

Curso: Ciência da Computação

Disciplina: Estruturas de Dados

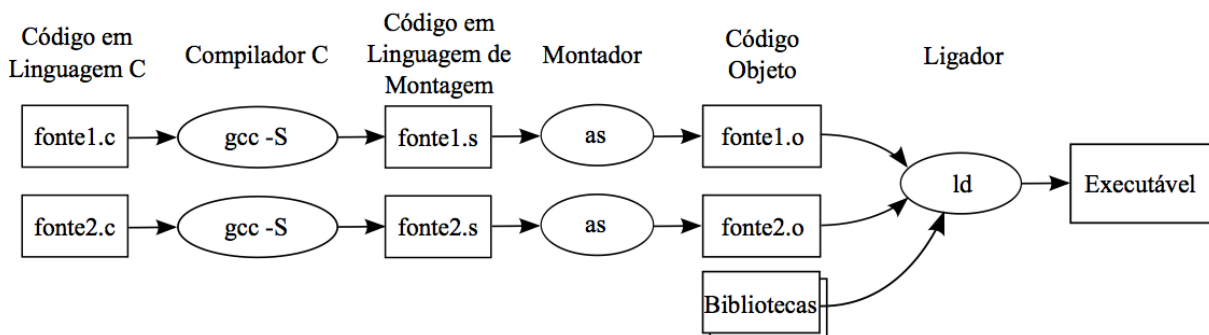
Profª Luciana Ap. Oliveira Betetto

Aula 02

1. Modularização

Um programa em pode ser dividido em vários arquivos fontes. Quando temos um arquivo com funções que representam apenas parte da implementação de um programa completo, denominamos esse arquivo de **módulo**. Assim, a implementação de um programa pode ser composta por um ou mais módulos.

No caso de um programa composto por vários módulos, cada um desses módulos deve ser compilado separadamente, gerando um arquivo objeto para cada módulo. Após a compilação de todos os módulos, uma outra ferramenta, denominada *ligador*, é usada para juntar todos os arquivos objeto em um único arquivo executável.



Para programas pequenos, o uso de vários módulos pode não se justificar. Mas para programas de médio e grande porte, a sua divisão em vários módulos é uma técnica fundamental, pois facilita a divisão de uma tarefa maior e mais complexa em tarefas menores e, provavelmente, mais fáceis de implementar e de testar. Além disso, um módulo com funções pode ser utilizado para compor vários programas, e assim poupar muito tempo de programação. Geralmente, um módulo agrupa vários tipos e funções com funcionalidades relacionadas, caracterizando assim uma finalidade bem definida.

2. Ponteiros

Quando se deseja manipular diretamente a variável, **ponteiros** devem ser usados como o recurso de acesso.

Ponteiros são muito úteis quando uma variável tem que ser acessada em diferentes partes de um programa. Neste caso, o código pode ter vários ponteiros espalhados por diversas partes do programa, “apontando” para a variável que contém o dado desejado. Caso este

dado seja alterado, não há problema algum, pois todas as partes do programa tem um ponteiro que aponta para o endereço onde reside o dado atualizado.

Ponteiros são variáveis que permitem armazenar um endereço de memória e modificar o que está armazenado naquele endereço.

A linguagem C implementa o conceito de ponteiro. O ponteiro é um tipo de dado como int, char ou float. Os ints guardam inteiros. Os chars guardam caracteres. Os floats guardam números de ponto flutuante. **Ponteiros guardam endereços de memória.** Quando você anota o endereço de um colega você está criando um ponteiro. O ponteiro é este seu pedaço de papel. Ele tem anotado um endereço. Qual é o sentido disto? Simples. Quando você anota o endereço de um colega, depois você vai usar este endereço para achá-lo. O C funciona assim. Você anota o endereço de algo numa variável ponteiro para depois usar.

Por meio deste endereço pode-se acessar a informação, dizendo que a *variável ponteiro* aponta para uma posição de memória. O maior problema em relação ao ponteiro é entender quando se está trabalhando com o seu valor, ou seja, o endereço, e quando se está trabalhando com a informação (conteúdo) apontada por ele.

Para declarar um ponteiro temos a seguinte forma geral:

```
tipo_do_ponteiro *nome_da_variável;
```

É o asterisco (*) que faz o compilador saber que aquela variável não vai guardar um valor mas sim um endereço para aquele tipo especificado. Exemplos de declarações:

Exemplos de declarações:

```
int *pt;
char *temp, *pt2;
```

O primeiro exemplo declara um ponteiro para um inteiro. O segundo declara dois ponteiros para caracteres. Eles ainda não foram inicializados (como toda variável do C que é apenas declarada). Isto significa que eles apontam para um lugar indefinido. Este lugar pode estar, por exemplo, na porção da memória reservada ao sistema operacional do computador. Usar o ponteiro nestas circunstâncias pode levar a um travamento do micro, ou a algo pior.

O ponteiro deve ser inicializado (apontado para algum lugar conhecido) antes de ser usado! Isto é de suma importância!

4.1. Operadores de Ponteiros: & e *

- **&** é um operador unário que devolve o endereço na memória do seu operando. (Um operador unário requer apenas um operando).
- ***** é um operador unário que devolve o valor da variável localizada no endereço que o segue.

O primeiro operador de ponteiro é **&**. Ele é um operador unário que devolve o endereço na memória de seu operando. Por exemplo: `m = &count;` põe o endereço na memória da variável `count` em `m`. Esse endereço é a posição interna da variável na memória do computador e não tem nenhuma relação com o valor de `count`. O operador **&** tem como significado *o endereço de*. O segundo operador é *****, que é o complemento de **&**. O ***** é um operador unário que devolve o valor da variável localizada no endereço que o segue. Por exemplo, se `m` contém o endereço da variável `count`: `q = *m;` coloca *o valor de* `count` em `q`. O operador ***** tem como significado *o conteúdo de*.

Lembrete: Cuidado para não confundir o operador de ponteiros (*****) como multiplicação na utilização de ponteiros e vice-versa.

Para atribuir um valor a um ponteiro recém-criado poderíamos igualá-lo a um valor de memória. Mas, como saber a posição na memória de uma variável do nosso programa? Seria muito difícil saber o endereço de cada variável que usamos, mesmo porque estes endereços são determinados pelo compilador na hora da compilação e realocados na execução. Podemos então deixar que o compilador faça este trabalho por nós. Para saber o endereço de uma variável basta usar o operador **&**. Veja o exemplo:

```
int count=10;
int *pt;
pt=&count; // retorna o endereço de count
```

Criamos um inteiro **count** com o valor 10 e um apontador para um inteiro **pt**. A expressão **&count** nos dá o endereço de `count`, o qual armazenamos em `pt`. Simples, não é? Repare que não alteramos o valor de `count`, que continua valendo 10.

A declaração de uma variável ponteiro é dada pela colocação de um asterisco (*****) na frente de uma variável de qualquer tipo. Na linguagem C, é possível definir ponteiros para os tipos básicos ou estruturas. A definição de um ponteiro não reserva espaço de memória para o seu valor e sim para o seu conteúdo (tipo de dado). Antes de utilizar um ponteiro, o mesmo deve ser inicializado, ou seja, deve ser colocado um endereço de memória válido para ser acessado posteriormente.

Veja esse exemplo (Celes, 2002): Quando escrevemos:

```
int a;
```

declaramos uma variável com nome `a` que pode armazenar valores inteiros. Automaticamente, reserva-se um espaço na memória suficiente para armazenar valores inteiros (geralmente 4 bytes).

Da mesma forma que declaramos variáveis para armazenar inteiros, podemos declarar variáveis que, em vez de servirem para armazenar valores inteiros, servem para armazenar valores de endereços de memória onde há variáveis inteiras armazenadas. C não reserva uma palavra especial para a declaração de ponteiros; usamos a mesma palavra do tipo com os nomes das variáveis precedidas pelo caractere *****.

Assim, podemos escrever:

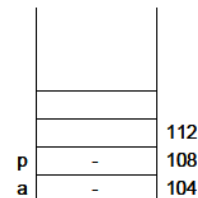
```
int *p;
```

Neste caso, declaramos uma variável com nome `p` que pode armazenar endereços de memória onde existe um inteiro armazenado.

Para exemplificar, vamos ilustrar esquematicamente, através de um exemplo simples, o que ocorre na pilha de execução. Consideremos o trecho de código mostrado na figura abaixo.

```
/*variável inteiro */
int a;

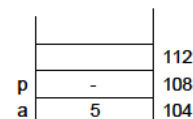
/*variável ponteiro p/ inteiro */
int *p;
```



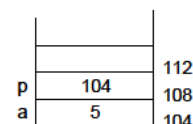
Efeito de declarações de variáveis na pilha de execução

Após as declarações, ambas as variáveis, `a` e `p`, armazenam valores "lixo", pois não foram inicializadas. Podemos fazer atribuições como exemplificado nos fragmentos de código da figura a seguir:

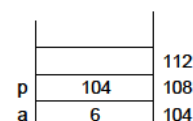
```
/* a recebe o valor 5 */
a = 5;
```



```
/* p recebe o endereço de a
   (diz-se p aponta para a) */
p = &a;
```

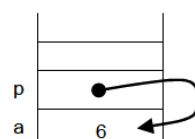


```
/* conteúdo de p recebe o valor 6 */
*p = 6;
```



Efeito de atribuição de variáveis na pilha de execução

Com as atribuições ilustradas na figura, a variável `a` recebe, indiretamente, o valor 6. Acessar `a` é equivalente a acessar `*p`, pois `p` armazena o endereço de `a`. Dizemos que `p` aponta para `a`, daí o nome ponteiro. Em vez de criarmos valores fictícios para os endereços de memória no nosso esquema ilustrativo da memória, podemos desenhar setas graficamente, sinalizando que um ponteiro aponta para uma determinada variável.



Representação gráfica do valor de um ponteiro

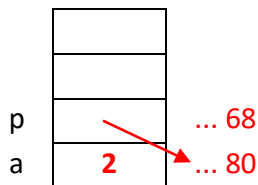
A possibilidade de manipular ponteiros de variáveis é uma das maiores potencialidades de C. Por outro lado, o uso indevido desta manipulação é o maior causador de programas que "voam", isto é, não só não funcionam como, pior ainda, podem gerar efeitos colaterais não previstos.

A seguir, apresentamos outros exemplos de uso de ponteiros. O código abaixo:

```
/* programa Aula2_Celes1.cpp */ // pág 48
#include <stdio.h>
int main ( void )
{
    int a;
    int *p;
    p = &a; // p recebe end de a
    *p = 2; // conteúdo de p
           recebe 2
    printf(" %d ", a);
    return 0;
}
```

```
2
-----
Process exited with return value 0
Press any key to continue . . .
```

imprime o valor 2.

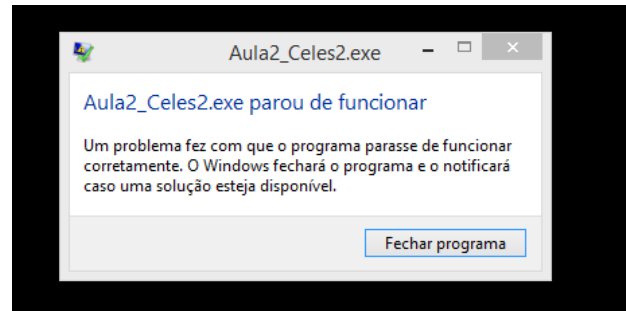


ACRESCENTAR:

```
printf("\n");
printf(" %d ", &p); // 6487568 – endereço de p
printf("\n");
printf(" %d ", &a); // 6487580 – endereço de a
printf("\n");
printf(" %d ", p); // 6487580 – o conteúdo de p é o endereço de a
return 0;
}
```

Agora, no exemplo abaixo:

```
/* programa Aula2_Celes2.cpp */
#include <stdio.h>
int main ( void )
{
    int a, b, *p;
    a = 2;
    *p = 3; // conteúdo de p recebe 3
    b = a + (*p);
    printf(" %d ", b);
    return 0;
}
```



cometemos um ERRO típico de manipulação de ponteiros. O pior é que esse programa, embora incorreto, às vezes pode funcionar. O erro está em usar a memória apontada por `p` para armazenar o valor 3. Ora, a variável `p` não tinha sido inicializada e, portanto, tinha armazenado um valor (no caso, endereço) "lixo". Assim, a atribuição `*p = 3;` armazena 3 num espaço de memória desconhecido, que tanto pode ser um espaço de memória não utilizado, e aí o programa aparentemente funciona bem, quanto um espaço que armazena outras informações fundamentais – por exemplo, o espaço de memória utilizado por outras variáveis ou outros aplicativos. Neste caso, o erro pode ter efeitos colaterais indesejados.

Um ponteiro pode ser utilizado de duas maneiras distintas. Uma maneira é trabalhar com o endereço armazenado no ponteiro e outro modo é trabalhar com a área de memória apontada pelo ponteiro. Quando se quiser trabalhar com o endereço armazenado no ponteiro, utiliza-se o seu nome sem o asterisco na frente. Sendo assim qualquer operação realizada será feita no endereço do ponteiro.

Como, na maioria dos casos, se deseja trabalhar com a memória apontada pelo ponteiro, alterando ou acessando este valor, deve-se colocar um asterisco antes do nome do ponteiro. Sendo assim, qualquer operação realizada será feita no endereço de memória apontado pelo ponteiro. Algumas demonstrações simples sobre a utilização de ponteiros:

Exemplo 1: o código `%p` usado na função `printf()` indica à função que ela deve imprimir um endereço.

```
/* programa Aula2_ponteiros1.cpp */
#include <stdio.h>
int main ()
{
    int num,valor;
    int *p;
    num=55;
    p=&num; /* p recebe endereço de num */
    valor=*p; /* Valor é igualado a num de uma maneira indireta */
    printf("\n\n%d\n",valor);
    printf("Endereco para onde o ponteiro aponta: %p\n",p);
    printf("Valor da variavel apontada: %d\n",*p);
    return(0);
}
```

```
55
Endereco para onde o ponteiro aponta: 0028FED4
Valor da variavel apontada: 55

-----
Process exited with return value 0
Press any key to continue . . .
```

p	55	
valor	55	
num	55	4

```

/* programa Aula2_ponteiro1.cpp */
#include <stdio.h>
int main ()
{
    int num,valor;
    int *p;
    num=55;
    p=&num;    /* Pega o endereco de num */
    valor=*p;   /* Valor é igualado a num de uma maneira indireta */
    printf ("\n\n%d\n",valor);
    printf ("Endereco para onde o ponteiro aponta: %p\n",p);
    printf ("Valor da variavel num: %d\n",*p); 55
    printf ("Valor da variavel valor: %d\n",valor); 55
    printf ("Valor da variavel num: %d\n",num); 55
    printf ("Endereco da variavel p: %p\n",&p); // endereço de p
    printf ("Endereco da variavel num: %p\n",&num); // endereço de num
    printf ("Endereco da variavel valor: %p\n",&valor); // endereço de valor
    return(0);
}

```

```

C:\Users\Public\Luciana\Unip\EstruturaDados\2017\Algoritmos\Aula2_ponteiro1.cpp - [Executing] - Dev-C++ 5.6.0
C:\Users\Public\Luciana\Unip\EstruturaDados\2017\Algoritmos\Aula2_ponteiro1.exe
55
Endereco para onde o ponteiro aponta: 0028FEDC
Valor da variavel num: 55
Valor da variavel valor: 55
Valor da variavel num: 55
Endereco da variavel p: 0028FED4
Endereco da variavel num: 0028FEDC
Endereco da variavel valor: 0028FED8
-----
Process exited with return value 0
Press any key to continue . . .
printf ("Endereco para onde o ponteiro aponta: %p\n",p);
printf ("Valor da variavel num: %d\n",*p);
printf ("Valor da variavel valor: %d\n",valor);
printf ("Valor da variavel num: %d\n",num);
printf ("Endereco da variavel p: %p\n",&p);
printf ("Endereco da variavel num: %p\n",&num);
printf ("Endereco da variavel valor: %p\n",&valor);
return(0);
}

```

Exemplo 2: altera o valor de **num** através do valor atribuído ao ponteiro.

/* programa Aula2_ponteiros2.cpp */

```
#include <stdio.h>
int main ()
{
    int num,*p;
    num=55;
    p=&num; /* Pega o endereço de num */

    printf ("\nValor inicial da variavel num: %d\n",num);
    *p=100; /* Muda o valor de num de uma maneira indireta */
    printf ("\nValor final da variavel num: %d\n",num);
    return(0);
}
```

```
valor inicial da variavel num: 55
valor final da variavel num: 100
-----
Process exited with return value 0
Press any key to continue . . .
```

Exemplo 3: demonstra a utilização de ponteiros para acesso à memória.

/* programa Aula2_ponteiros3.cpp */

```
#include <stdio.h>
int main (void)
{
    int *piValor; /* ponteiro para inteiro */
    int iVarivel = 27121975;
    piValor = &iVariavel; /* pegando o endereço de memória da variável */

    printf ("Endereco: %d\n", piValor); // é dado pelo compilador
    printf ("Valor : %d\n", *piValor);

    *piValor = 18011982; // pivalor =&ivariavel, então altera o valor
                        // da ivariavel
    printf ("Valor alterado: %d\n", iVarivel);
    printf ("Endereco : %d\n", piValor);
    return 0;
} // alterou o valor (conteúdo), mas o endereço é o mesmo
```

```
Endereco: 2686680
Valor : 27121975
Valor alterado: 18011982
Endereco : 2686680
-----
Process exited with return value 0
Press any key to continue . . .
```

3. Aritmética com ponteiros

Podemos fazer algumas operações aritméticas com ponteiros. A primeira, e mais simples, é igualar dois ponteiros. Se temos dois ponteiros p1 e p2 podemos igualá-los fazendo p1=p2 . **Repare que estamos fazendo com que p1 aponte para o mesmo lugar que p2** . Se quisermos que a variável apontada por p1 tenha o mesmo conteúdo da variável apontada por p2 devemos fazer *p1=*p2 .

As próximas operações, também muito usadas, são o incremento e o decremento. Quando incrementamos um ponteiro ele passa a apontar para o próximo valor do mesmo tipo para o qual o ponteiro aponta. Isto é, se temos um ponteiro para um inteiro e o incrementamos ele

passa a apontar para o próximo inteiro. Esta é mais uma razão pela qual o compilador precisa saber o tipo de um ponteiro: se você incrementa um ponteiro **char*** ele anda 1 byte na memória e se você incrementa um ponteiro **double*** ele anda 8 bytes na memória. O decremento funciona semelhantemente.

Estamos falando de *operações com ponteiros* e não de *operações com o conteúdo das variáveis* para as quais eles apontam. Por exemplo, para incrementar o conteúdo da variável apontada pelo ponteiro p, faz-se: `(*p)++`;

Outras operações aritméticas úteis são a soma e subtração de inteiros com ponteiros. Vamos supor que você queira incrementar um ponteiro de 15. Basta fazer: `p=p+15`; ou `p+=15`; E se você quiser usar o conteúdo do ponteiro 15 posições adiante: `*(p+15)`; A subtração funciona da mesma maneira.

Uma outra operação, às vezes útil, é a comparação entre dois ponteiros. Mas que informação recebemos quando comparamos dois ponteiros? Bem, em primeiro lugar, podemos saber se dois ponteiros são iguais ou diferentes (`==` e `!=`). No caso de operações do tipo `>`, `<`, `>=` e `<=` estamos comparando qual ponteiro aponta para uma posição mais alta na memória. Então uma comparação entre ponteiros pode nos dizer qual dos dois está "mais adiante" na memória. A comparação entre dois ponteiros se escreve como a comparação entre outras duas variáveis quaisquer: `p1>p2`

Há entretanto operações que você **não** pode efetuar num ponteiro. Você **não** pode dividir ou multiplicar ponteiros, adicionar dois ponteiros, adicionar ou subtrair float's ou double's de ponteiros.

Exercício:

a) Explique a diferença entre `p++`; `(*p)++`; `*(p++)`;

`p++`: incremento da variável p

`(*p)++`: incrementa o conteúdo da variável apontada pelo ponteiro p

`*(p++)`: incrementa p (como em `p++`) e acessa o valor encontrado na nova posição. Se em um vetor, esta expressão acessa o valor da posição imediatamente superior a armazenada em p antes do incremento.

b) O que quer dizer `*(p+10)`;

Acessa o conteúdo do ponteiro 10 posições adiante.

Exemplo 4:

```
#include <stdio.h>
int main (void)
{
    int x;
    x=25;
    int *p1, *p2;

    p1 = &x;
    p2 = p1;

    printf("Valor de x = %d\n", x);
    printf("Valor do endereço de x = %p\n", &x); /*escreve o endereço de x, não seu valor! */
    printf("Valor de p1 = %p\n", p1); /*escreve o endereço do ponteiro p1 */
    printf("Valor de p2 = %p\n", p2); /*escreve o endereço do ponteiro p2 */
    printf("Valor do conteúdo de p2= %d\n", *p2); /*escreve o valor 25
}
```

```
Valor de x = 25
Valor do endereço de x = 0028FED4
Valor de p1 = 0028FED4
Valor de p2 = 0028FED4

-----
Process exited with return value 0
Press any key to continue . . . _
```

As variáveis ponteiros sempre devem apontar para o tipo de dado correto. Por exemplo, quando um ponteiro é declarado como sendo do tipo int, o ponteiro assume que qualquer endereço que ele contenha aponta para uma variável inteira. Como C permite a atribuição de qualquer endereço a uma variável ponteiro, o fragmento de código seguinte compila sem nenhuma mensagem de erro (ou apenas uma advertência, dependendo do seu compilador), mas não produz o resultado esperado.

Exemplo 5:

```
#include <stdio.h>
int main (void)
{
    float x, y;
    int *p;

    p = &x; /* esse comando faz com que p (que é ponteiro para inteiro) aponte para um float. */
    y = *p; /*esse comando não funciona como esperado. */
}
```

Compiler (2) Resources Compile Log Debug Find Results Close			Message
Line	Col	File	
		C:\Users\Public\Luciana\Unip\EstruturaDados\2014\t...	In function 'int main()':
8	4	C:\Users\Public\Luciana\Unip\EstruturaDados\2014\test...	[Error] cannot convert 'float*' to 'int*' in assignment

Com uma variável do tipo ponteiro, é possível realizar operações de soma e subtração. Será somada ou diminuída no ponteiro a quantidade de endereços de memória relativos ao tipo do ponteiro. Por exemplo, um ponteiro para int ocupa 4 bytes, uma operação de soma neste ponteiro irá acrescentar 4 posições (unidades) na memória. No Exemplo 3 é visto como os

endereços de memória são manipulados através de aritmética de ponteiros, e a diferença entre o tipo **char** - que ocupa **1 byte** de memória - e o tipo **int** - que ocupa **4 bytes**.

Exemplo 6: aritmética com ponteiros.

```
/* programa Aula2_ponteiros6.cpp */
#include <stdio.h>
int main (void)
{
    int *piValor;
    int iValor;
    char *pcValor;
    char cValor;

    piValor = &iValor;
    pcValor = &cValor;
    printf ("Endereco de piValor = %p\n", piValor);
    printf ("Endereco de pcValor = %p\n", pcValor);
    piValor++; /* somando uma unidade (4 bytes) na memória */
    pcValor++; /* somando uma unidade (1 byte) na memória */

    printf ("Endereco de piValor = %p\n", piValor);
    printf ("Endereco de pcValor = %p\n", pcValor);
    return 0;
}
```

```
Endereco de piValor = 0028FED4
Endereco de pcValor = 0028FED3
Endereco de piValor = 0028FED8
Endereco de pcValor = 0028FED4

-----
Process exited with return value 0
Press any key to continue . . . _
```

4. Vetores e Matrizes como ponteiros

Vetores nada mais são do que ponteiros com alocação estática de memória, logo, todo vetor na linguagem C é um ponteiro, tal que o acesso aos índices do vetor podem ser realizados através de aritmética de ponteiros. Observe a figura 1 abaixo:

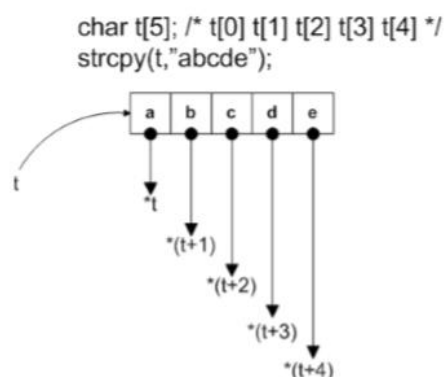


Figura 1: Vetor como Ponteiro em C

Existem 2 formas para se obter o endereço (ponteiro) do início do vetor ou matriz:

- `&t[0];`
- `t`

Sendo a segunda opção (`t`) a melhor, por ser mais simples. O endereço (ponteiro) para o primeiro elemento do vetor ou matriz é dado pelo nome do vetor sem colchetes.

Exemplo 7: mostra a utilização de um vetor como ponteiro.

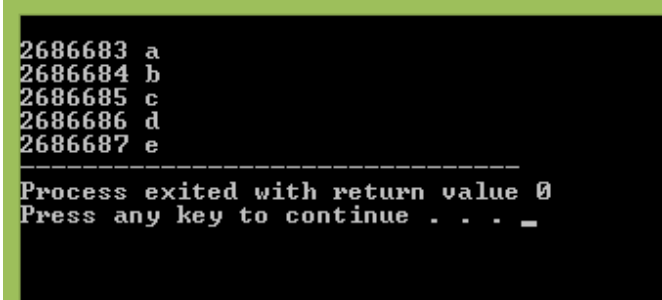
```
/* programa Aula2_ponteiros7.cpp */
#include <stdio.h>
#include <string.h>

int main(void)
{
    char t[5];

    strcpy(t,"abcde");

    printf("\n%d %c", t, *t);
    printf("\n%d %c", t+1, *(t+1));
    printf("\n%d %c", t+2, *(t+2));
    printf("\n%d %c", t+3, *(t+3));
    printf("\n%d %c", t+4, *(t+4));

    return 0;
}
```



```
2686683 a
2686684 b
2686685 c
2686686 d
2686687 e
-----
Process exited with return value 0
Press any key to continue . . . _
```

Exercício Extra:

```
#include <stdio.h>
void soma (int, int, int *);
int main (void)

{
    int iValorA;
    int iValorB;
    int iResultado;

    printf ("Entre com os valores:");

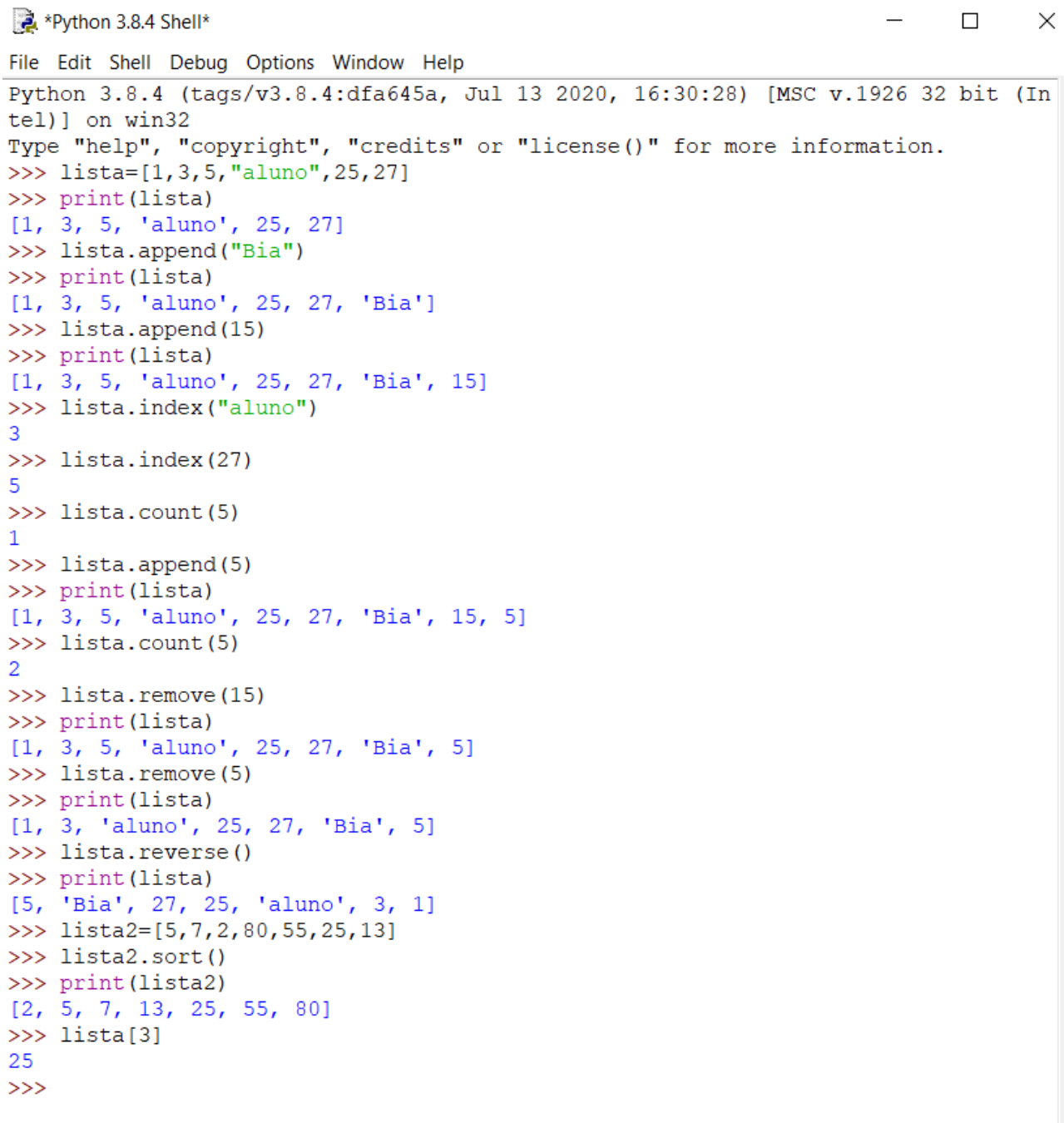
    scanf ("%d %d", &iValorA, &iValorB);
    printf("Endereco de iResultado = %d\n", &iResultado);

    soma (iValorA, iValorB, &iResultado);/* está sendo passado o endereço de memória da variável, qualquer alteração estará
                                         sendo realizada na memória */

    printf ("Soma : %d\n", iResultado);

    return 0;
}

void soma (int piValorA, int piValorB, int * piResultado)
{
    printf("Endereco de piResultado = %d\n", piResultado); /* o valor está sendo colocado diretamente na memória */
    *piResultado = piValorA + piValorB;
    return;
}
```



```

Python 3.8.4 Shell
File Edit Shell Debug Options Window Help
Python 3.8.4 (tags/v3.8.4:dfa645a, Jul 13 2020, 16:30:28) [MSC v.1926 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> lista=[1,3,5,"aluno",25,27]
>>> print(lista)
[1, 3, 5, 'aluno', 25, 27]
>>> lista.append("Bia")
>>> print(lista)
[1, 3, 5, 'aluno', 25, 27, 'Bia']
>>> lista.append(15)
>>> print(lista)
[1, 3, 5, 'aluno', 25, 27, 'Bia', 15]
>>> lista.index("aluno")
3
>>> lista.index(27)
5
>>> lista.count(5)
1
>>> lista.append(5)
>>> print(lista)
[1, 3, 5, 'aluno', 25, 27, 'Bia', 15, 5]
>>> lista.count(5)
2
>>> lista.remove(15)
>>> print(lista)
[1, 3, 5, 'aluno', 25, 27, 'Bia', 5]
>>> lista.remove(5)
>>> print(lista)
[1, 3, 'aluno', 25, 27, 'Bia', 5]
>>> lista.reverse()
>>> print(lista)
[5, 'Bia', 27, 25, 'aluno', 3, 1]
>>> lista2=[5,7,2,80,55,25,13]
>>> lista2.sort()
>>> print(lista2)
[2, 5, 7, 13, 25, 55, 80]
>>> lista[3]
25
>>>

```

Referências:

Ascencio, A.F.G. & Araujo, G.S. Estruturas de Dados. Editora Pearson, 2011.
 Celes, Waldemar & Rangel, José Lucas. Apostila de Estruturas de Dados. PUC RIO, 2002.
 Laureano, Marcos. Estrutura de Dados com Algoritmos e C. Editora Brasport, 2008.