



Ponteiros em C

Parte 1



Unesp – Campus de Guaratinguetá

Curso: Programação de Computadores

Prof. Aníbal Tavares

Profa. Cassilda Ribeiro

1



10.1 Ponteiros: *Definições*

- **Ponteiros** é um dos tópicos mais difíceis da linguagem C.
- A linguagem C é altamente dependente dos ponteiros. Para ser um bom programador em C é fundamental que se tenha um bom domínio deles.
- Ponteiros são tão importantes na linguagem C que você já os viu e nem percebeu, pois mesmo para se fazer uma introdução básica à linguagem C precisa-se deles.

2



10.1 Ponteiros: *Definições*

Organizando as informações com variáveis

- ☞ Na linguagem C, cada vez que é declarada uma variável (**por exemplo, `char ch1`**) é associado a esta variável um número hexadecimal (**por exemplo, `0022FF77`**) que é denominado de endereço.
- ☞ Esse número realiza a associação entre o nome da variável com o espaço físico que ela ocupa na memória do computador.
- ☞ Sem o uso de endereços não seria possível distinguir ou recuperar informações armazenadas na memória do computador.

3



10.2 Ponteiros: *Como Funcionam*

Como Funcionam os Ponteiros

Quando escrevemos: **`int a`**; declaramos uma variável com nome **a** que pode armazenar valores inteiros. Automaticamente, é reservado 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 **endereços de memória** onde há variáveis inteiras armazenadas.

Ponteiros são variáveis que guardam endereços de memória.

4



10.2 Ponteiros: *Como Funcionam*

Resumindo:

- Um **ponteiro** nada mais é que uma variável capaz de armazenar um número hexadecimal que corresponde a um endereço de memória de outra variável.
- É importante lembrar que a declaração de uma variável do tipo caractere, por exemplo, implica em reservar um espaço de memória para armazenar a informação correspondente:
- Para atribuir e acessar endereços de memória, a linguagem utiliza dois operadores unários: o **&** e o *****.

5



10.2 Ponteiros: *Como Funcionam*

- O operador unário **&** (“endereço de”), aplicado a variáveis, resulta no endereço da posição da memória reservada para a variável.
- O operador unário ***** (“conteúdo de”), aplicado a variáveis do tipo ponteiro, acessa o conteúdo do endereço de memória armazenado pela variável ponteiro.
- Suponha por exemplo que a variável **val**, armazene dentro dela um valor inteiro.
- Quando ela é declarada, o compilador reservará automaticamente um espaço na memória (caixinha) para ela.

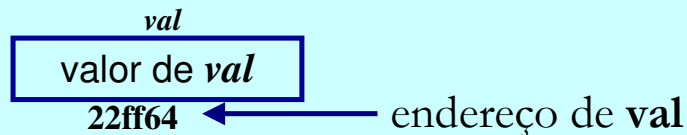
6



10.2 Ponteiros: *Como Funcionam*

- O conteúdo desta variável pode ser acessado através do nome ***val***, mas este conteúdo também pode ser acessado através do seu endereço na memória.
- O endereço da localização de memória da variável ***val*** pode ser determinado pela expressão ***&val***.

Ex: `int val;`



Quando a variável ***val*** é declarada, o que ocorre, na verdade, é que o espaço de memória, cujo endereço é **22ff64**, fica disponível para ocupar um inteiro:

PS: Observe que o endereço da variável é um número hexadecimal.

7



10.2 Ponteiros: *Como Funcionam*

Vamos agora atribuir o endereço da variável ***val*** a uma outra variável, ***pval***. Então:

pval = &val;

Essa nova variável é chamada de **ponteiro para *val***, pois ela aponta para a localização onde ***val*** está armazenada na memória.

- Lembre-se que ***pval*** armazena o endereço de ***val*** e não o seu valor

8



10.2 Ponteiros: *Como Funcionam*

- Assim como as demais variáveis, um ponteiro também tem **tipo**.
- No C quando declaramos ponteiros nós informamos ao compilador para que tipo de variável vamos apontá-lo. Um ponteiro **int** aponta para um **inteiro**, isto é, **guarda o endereço de um inteiro**.

■ Os ponteiros são utilizados para alocação dinâmica, e podem substituir matrizes com mais eficiência. Eles também fornecem um modo de se passar informações entre uma função e seu ponto de referência. Eles fornecem uma maneira de retornar múltiplos dados de uma função através dos parâmetros.

9



10.2 Ponteiros: *Como Funcionam*

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

Isto é de suma importância!

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?

10



10.2 Ponteiros: *Como Funcionam*

■ Seria muito difícil saber o endereço de cada variável que usamos, mesmo porque estes endereços são determinados pelo compilador na hora da compilação e re-allocados na execução.

- Podemos então deixar que o compilador faça este trabalho por nós.
- Para saber o endereço de uma variável basta usar o operador **&**.

O ponteiro também pode ser inicializado com o valor nulo, para tanto basta fazermos: `int *p=NULL;`

11



10.3 Ponteiros: *Declarando e Utilizando Ponteiros*

Declarando e Utilizando Ponteiros

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:

```
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.

12



10.3 Ponteiros: *Declarando e Utilizando Ponteiros*

Veja o exemplo:

```
int count=10;
int *pt1, *pt2=NULL;
pt1=&count;
```

Criamos um inteiro **count** com o valor 10 e dois apontadores para um inteiro **pt1** e **pt2**. A expressão **&count** nos dá o endereço de **count**, o qual armazenamos em **pt1**. **pt2** foi inicializado com **nulo**. Simples, não é?

→ Repare que **não** alteramos o valor de **count**, que continua valendo 10.

Como foi colocado um endereço em **pt1** e **pt2**, ele está agora "**liberado**" para ser usado. Pode-se, por exemplo, alterar o valor de **count** usando **pt1**. Para tanto vamos usar o operador "**inverso**" do operador **&**. É o operador *****.

■ No exemplo acima, uma vez que fizemos **pt1=&count** a expressão ***pt1** é equivalente ao próprio **count**. Isto significa que, para se mudar o valor de **count** para 12, basta fazer ***pt=12**.

13



10.3 Ponteiros: *Declarando e Utilizando Ponteiros*

Exemplo 1: considere o programa simples de utilização de ponteiros abaixo:

```
#include <stdlib.h>
#include <stdio.h>
main()
{ int *px1;
  float *px2;
  char *px4;
  double *px3;
  int i=1;
  float f= 0.3;
  double d= 0.005;
  char c = '*';
  px1=&i;
  px2=&f;
  px3=&d;
  px4=&c;
```

```
printf("valores:\t\t i=%d
\t f=%f \t d=%f \t c=%c
\n\n", i, f, d, c);
printf("Enderecos:\t\t &i=%x
&f=%x &d=%x &c=%x\n\n",
&i, &f, &d, &c);
printf("Valores dos ponteiros:
px1= %x px2=%x px3=%x\
px4=%x\n\n", px1, px2, px3,
px4);
system("PAUSE");
}
```

OBS: O endereço de uma variável pode ser impresso ou lido com **printf** ou **scanf** a partir das tags **%x** ou **%X**.

14



10.3 Ponteiros: *Declarando e Utilizando Ponteiros*

```
C:\users\Cassilda\DMA_66\FEG_CURSOS\Curso_C\Curso_C_Cassilda2008\Progra...
valores:          i=1      f=0.300000      d=0.005000      c=*
Enderecos:         &i=22ff64    &f=22ff60    &d=22ff58    &c=22ff57
Valores dos ponteiros: px1= 22ff64  px2=22ff60  px3=22ff58  px4=22ff57
Pressione qualquer tecla para continuar. . .
```

- Esse programa exibe valores e endereços associados a quatro tipos de variáveis: variável inteira i, variável real f, variável de dupla precisão d, e uma variável caractere c.
- O programa também utiliza px1, px2, px3 e px4, que representam respectivamente os endereços de i, f, d, c.
- A primeira linha exibe os valores das variáveis. A segunda linha, seus endereços conforme assinalado pelo compilador, e a terceira linha exibe o endereço representado pelas expressões ponteiros.

15



10.3 Ponteiros: *Declarando e Utilizando Ponteiros*

Exemplo 2: O programa a seguir imprime o valor de i, o endereço da variável i, o valor armazenado na variável ponteiro p e o endereço de p.

```
#include <stdlib.h>
#include <stdio.h>
main()
{int *p, i =4;
  p = &i;
  printf("  i = %d \n",i);
  printf(" &i = %x \n",&i);
  printf("  p = %p \n",p);
  printf(" &p = %X \n",&p);
  printf("\n Conteudo do endereco apontado por p: ");
  printf("%d\n",*p; //conteúdo do endereço apontado por p
  printf("\n\n");
  system("PAUSE");
}
```

16




10.3 Ponteiros: *Declarando e Utilizando Ponteiros*

Ao executar o programa do Exemplo 2 teremos:

```
C:\users\Cassilda\DMA_66\FEG_CURSOS\Curso_C\Curso_C_Cassilda2008\Programas\Pont...
i = 4
&i = 22ff70
p = 0022FF70
&p = 22FF74

Conteudo do endereco apontado por p: 4


Pressione qualquer tecla para continuar. . .
```

 **LEMBRE-SE** que quando queremos acessar o conteúdo do endereço para o qual o ponteiro aponta fazemos ***p**. Isto é, usamos o operador "*inverso*" do operador & que é o operador *.

17



10.3 Ponteiros: *Declarando e Utilizando Ponteiros*

 Já vimos que o endereço das variáveis são números HEXADECIMAIS, contudo aqui, para manter a clareza, mas sem perda de generalidade, em todos os exemplos descritos a seguir, os endereços serão representados por números na base 10.

Exemplo 3: O programa a seguir atribui o valor 55 para a variável **num** e em seguida atribui o endereço de num para o ponteiro **p**. A seguir é atribuído um valor para a variável **valor**, através do ponteiro **p**.

18



10.3 Ponteiros: *Declarando e Utilizando Ponteiros*

```
//Exemplo 3
#include <stdlib.h>
#include <stdio.h>
main ()
{ int num, valor;
  int *p;
  num=55;
  p=&num; // Pega o endereço de num
  valor=*p; //valor é igualado a num de maneira
            //indireta
  printf ("\n\n%d\n", valor);
  printf ("Endereco para onde o ponteiro aponta:
%d\n", p);
  printf ("Valor da variavel apontada: %d\n", *p);
  system("pause");
}
```

19



10.3 Ponteiros: *Declarando e Utilizando Ponteiros*

OBS: Neste exemplo, a variável **valor** recebeu o valor **55** através do ponteiro **p** (endereço da variável **num**) e do operador *****. Isto é, através da linha de comando: **valor=*p**.

```
C:\users\Cassilda\DMA_66\FEG_CURSOS\Curso_C\Curso_C_Cassilda2008\Programas\Ponteir...
55
Endereco para onde o ponteiro aponta: 2293620
Valor da variavel apontada: 55
Pressione qualquer tecla para continuar. . .
```

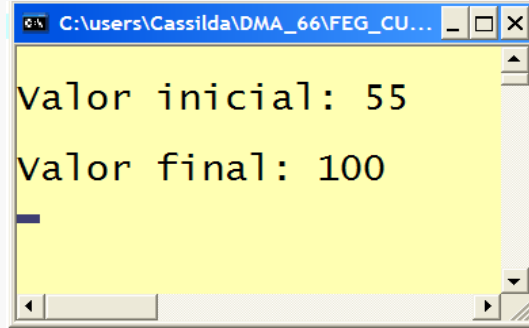
20



10.3 Ponteiros: *Declarando e Utilizando Ponteiros*

```
//Exemplo 4
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>

int main ()
{
    int num, *p;
    num=55;
    p=&num; /* Pega o endereço de num */
    printf ("\nValor inicial: %d\n", num);
    *p=100; // Muda o valor de num de uma maneira
    //indireta
    printf ("\nValor final: %d\n", num);
    getch(); //para a tela de execução. É usado com
    a biblioteca conio.h
}
```



21



10.3 Ponteiros: *Declarando e Utilizando Ponteiros*

Resumindo:

- Operador (*) é chamado de Operador de dereferência.
- Na linguagem C, o operador de dereferencia (*) além de servir para declarar que um variável é do tipo ponteiro serve também para saber qual o valor contido no endereço armazenado (apontado) por uma variável do tipo ponteiro

22



10.4 Ponteiros: *Operações Aritméticas*

- ➡ Podemos fazer algumas operações aritméticas com ponteiros.
- ➡ A operação mais simples, é igualar dois ponteiros. Se temos dois ponteiros **p1** e **p2** podemos igualá-los fazendo **p1=p2**.
- ➡ Observe que estamos fazendo **p1** apontar 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**.
- ➡ Depois que se aprende a usar os dois operadores (**&** e *****) fica fácil entender operações com ponteiros.

23



10.4 Ponteiros: *Operações Aritméticas*

- ➡ Um tipo de operação muito usada, é o incremento e o decremento (por exemplo: **p+1** e **p-1**).
- ➡ Quando incrementamos um ponteiro **p** 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. Supondo que **p** é um ponteiro, as operações podem ser escritas como:

p++;

p--;

24

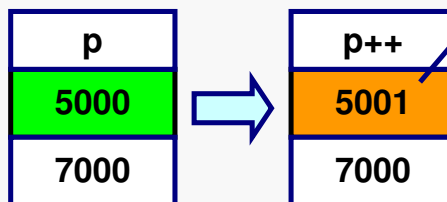


10.4 Ponteiros: *Operações Aritméticas*

Definição – Aritmética de Ponteiros

Uma variável do tipo ponteiro está sempre associada a um tipo. Ou seja, um ponteiro para um dado tipo **t** endereça o número de bytes que esse tipo **t** ocupa em memória, isto é, endereça **sizeof(t)** bytes. Se um ponteiro para uma variável do tipo **t**, que armazena o endereço **x**, for incrementada através do operador **++**, automaticamente este ponteiro passará a ter o valor **x + sizeof(t)**.

```
char ch, *p;  
ch = 'a';  
p = &ch;  
p++;
```



Tipo char só ocupa 1 byte !

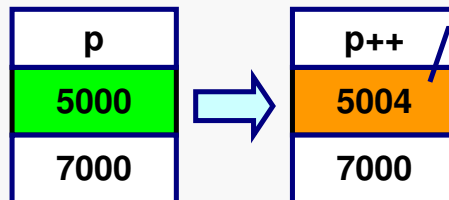
25



10.4 Ponteiros: *Operações Aritméticas*

```
int x, *p;  
x = 10;  
p = &x;  
p++;
```

Memória



Tipo int ocupa 4 bytes !

Em Detalhes

Espaço de 4 bytes para armazenar a variável x tipo int.

A operação **p++** percorre **sizeof(tipo p)** bytes !

p				p++			
x	x	x	x	?	?	?	?
5000	5001	5002	5003	5004	5005	5006	5007

26



10.4 Ponteiros: *Operações Aritméticas*

//Exemplo 5

```
main()
{ int *px1;
  float *px2;
  char *px4;
  double *px3;
  int i=1;
  float f= 0.3;
  double d= 0.005;
  char c = '*';
  px1=&i;
  px2=&f;
  px3=&d;
  px4=&c;
  printf("Valores variaveis antes ++:i=%d f=%f d=%f
        c=%c\n\n",i,f,d,c);
```

27



10.4 Ponteiros: *Operações Aritméticas : Exemplo 5*

```
printf("Endereco variaveis antes ++:i=%d &f=%d
      &d=%d &c=%d\n\n",&i,&f,&d,&c);
printf("Valores ponteiros antes ++:px1=%d px2=%d
      px3=%d px4=%d \n\n",px1, px2,px3,px4);
printf("Endereco ponteiros antes ++: &px1=%d
      &px2=%d &px3=%d &px4=%d\n\n",&px1,&px2,&px3,
      &px4);
puts("");
puts("");
px1++;
px2++;
px3++;
px4++;
printf("valores var depois ++: i=%d f=%f d=%f
      c=%c\n\n",i,f,d,c);
```

28



10.4 Ponteiros: *Operações Aritméticas: Exemplo 5*

```
printf("Enderecos var depois ++:&i=%d&f=%d &d=%d  
      &c=%d\n\n", &i, &f, &d, &c);  
printf("Valores dos ponteiros depois ++:px1=%d  
      px2=%d px3=%d px4=%d\n\n", px1, px2, px3, px4);  
printf("Endereco ponteiros depois ++:&px1=%d  
      &px2=%d &px3=%d &px4=%d\n\n", &px1, &px2, &px3,  
      &px4);  
system("PAUSE");  
}
```

29



10.4 Ponteiros: *Operações Aritméticas: Exemplo 5*

```
C:\users\Cassilda\FEG_CURSOS\Curso_C\Curso_C_Cassilda2008\Programas\Ponteiros\Proj01_Ponteiro....  
Valores variaveis antes ++: i=1          f=0.300000 d=0.005000 c=*  
Endereco variaveis antes ++: i=2293604   &f=2293600   &d=2293592   &c=2293591  
Valores ponteiros antes ++: px1=2293604  px2=2293600  px3=2293592  px4=2293591  
Endereco ponteiros antes ++: &px1=2293620 &px2=2293616 &px3=2293608 &px4=2293612  
  
valores var depois ++:      i=1          f=0.300000 d=0.005000 c=*  
Enderecos var depois ++:    &i=2293604   &f=2293600   &d=2293592   &c=2293591  
Valores ponteiros depois ++:px1=2293608  px2=2293604  px3=2293600  px4=2293592  
Endereco ponteiros depois ++:&px1=2293620 &px2=2293616 &px3=2293608 &px4=2293612  
  
Pressione qualquer tecla para continuar. . .
```

30



10.4 Ponteiros: *Operações Aritméticas: Exemplo 5*

- ➡ Na primeira linha está impressa o valor das variáveis *i, f, d, c*.
- ➡ Na segunda linha está impressa o endereço das variáveis através do operador & antes da operação aritmética com o ponteiro.
- ➡ Na terceira linha está impressa o valor do ponteiro, antes da operação aritmética com o ponteiro.
- ➡ Na quarta linha está impressa o endereço do ponteiro antes da operação aritmética com o ponteiro.
- ➡ Na quinta linha está impressa o valor das variáveis *i, f, d, c*, depois da operação aritmética com o ponteiro.

31



10.4 Ponteiros: *Operações Aritméticas: Exemplo 5*

- ➡ Na sexta linha está impressa o endereço das variáveis através do operador &, depois da operação aritmética com o ponteiro.
- ➡ Na sétima linha está impressa o valor do ponteiro, depois da operação aritmética com o ponteiro.
- ➡ Na oitava linha está impressa o endereço do ponteiro, depois da operação aritmética com o ponteiro.

Observe que só ocorreu alteração nos valores da sétima linha onde está impressa o conteúdo armazenado no ponteiro, depois da operação aritmética com o ponteiro.

O valor de `px1` passou de `px1= 2293604` (linha 3) para `px1=2293608` porque `px1` é um ponteiro para inteiro e o numero inteiro ocupa 4 bytes de memória.

32



10.4 Ponteiros: *Operações Aritméticas: Exemplo 5*

O valor de px2 passou de px2=2293600 para px2=2293604 22ff64 porque px2 é um ponteiro para float que ocupa 4 bytes de memória.


O valor de px3 passou de px3=2293592 para px3=2293600 porque px3 é um ponteiro para double que ocupa 8 bytes de memória.

O valor de px4 passou de px4=2293591 para px4=2293592 porque px4 é um ponteiro para char que ocupa 1 byte de memória.


33



10.4 Ponteiros: *Operações Aritméticas*


 **LEMBRE-SE** que 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. Suponha que você queira incrementar um ponteiro de 15. Basta fazer:

`p=p+15; ou p+=15;`

 Se você quiser usar o conteúdo do ponteiro 15 posições adiante basta fazer **`*(p+15);`**

 A subtração funciona da mesma maneira que a adição.

34



10.4 Ponteiros: *Operações Aritméticas*

☞ Uma outra operação que pode ser feita com ponteiros é a comparação entre dois ponteiros.

☞ Mas, que informação recebemos quando comparamos dois ponteiros?

Ao se comparar dois ponteiros podemos saber se eles são iguais ou diferentes (`==` e `!=`).

No caso de operações do tipo `>`, `<`, `>=` e `<=` estamos comparando qual ponteiro aponta para uma posição mais alta *na memória*.

☞ Uma comparação entre ponteiros pode nos dizer qual dos dois está "**mais adiante**" na memória.

35



10.4 Ponteiros: *Operações Aritméticas*

☞ A comparação entre dois ponteiros se escreve como a comparação entre outras duas variáveis quaisquer, isto é:

`p1 > p2`

☞ Há operações que *não* podem ser feitas num ponteiro. São elas:

- 1- dividir ou multiplicar ponteiros,
- 2- adicionar dois ponteiros,
- 3 - adicionar ou subtrair **floats** ou **doubles** de ponteiros.

O exemplo 5 a seguir mostra algumas operações com ponteiros

36



10.4 Ponteiros: *Operações Aritméticas* - Exemplo 6

```
#include <stdlib.h>
#include <stdio.h>
main()
{
    int x=5, *px = &x;
    double y=5.0, *py = &y;
    printf("%d %ld\n", x, px);
    printf("%d %ld\n", x+1, (long) (px+1));
    printf("%f %ld\n", y, (long) py);
    printf("%f %ld\n", y+1, (long) (py+1));
    system("PAUSE");
}
```

☞ Na primeira linha está impressa o valor da variável **x** e o seu endereço convertido para o decimal **long int**, pois está sendo usado no printf o (**%ld**) para imprimir **px**.

37



10.4 Ponteiros: *Operações Aritméticas* Exemplo 6

☞ Na segunda linha está sendo impressa o valor da variável **x** acrescida de uma unidade e o endereço mais próximo (**px+1**) do endereço de **x**. Aqui também o endereço está sendo impresso em decimal, pois estamos utilizando o (**%ld**) para imprimir (**px+1**)

☞ Como a variável **x** é do tipo **int**, ela ocupa **4 bytes** de memória. Então o endereço mais próximo de **x** é o **2293624**.

☞ Na terceira linha está sendo impressa o valor da variável **y** e o seu endereço, também em decimal.

☞ Na quarta linha está impressa o valor da variável **y** acrescida de uma unidade e o endereço mais próximo (**py+1**) do endereço de **y**.

38



10.4 Ponteiros: *Operações Aritméticas Exemplo 6*

```
C:\users\Cassilda\DMA_66\FEG_CURSOS\Curso_C\Curso_C_Cassilda2008\Programas\Pont...
5    2293620
6    2293624
5.000000    2293608
6.000000    2293616
Pressione qualquer tecla para continuar. . .
```

👉 Observe na primeira linha que o endereço da variável x é **2293620**.

👉 Observe na segunda linha que o endereço mais próximo ($px+1$) do endereço de x é: **2293624**. Isso porque o x é do tipo `int` e ele ocupa **4 bytes** de memória.

👉 O mesmo pode ser observado nas linhas 3 e 4 com relação ao endereço de y e seu endereço mais próximo $py+1$. O endereço de y é **2293608** e o endereço mais próximo é **2293616**, porque y ocupa **8 bytes** de memória.

39



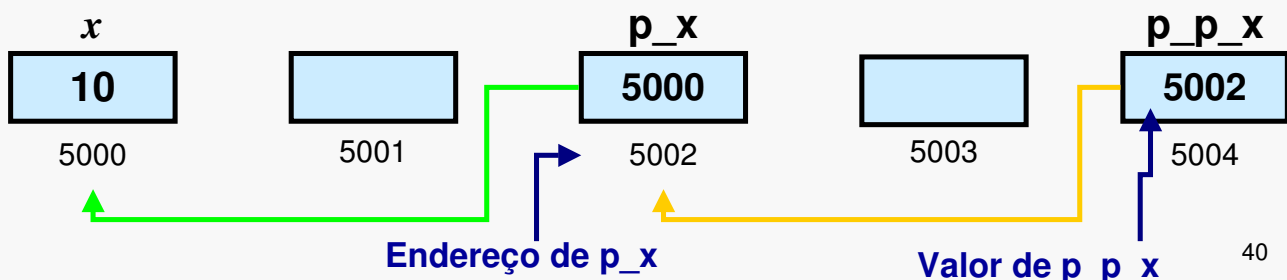
10.4 Ponteiros: *Operações Aritméticas*

Definição – Ponteiros para Ponteiros (tipo `**ponteiro`)

👉 Uma vez que as variáveis do tipo ponteiro ocupam um endereço de memória é possível criar uma nova variável para também armazenar este endereço. Ou seja, é possível criar ponteiros para ponteiros.

```
int x, *p_x, **p_p_x;
x = 10;
```

```
p_x = &x;
p_p_x = &p_x;
```



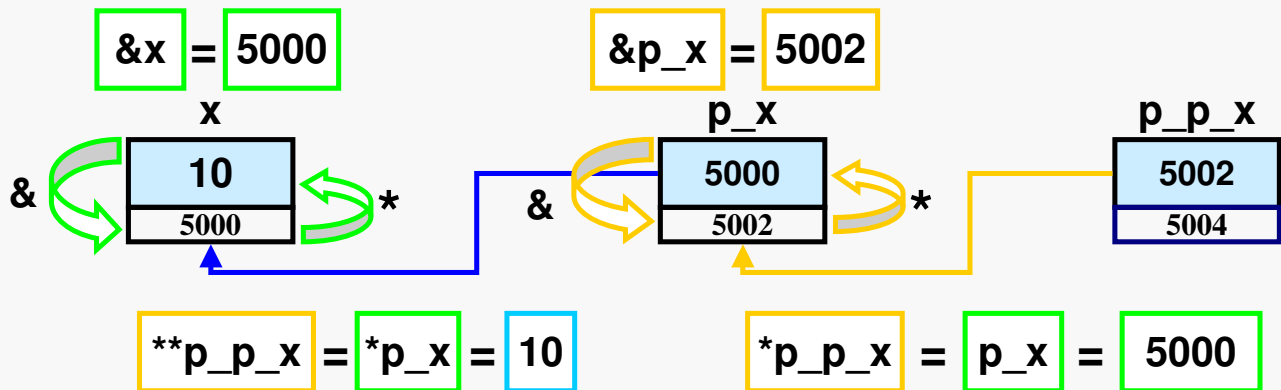
40



10.4 Ponteiros: Operações Aritméticas

👉 O esquema a seguir ilustra a utilização de ponteiros, o operador de endereço & (que obtém o endereço de memória de uma variável) e o operador de dereferência * (que obtém o conteúdo de um endereço de memória).

👉 O resultado da expressão `**p_p_x` é mais facilmente obtido se obtido em etapas: `*(*(p_p_x))` → `*(5002)` → `*(5000)` → 10.



41

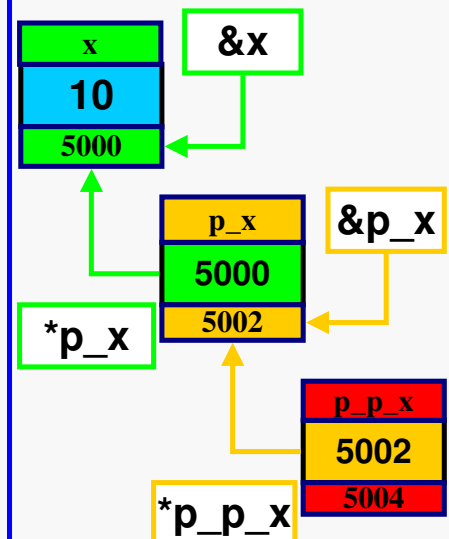


10.4 Ponteiros: Operações Aritméticas - Exemplo 7

Exemplo 7: Utilizando ponteiros para ponteiros e os operadores de endereço (&) e dereferência (*).

```
//Exemplo 7
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
main()
{
    int x, *p_x, **p_p_x;
    p_x = &x; p_p_x = &p_x;
    printf("Conteudo de x = %d\n", x);
    printf("Endereco de x = %x\n", p_x);
    printf("Endereco de p = %p\n", p_p_x);
    printf("Conteudo de x = %d\n", *p_x);
    printf("Conteudo de x = %d\n", **p_p_x);
    printf("Endereco de p_x = %x\n", &p_x);
    getch();
}
```

Em termos de memória



42



10.4 Ponteiros: *Operações Aritméticas* - Exemplo 7

Executando este programa tem-se:

```
C:\users\Cassilda\DMA_66\FEG_CURSOS\Curso_C\Curso_C_Cassilda2008\Programas\Ponteiro...
Conteudo de x = 2
Endereco de x = 22ff74
Endereco de p = 0022FF70
Conteudo de x = 2
Conteudo de x = 2
Endereco de p_x = 22ff70
```

43



10.5 Ponteiros – *Relação entre Ponteiros e Vetores*

Vetores como ponteiros

➤ Ponteiros e vetores têm uma ligação muito forte.

Vamos então dar agora uma idéia de como o C trata vetores.

Vimos que um vetor é declarado da seguinte forma:

tipo_da_variável nome_da_variável [tam];

Quando isto é feito, o compilador C, calcula o tamanho, em bytes, necessário para armazenar o vetor. Este tamanho é:

tam x tamanho_do_tipo

44



10.5 Ponteiros – Relação entre Ponteiros e Vetores

- A seguir, o compilador aloca o número de bytes que ele calculou, em um espaço livre de memória.
- O *nome do vetor* declarado é na verdade *um ponteiro para o tipo de variável do vetor*.
- Este conceito é fundamental porque, uma vez que foi alocado na memória o espaço para o vetor, o compilador pega o nome da variável (que é um ponteiro) e aponta para o *primeiro* elemento do vetor.
- Além do mais, quando um vetor é declarado seus elementos são alocados em espaços de memória vizinhos.

45



10.5 Ponteiros – Relação entre Ponteiros e Vetores

- Então o nome de um vetor equivale ao endereço do primeiro elemento do vetor, ou seja, se um vetor possui nome `vet`, então, `vet` equivale à `&vet[0]` (`vet ↔ &vet[0]`).
- Em outras palavras se `vet` é um vetor, então o endereço do primeiro elemento desse vetor pode ser expresso como `&vet[0]` ou simplesmente `vet`.
- Além disso, o endereço do segundo elemento desse vetor pode ser expresso como `&vet[1]` ou simplesmente `(vet+1)`, e assim por diante.

46



10.5 Ponteiros – Relação entre Ponteiros e Vetores

- Então, endereço do $(i+1)$ ésimo elemento do vetor pode ser expresso por $\&\text{vet}[i]$ ou por $(\text{vet}+i)$.
- Observe então que temos dois modos de escrever o endereço de qualquer elemento do vetor.
- Vale notar que a expressão $(\text{vet}+i)$ é um tipo muito especial e incomum, porque não estamos adicionando valores numéricos uma vez que vet representa um endereço enquanto i representa uma quantidade inteira.
- A expressão $(\text{vet}+i)$ esta especificando um endereço que é um numero (i) de células de memória depois do endereço o primeiro elemento do vetor. Além do mais, vet é o nome do vetor cujos elementos podem ser caracteres, inteiro, reais, etc.

47



10.5 Ponteiros – Relação entre Ponteiros e Vetores

- Como $\&\text{vet}[i]$ e $(\text{vet}+i)$ representam o endereço do i -ésimo elemento de vet , é razoável que $\text{vet}[i]$ e $\&(\text{vet}+i)$ representem o conteúdo deste endereço, isto é o valor do i -ésimo elemento de vet .

➤ Deste modo a expressão *nome_da_variável[índice]* **é absolutamente equivalente a** $\&(\text{nome_da_variável}+\text{índice})$

O exemplo 7 a seguir ilustra este fato.

48



10.5 Ponteiros – Relação entre Ponteiros e Vetores

Exemplo 8 – Vetor+Ponteiro

//Exemplo 8

```
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
main()
{
    int i, *ptr, v[5];
    ptr = &v[0]; //ou ptr = &v;
    // Usando indices para acessar
    //vet.
    printf("\n Usando v[i]:\n");
    for (i=0; i < 5; i++)
        scanf("%d", &v[i]);
    for (i=0; i < 5; i++)
    {
        printf("v[%d]=%d &v[%d]=%x",
            i, v[i], i, &v[i]);
        printf("\n");
    }
}
```

```
// Usando ponteiros (v+i).
printf("\n Usando ponteiro
(v+i):\n");
for (i=0; i < 5; i++)
{
    scanf("%d", (v+i));
    printf(" i= %d, *(v+%d)
= %d (v+%d)= %x", i, i,
*(v+i), i, (v+i));
    printf("\n");
}
// Usando ponteiros ptr.
printf("\n Usando ponteiro
ptr:\n");
for (i=0; i < 5; i++, ptr++)
{scanf("%d", ptr);
printf("i= %d, *ptr= %d
ptr= %x", i, *ptr, ptr);
printf("\n");
}
getch();
}
```

49



10.5 Ponteiros – Relação entre Ponteiros e Vetores

Executando o programa do Exemplo 8 temos:

```
C:\users\Cassilda\FEG_CURSOS\Curso_C\Curso_... - [x]
Usando v[i]:
5 6 7 8 9
v[0]= 5      &v[0]=22ff40
v[1]= 6      &v[1]=22ff44
v[2]= 7      &v[2]=22ff48
v[3]= 8      &v[3]=22ff4c
v[4]= 9      &v[4]=22ff50

Usando ponteiro (v+i):
5 6 7 8 9
i= 0, *(v+0) = 5      (v+0) = 22ff40
i= 1, *(v+1) = 6      (v+1) = 22ff44
i= 2, *(v+2) = 7      (v+2) = 22ff48
i= 3, *(v+3) = 8      (v+3) = 22ff4c
i= 4, *(v+4) = 9      (v+4) = 22ff50

Usando ponteiro ptr:
5 6 7 8 9
i= 0, *ptr= 5      ptr= 22ff40
i= 1, *ptr= 6      ptr= 22ff44
i= 2, *ptr= 7      ptr= 22ff48
i= 3, *ptr= 8      ptr= 22ff4c
i= 4, *ptr= 9      ptr= 22ff50
```

Observe que:

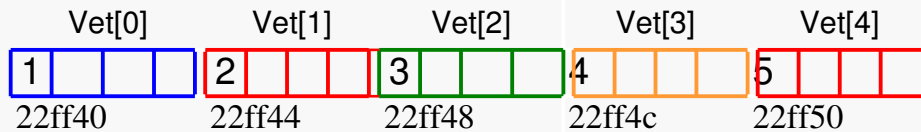
- Para imprimir os elementos de v foram usadas as notações: v[i], *(v+i) e * ptr .
- Para imprimir os endereços de v foram usadas as notações &v[i], (v+i) e ptr. Todas elas são equivalentes.
- Foi utilizado um incremento no ponteiro ++ptr para imprimir os elementos: v[0], v[2], v[2], v[3] e v[4].

50



10.5 Ponteiros – Relação entre Ponteiros e Vetores

Em termos de memória temos:



Observe que:

- O primeiro elemento do vetor tem valor 1 e foi atribuído ao endereço hexadecimal 40, o segundo tem valor 2 e foi atribuído ao 44 e assim sucessivamente. A diferença entre os endereços é de 4 porque o inteiro usa 4 bytes de memória, isto é 4 blocos de 1 byte cada.
- Os demais elementos dos vetores são alocados em endereços subsequentes.

51



10.5 Ponteiros – Relação entre Ponteiros e Vetores

Cuidados a Serem Tomados ao se Usar Ponteiros

O principal cuidado ao se usar um ponteiro deve ser:

- saiba sempre **para onde** o ponteiro está apontando. Isto é, nunca use um ponteiro que não foi inicializado.
- Um pequeno programa que demonstra como **não** usar um ponteiro:

```
// Errado - Não Execute
main ()
{
    int x,*p;
    x=13;
    *p=x;
    return(0);
}
```

Este programa compilará e rodará. **O que acontecerá? Ninguém sabe.**

O ponteiro **p** pode estar apontando para qualquer lugar, e você vai gravar o no. 13 em um lugar desconhecido. Com um número apenas, você provavelmente não vai ver nenhum defeito. Agora, se você começar a gravar números em posições aleatórias no seu computador, não vai demorar muito **Para Travar o micro** (se não acontecer coisa pior).

52



Ponteiros em C

Parte 2 – Funções com Parâmetros



Unesp – Campus de Guaratinguetá

Curso: Programação de Computadores

Prof. Aníbal Tavares

Profa. Cassilda Ribeiro

53



Até agora foi visto que:

- um programa em C é composto de uma função principal chamada `main` (programa principal) e várias funções (subprogramas). A função principal é quem coordena as demais funções.
- Os subprogramas (funções) devem ser criados para serem genéricos o bastante para se adaptarem a qualquer situação, visando justamente a possibilidade de reutilização do código.

Por esta razão foi criado o conceito de **passagem de parâmetros**, ou seja, passagem de informações para serem utilizadas dentro da função.

54



10.6 Ponteiros: *Funções com passagem de parâmetros*

Existem duas formas básicas de passagem de parâmetros na linguagem C:

- 1 - por valor;
- 2 - por referência ou endereço.

➤ Quando um dado é passado para uma função através de parâmetros e, apesar deste dado sofrer alterações dentro da função, ele não retorna ao programa principal com seu valor alterado, diz-se que este dado foi passado para a função por **VALOR**.

➤ Diz-se que um determinado dado é passado para uma função **por referência ou endereço** quando ao sofrer uma alteração dentro da função, ele retorna para o programa principal ou para a parte do programa onde a função foi chamada, com seu valor alterado.

55



10.6 Ponteiros: *Funções com passagem de parâmetros*

➤ A passagem de dados para uma função por **Referência ou por Endereço** é feita através de ponteiros.

➤ Para tanto, os ponteiros são passados para as funções através dos parâmetros (argumentos). Isso permite que os dados do programa principal (main), ou da parte responsável pela chamada das funções, sejam acessados e alterados pelas funções e sejam devolvidos alterados para o main ou para o local onde a função foi chamada.

Porque que a utilização de ponteiros nas funções permitem que os dados alterados dentro da função voltem para o programa principal?

56

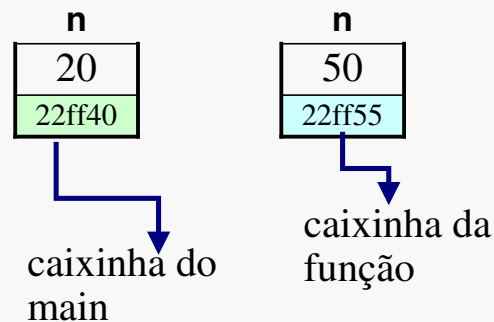


10.6 Ponteiros: *Funções com passagem de parâmetros*

➤ Quando se utiliza passagem de **dados por valor**, a função trabalha com uma área de memória e o programa principal com outra. Isto é, a função tem suas caixinhas para guardar suas variáveis e o main tem as deles. Quando a função acaba de ser executada ela se desfaz das suas caixinhas e o main não tem nenhum acesso a elas.

Suponha, por exemplo, o seguinte trecho de programa: $n=20$.

```
#include<stdio.h>
void modifica(int n)
{ n=n+30;
  printf(“%d”,n)
}
main()
{int n=20,
  modifica(n);
  printf(“%d”,n)
```



Observe que as duas caixinhas tem endereços diferentes

57



10.6 Ponteiros: *Funções com passagem de parâmetros*

➤ Quando se utiliza passagem de **dados por endereço**, a função trabalha com a mesma área de memória do programa principal. Isto é, a função usa as mesmas caixinhas que o main. Quando a função acaba de ser executada tudo ficou gravado e o main pode acessá-las sem nenhum problema.

➤ Isto ocorre porque não é passado para a função o valor da variável, mas sim o endereço. Deste modo a função trabalha diretamente nas caixinhas do main.

Para melhor compreender tudo isto, vamos então relembrar um pouco do que já vimos em funções.

58



10.6 Ponteiros: *Funções com passagem de parâmetros*

Toda função na linguagem C possui 4 partes:

(1) Tipo de Retorno: char

(2) Nome: misterio

```
char misterio ( int n, char c )  
{  
    char c2;  
    c2 = (char) (c+n);  
    return c2;  
}
```

(3) Lista Parâmetros:
(int n, char c)

(4) Corpo da Função: Comandos entre { }

59



10.6 Ponteiros: *Funções com passagem de parâmetros*

```
ch =misterio( 5, 'a');
```

Passo 1

**Chamada à
misterio**

```
char misterio( int n, char c )  
{  
    char c2;  
    c2 = (char) (c+n);  
    return c2;  
}
```

Passo 2

**Executa
misterio**

```
ch =(char)(5+'a');
```

Passo 3

**Retorna valor
de misterio**

Explicação

Ao esquema ao lado ilustra como é realizada a passagem de parâmetros:

Passo 1: Uma função, por exemplo o *main*, chama a **função misterio** e fornece os Valores iniciais para **n** e **c**.

Passo 2: É retornado o valor da variável **c2** obtido após a avaliação de uma expressão.

Passo 3: O fluxo de execução do programa é retornado ao ponto de chamada. A variável **ch** é substituída pelo valor retornado. 60



10.6 Ponteiros: *Funções com passagem de parâmetros*

- No exemplo anterior pode-se dizer que ocorreu uma passagem de parâmetros por valor, pois uma cópia dos valores das variáveis *n* e *c* é passada através da chamada à *função misterio*.
- Na *função misterio* esses valores são atribuídos as variáveis locais, isto é são armazenados em caixinhas de memórias que são conhecidas apenas dentro da *função misterio*.
- A seguir os valores de *n* e *c* são manipulados e o resultado dessa manipulação é armazenado em uma caixinha (variável local *c2*) que também só é conhecida dentro da função *misterio*.
- Para retornar o resultado dessa manipulação é empregado a palavra reservada *return*, bem como é especificado um tipo de retorno no parâmetro de saída coerente com o tipo da função (*char*).

O Exemplo 8 a seguir, ilustra como é feita a passagem de parâmetros por valor em termos de memória.

61



10.6 Ponteiros: *Funções com passagem de parâmetros*

1- Passagem por valor

Exemplo 8: Programa Maior

```
#include <stdlib.h>
#include <stdio.h>
int Maior(int x, int y)
{
    return (x > y) ? x:y;
}
main()
{
    int a, b, max;
    printf("Entre com a e b:");
    scanf("%d %d", &a, &b);
    max = Maior(a,b);
    printf("Max{a,b} = %d \n",max);
}
```

Memória da Função Main

a	b
4	5
5000	5050

Memória da Função Maior

x	y
4	5
6000	7050

return 5;

62



10.6 Ponteiros: *Funções com passagem de parâmetros*

1- Passagem por valor

Exemplo 8: Programa Maior

```
#include <stdlib.h>
#include <stdio.h>
int Maior(int x, int y)
{
    return (x > y) ? x:y;
}
main()
{
    int a, b, max;
    printf("Entre com a e b:");
    scanf("%d %d", &a, &b);
    max = Maior(a,b);
    printf("Max{a,b} = %d \n",max);
}
```

No programa maior a função main chama a função Maior e passa os valores contidos nas variáveis a e b (cujo escopo é a função main) para as variáveis x e y (cujo escopo é a função Maior). Diz-se que este tipo de passagem de parâmetros é **passagem por valor**, pois é passada uma cópia dos valores das variáveis contidas em um função para as variáveis de outra função. Observe que para isto, a função Maior tem especificado um parâmetro de saída (tipo int) e emprega o comando return.

63



10.6 Ponteiros: *Funções com passagem de parâmetros*

2- Passagem por referência

Na passagem de dados **por referência** não é passado o valor de uma variável e sim o seu endereço.

A partir deste endereço é possível manipular uma variável x declarada na função f1, a partir de uma outra função f2. Basta, para tanto, que uma variável local de f2, por exemplo y, seja declarada do tipo ponteiro e receba o endereço da variável x declarada em f1.

Assim qualquer modificação realizada em y dentro de f2 irá acarretar mudanças em x, pois f1 e f2 estão usando a mesma caixinha de memória.

Um exemplo disso é ilustrado com o Exemplo 9.

64



10.6 Ponteiros: *Funções com passagem de parâmetros*

Exemplo 9: Programa Troca

```
#include <stdlib.h>
#include <stdio.h>
int Troca(int *x, int *y)
{
    int tmp = *x;
    *x = *y;
    *y = tmp;
}
main()
{int a, b, max;
 printf("Entre com a e b:");
 scanf("%d %d", &a, &b);
 printf("a=%d - b=%d\n",a,b);
 Troca(&a, &b);
 printf("a=%d - b=%d\n",a,b);}
```

Memória da Função main

a	b
4	5
5000	5050

Função Troca

Memória da Função Troca

a	b
5	4
5000	5050

65



10.6 Ponteiros: *Funções com passagem de parâmetros*

➤ Para simplificar, podemos imaginar que a **passagem de parâmetros por referência** é como uma rua de mão dupla para a função, isto é: Tudo que vai para dentro da função, volta para o programa principal.

➤ Nesta analogia, a **passagem de parâmetros por valor** seria uma rua de mão única, pois tudo que vai do programa principal para a função **não volta**.

66



10.6 Ponteiros: *Funções com passagem de parâmetros*

Exemplo 10—Referência ou Valor

```
#include <stdio.h>
void printV(int X[], int n)
{ int i;
  for (i = 0; i < 10; i++)
    printf(" [%4d] ",X[i]); puts("");}
void preencheV(int X[], int n)
{ int i;
  for (i = 0; i < n; i++)
    X[i] = 2*i + 1;}
main ( )
{ int i, n=10, V[10] = {0};
  printV(V, n);
  preencheV(V, n);
  printV(V, n);
}
```

Observação

É importante observar que a discussão anterior entre passagem por valor versus passagem por referência aparentemente não é válida para vetores tal como descrito no Exemplo 10.

Observe que a função preencheV modifica os valores contidos em V sem empregar o comando return e sem possuir um parâmetro de retorno (o tipo de retorno de preencheV é void)! A explicação para isto é que a chamada da função preencheV passa o endereço de V[0].

67



10.6 Ponteiros: *Funções com passagem de parâmetros*

Exemplo 10—Referência ou Valor

```
#include <stdio.h>
void printV(int X[], int n)
{ int i;
  for (i = 0; i < 10; i++)
    printf(" [%4d] ",X[i]); puts("");}
void preencheV(int X[], int n)
{ int i;
  for (i = 0; i < n; i++)
    X[i] = 2*i + 1;}
main ( )
{ int i, n=10, V[10] = {0};
  printV(V, n);
  preencheV(V, n);
  printV(V, n);
}
```

Escopo da Função main

V[0]	V[1]	...	V[9]
0	0	...	0
5000	5004	...	5036

Função preencheV

Escopo da Função preencheV

X[0]	X[1]	...	X[9]
1	3	...	17
5000	5004	...	5036

68



10.6 Ponteiros: *Funções com passagem de parâmetros*

No exemplo anterior, a função main contém uma chamada a função preencheV através do comando: preenche(V, n).

É importante observar que esta chamada é tal que fornece o endereço inicial do primeiro elemento de V (ou seja, &V[0], pois lembre-se que $V \leftrightarrow \&V[0]$) e o valor contido em n.

Assim, o endereço do primeiro elemento de V é associado ao nome do vetor X e qualquer modificação realizada nos elementos de X na verdade estará alterando o conteúdo dos elementos do vetor V.

Então, a notação V[i] é um atalho para um conjunto de endereços de memória e T também é vinculado a este mesmo conjunto de endereços ao se realizar a chamada a função preencheV.

Deste modo, **quando são empregados vetores na chamada de funções, ocorre, implicitamente a passagem por referência.**

69



10.6 Ponteiros: *Funções com passagem de parâmetros*

Observação

Dado que a passagem de vetores é realizada por referência (ou seja, endereço de memória), é possível empregar a notação de ponteiros na definição das funções, bem como no acesso aos elementos de V através do nome X. Para tanto, pode-se usar a notação que **emprega a aritmética de ponteiros** mais o operador de **dereferência** para acessar o elemento V[i], ou seja, empregar $*(X+i)$ tal como ilustrado nos Exemplos 11 e 12.

70



10.6 Ponteiros: *Funções com passagem de parâmetros*

Exemplo 11 – Ponteiros

```
#include <stdio.h>
void printV(int *X, int n)
{ int i;
  for (i = 0; i < 10; i++)
    printf(" [%4d] ",*(X+i)); puts("");}
void preencheV(int *X, int n)
{ int i;
  for (i = 0; i < 10; i++)
    *(X+i) = 2*i + 1;}
main ( )
{ int i, n=10, V[10] = {0};
  printV(V, n);
  preencheV(V, n);
  printV(V, n);
}
```

71



10.6 Ponteiros: *Funções com passagem de parâmetros*

Exemplo 12–Cadastro+Funções

```
#include <conio.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
//Definindo o novo tipo PESSOA.
typedef struct PESSOA
{ char nome[100];
  float salario;
  char sexo;}PE;
// Protótipos das funções.
void Leitura(PE *V, int n);
void Imprime(PE *V, int n);
main ( ) // Função Principal
{ PE V[10]; int i, n;
  printf("Tamanho de V:");
  scanf("%d", &n);
```

```
Leitura(V, n);
Imprime(V, n);
system("pause");
} // Fim da função main
void Leitura(PE *V, int n)
{ int i;
  for (i=0; i < n; i++)
  { printf("Cadastro %d \n", i+1);
    fflush(stdin);
    printf("Entre com o nome: ");
    gets((V+i)->nome);
    printf("Entre com salario: ");
    scanf("%f",&((V+i)->salario));
    printf("Entre com sexo: ");
    (V+i)->sexo = getche();
    fflush(stdin);
    system("CLS"); } } // Fim Leitura
```

72



10.6 Ponteiros: *Funções com passagem de parâmetros*

Exemplo 12 – continuação

```
void Imprime(PE *V, int n)
{ int i;
for (i=0; i < n; i++)
{ printf("----- \n");
printf("Cadastro %4d: \n", i+1);
printf("Nome: %s \n", (*(V+i)).nome);
printf("Salario: %f \n", V[i].salario);
printf("Sexo: %c \n", (V+i)->sexo);
printf("-----\n\n");
}
printf("\n");
}
```

Observações

No Exemplo 12 é empregado o tipo **PESSOA** de modo a se poder criar um vetor **V** cujos elementos são do tipo **PESSOA**. Assim, cada elemento **V[i]** possui 3 campos: nome, salario e sexo. Para acessar a informação destes campos existem 3 formas equivalentes:

- (i) **V[i].salario**
- (ii) ***(V+i).salario**
- (iii) **(V+i)->salario**

73



10.6 Ponteiros: *Funções com passagem de parâmetros*

Exemplo 12: Cadastro+Funções

Versão 2- Sem usar ponteiros

```
#include <conio.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
// Definindo o novo tipo PESSOA.
typedef struct PESSOA
{ char nome[100];
float salario;
char sexo;}PE;
```

```
// Protótipos das funções.
void Leitura(PE V[], int n);
void Imprime(PE V[], int n);
// Função Principal
main ( )
{ PE V[10]; int i, n;
printf("Tamanho de V:");
scanf("%d", &n);
Leitura(V, n);
Imprime(V, n);
system("pause");
} // Fim da função main
```

74



10.6 Ponteiros: *Funções com passagem de parâmetros*

Exemplo 12 Versão 2 – continuação

```
void Leitura(PE V[], int n)
{ int i;
  for (i=0; i < n; i++)
  { printf("Cadastro %d \n", i+1);
    fflush(stdin);
    printf("Entre com o nome: ");
    gets(V[i].nome);
    printf("Entre com salario: ");
    scanf("%f",&((V+i)->salario));
    printf("Entre com sexo: ");
    (V+i)->sexo = getche();
    fflush(stdin);
    system("CLS"); } } // Fim Leit
```

Exemplo 12 Versão 2 – continuação

```
void Imprime(PE V[], int n)
{ int i;
  for (i=0; i < n; i++)
  { printf("-----
    \n");
    printf("Cadastro %4d: \n", i+1);
    printf("Nome: %s \n",
      (*(V+i)).nome);
    printf("Salario: %f \n", V[i].salario);
    printf("Sexo: %c \n", (V+i)->sexo);
    printf("-----\n\n");
  }
  printf
```

75



10.6 Ponteiros: *Funções com passagem de parâmetros*

FIM

Ponteiros em C

76