>:   pop    %rbp
   0x0000000000400544 <+30>:   retq
End of assembler dump.
```

Exercícios de Aprendizagem

Exercício 1

Crie um vetor de inteiros com n posições (n informado pelo usuário) e preencha com números aleatórios entre 1 e 50. Em seguida, passe o vetor para a função **`square_vector`** como um ponteiro. Esta função deverá elevar todos os elementos do vetor ao quadrado. Imprima o vetor antes e após a chamada da função, cuja assinatura é dada abaixo:

```
void square_vector(int *v, int sz)
```

Exemplo de Funcionamento

```
Informe o tamanho do vetor: 10
-------------------------------------------------
Vetor original:
[    6    8   18   10   16   16   50   11   49   13 ]
Vetor ao quadrado:
[   36   64  324  100  256  256 2500  121 2401  169 ]
-------------------------------------------------
```

Exercícios de Aprendizagem

Exercício 2

Reimplemente a função **`troca`** apresentada no Exemplo 11, **sem uso** de variáveis auxiliares ou arranjos.

- Use apenas operações com ponteiros;
- Os valores de **`x`** e **`y`** devem ser informados pelo usuário;
- Trabalhe com valores de ponto flutuante (**`float`**).

Exemplo de Funcionamento

```
**************************************************
Função de troca de valores sem uso de variável auxiliar
**************************************************
Informe dois valores numéricos: x y = -3 5.7
Valores informados:   x = -3 e y = 5.7
Valores após a troca: x = 5.7 e y = -3
```

Exercícios de Aprendizagem

Exercício 3

Declare a string **s1**, abaixo, na função **main**, calcule o comprimento de **s1** como vetor (via **sizeof**) e como string (usando **strlen**, cabeçalho **string.h**).

```
char s1[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

Logo após, construa a função **strflip**:

```
void strflip(char *dest, const char *src)
```

para inverter **s1**. Faça a impressão do resultado como vetor e como string.

Exemplo de Funcionamento

```
s1:
   [ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z ]
   ABCDEFGHIJKLMNOPQRSTUVWXYZ
   sizeof = 27; strlen = 26
s2:
   [ Z Y X W V U T S R Q P O N M L K J I H G F E D C B A ]
   ZYXWVUTSRQPONMLKJIHGFEDCBA
```

Exercícios de Aprendizagem

Definição (Matriz Triangular)

Matrizes triangulares são matrizes quadradas, onde os elementos acima (ou abaixo) da diagonal principal são nulos. Podem ser classificadas em triangular superior ou inferior, de acordo com a posição desses elementos em relação à diagonal principal.

Exercício 4

Crie a função **`triangular_matrix`** para produzir uma matriz triangular superior $n \times n$, contendo valores inteiros aleatórios em um intervalo $[a, b]$.

Passos:

1. O usuário deve informar n, a e b;
2. A função deve apresentar a seguinte assinatura:

```
void triangular_matrix(int *M, int n)
```

3. Produza a matriz M de acordo com os parâmetros informados e imprima-a;
4. Mude os elementos abaixo da diagonal principal para zero;
5. Imprima a matriz triangular obtida.

Exercícios de Aprendizagem

Exercício 4

Exemplo de Funcionamento

```
*** Matriz Triangular de Inteiros ***
Informe a dimensão: n = 5
Informe a faixa de valores: [a, b] = -5 12
-----------------------------------------
Matriz de inteiros aleatórios entre [-5, 12]:
|   -5     6     7    -2    -4 |
|    5     6    -1    11     3 |
|    5    -2    12    -2    -1 |
|    6     0    -5     3     8 |
|    4    11     4    11    -2 |

Matriz triangular superior:
|   -5     6     7    -2    -4 |
|    0     6    -1    11     3 |
|    0     0    12    -2    -1 |
|    0     0     0     3     8 |
|    0     0     0     0    -2 |
```

Considerações Finais

- Nesta aula foram apresentados os principais conceitos relacionados a:
  - Ponteiros
  - Operadores e aritmética com ponteiros
  - Passagem de parâmetros
  - Retorno de funções
  - Arranjos $\times$ ponteiros

✓ *É importante rever os conceitos apresentados na aula e consultar a bibliografia sugerida sobre o assunto.*

Referências Bibliográficas I

Aguilar, L. J. (2008).

Programação em C++: Algoritmos, Estruturas de Dados e Objetos.
McGraw-Hill, São Paulo.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., e Stein, C. (2002).

Algoritmos: Teoria e Prática.
Elsevier, Rio de Janeiro.

Drozdek, A. (2009).

Estrutura de Dados e Algoritmos em C++.
Cengage Learning, São Paulo.

Forbellone, A. L. V. e Eberspacher, H. F. (2005).

Lógica de Programação: A Construção de Algoritmos e Estrutura de Dados.
Pearson Prentice Hall, São Paulo, 3 edition.

Kernighan, B. W. e Ritchie, D. M. (1988).

The C Programming Language.
Prentice-Hall, Englewood Cliffs, New Jersey, 2nd edition.

Knuth, D. E. (2005).

The Art of Computer Programming.
Addison-Wesley, Upper Saddle River, NJ, USA.

Referências Bibliográficas II

Pinheiro, F. d. A. C. (2012).

Elementos de Programação em C.
Bookman, Porto Alegre.

Pisani, P. H. (2018).

Programação Estruturada - Notas de Aula.
Universidade Federal do ABC (UFABC).
Disponível em: <http://professor.ufabc.edu.br/~paulo.pisani/>.

Sedgewick, R. (1998).

Algorithms in C: Parts 1-4, Fundamentals, Data Structures, Sorting, Searching.
Addison-Wesley, Boston, 3rd edition.

Szwarcfiter, J. L. e Markenzon, L. (1994).

Estruturas de Dados e Seus Algoritmos.
LTC, Rio de Janeiro.

Tenenbaum, A. A., Langsam, Y., e Augenstein, M. J. (1995).

Estruturas de Dados Usando C.
Makron Books, São Paulo.

Referências Bibliográficas III

Terra, R. (2014).

Linguagem C - Notas de Aula.

Universidade Federal de Lavras (UFLA).

Disponível em:
<http://professores.dcc.ufla.br/~terra/public_files/2014_apostila_c_ansi.pdf>. Acesso em: 2018-09-16.