



Mais sobre memória Ponteiros



Prof. Bruno Travençolo

Observações sobre a memória

Endereço	Blocos	Tamanho
1		(1 byte)
2		(1 byte)
3		(1 byte)
4		(1 byte)
5		(1 byte)
6		(1 byte)
7		(1 byte)
8		(1 byte)
9		(1 byte)
10		(1 byte)
11		(1 byte)
12		(1 byte)
13		(1 byte)
14		(1 byte)
....		



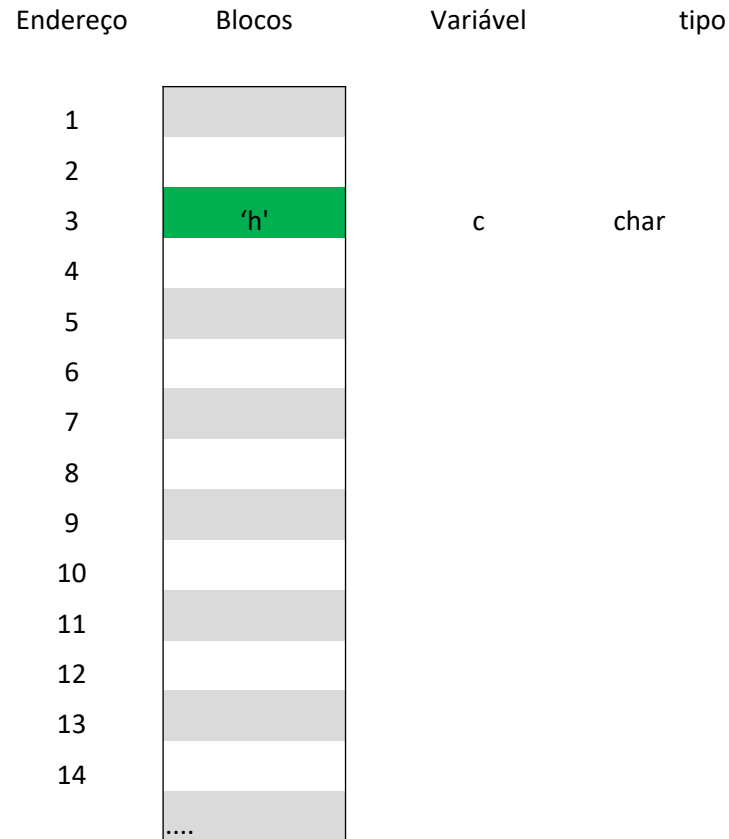
Observações sobre a memória

	Endereço	Blocos	Variável	tipo
char <i>c</i> ;	1		c	char
	2			
	3			
	4			
	5			
	6			
	7			
	8			
	9			
	10			
	11			
	12			
	13			
	14			
			



Observações sobre a memória

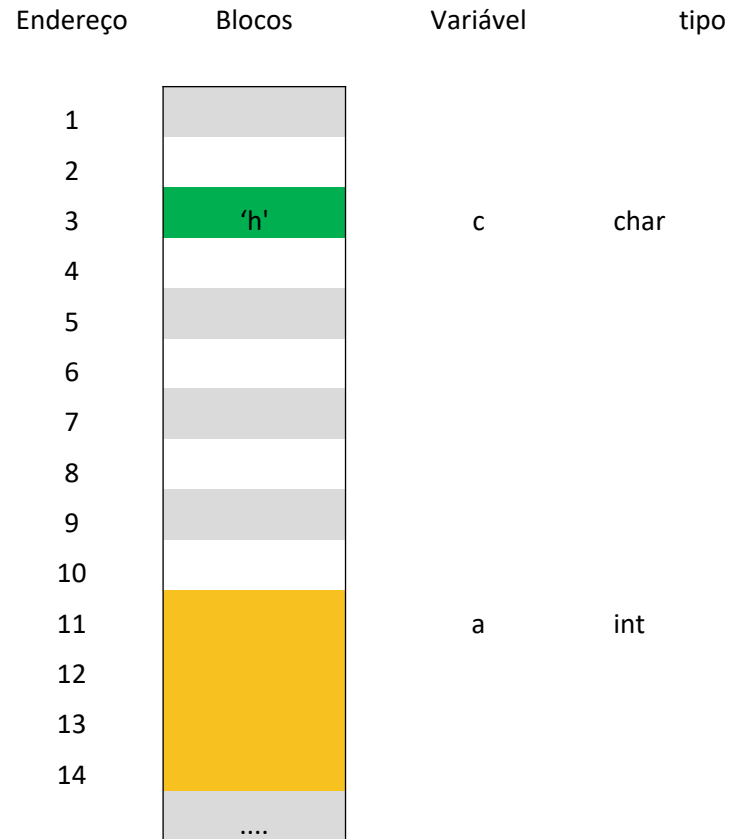
```
char c;  
c = 'h';
```



Observações sobre a memória

```
char c;  
c = 'h';
```

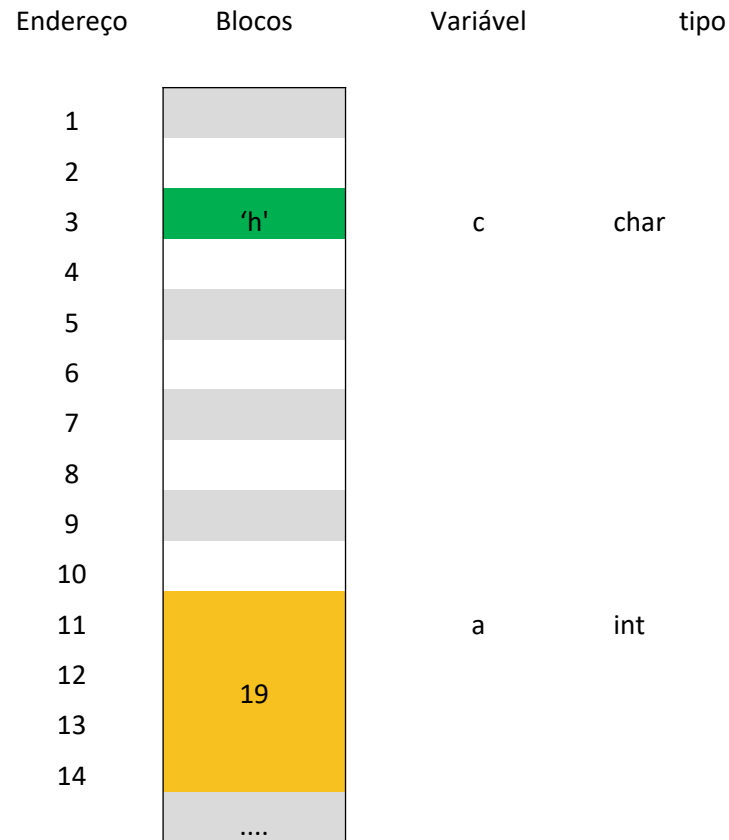
```
int a;
```



Observações sobre a memória

```
char c;  
c = 'h';
```

```
int a;  
a = 19;
```

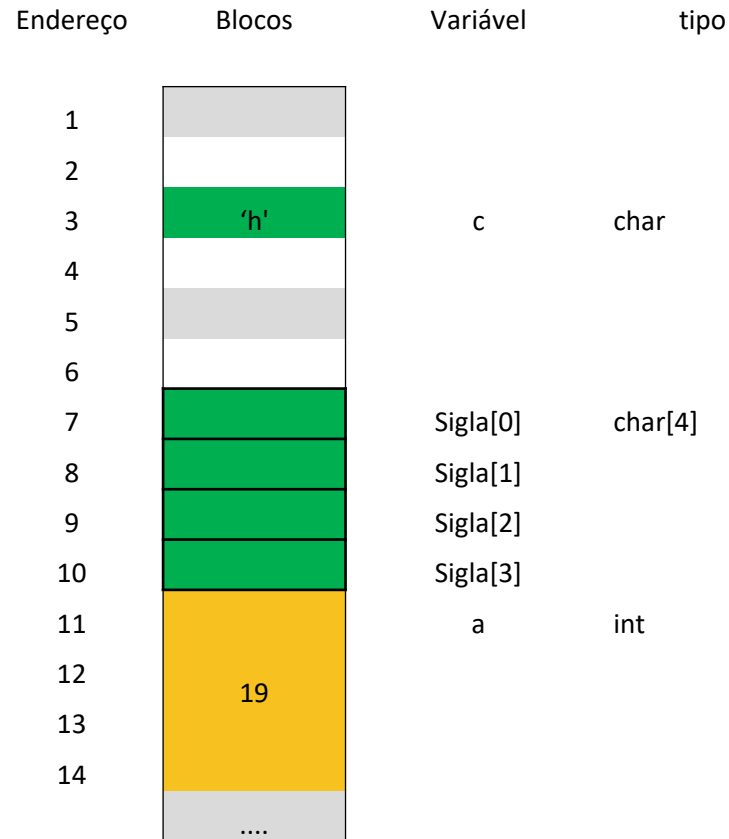


Observações sobre a memória

```
char c;  
c = 'h';
```

```
int a;  
a = 19;
```

```
char Sigla[4];
```



Observações sobre a memória

```
char c;  
c = 'h';
```

```
int a;  
a = 19;
```

```
char Sigla[4];  
Sigla[0] = 'U';  
Sigla[1] = 'F';  
Sigla[2] = 'U';  
Sigla[3] = '\0';
```

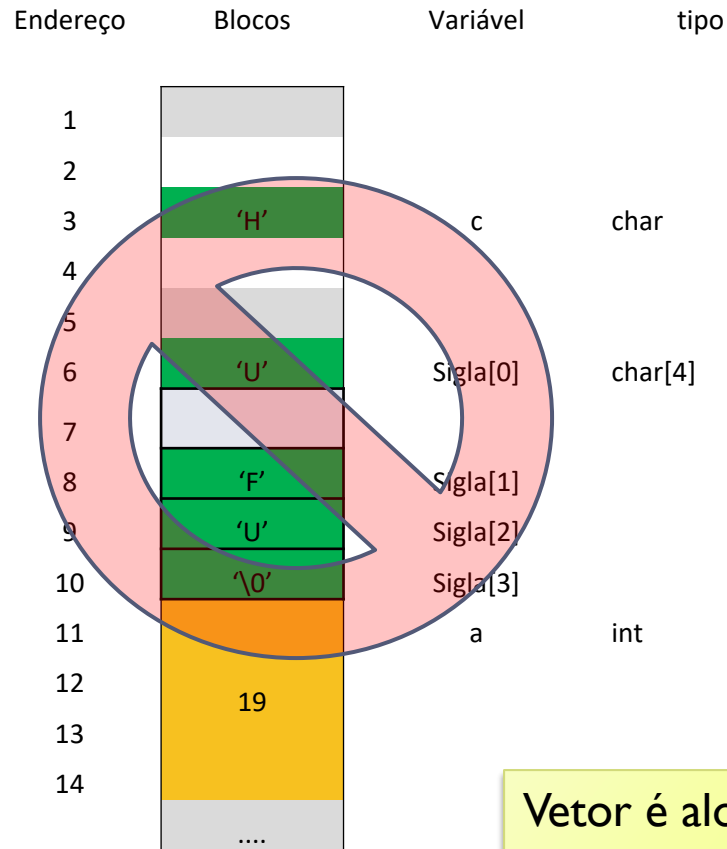
Endereço	Blocos	Variável	tipo
1			
2			
3	'H'	c	char
4			
5			
6			
7	'U'	Sigla[0]	char[4]
8	'F'	Sigla[1]	
9	'U'	Sigla[2]	
10	'\0'	Sigla[3]	
11	19	a	int
12			
13			
14			
		

Observações sobre a memória

```
char c;  
c = 'h';
```

```
int a;  
a = 19;
```

```
char Sigla[4];  
Sigla[0] = 'U';  
Sigla[1] = 'F';  
Sigla[2] = 'U';  
Sigla[3] = '\0';
```



Vetor é alocado em blocos contínuos de memória

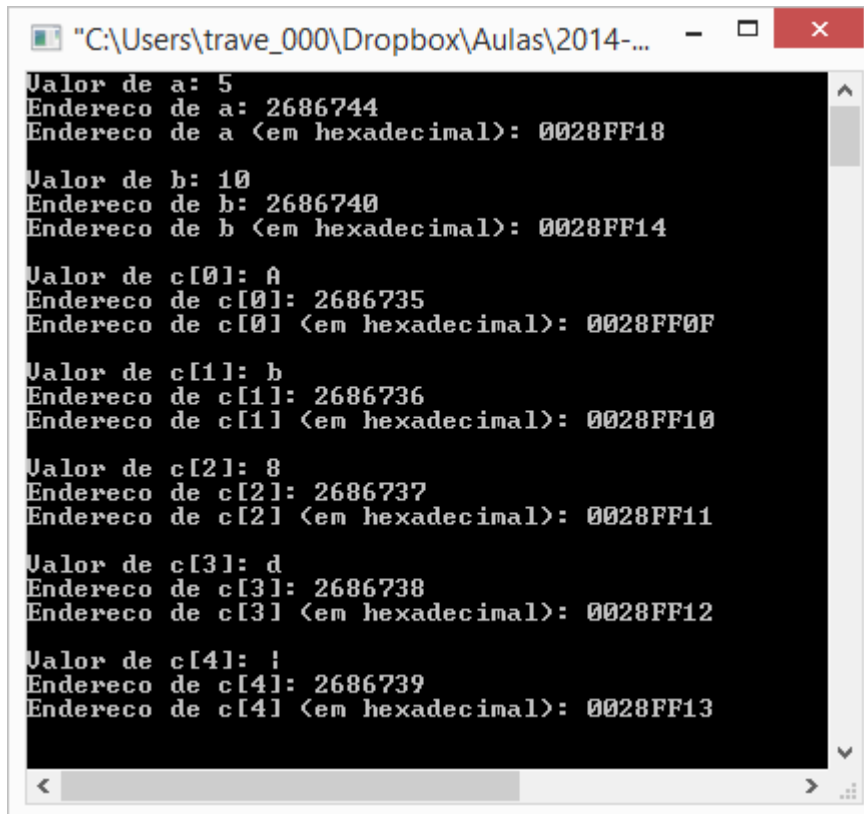
Endereço de variáveis

- Para descobrir o endereço de uma variável em C, use o operador &

```
6     int i;  
7     int a = 5;  
8     int b = 10;  
9     char c[5] = {'A', 'b', '8', 'd', '|'};  
12    printf("Valor de a: %d \n", a);  
13    printf("Endereco de a: %d \n", &a);  
14    printf("Endereco de a (em hexadecimal): %p \n\n", &a);  
15  
16    printf("Valor de b: %d \n", b);  
17    printf("Endereco de b: %d \n", &b);  
18    printf("Endereco de b (em hexadecimal): %p \n\n", &b);  
19  
20    for (i=0; i < 5; i++){  
21        printf("Valor de c[%d]: %c \n", i, c[i]);  
22        printf("Endereco de c[%d]: %d \n", i, &c[i]);  
23        printf("Endereco de c[%d] (em hexadecimal): %p \n\n", i, &c[i]);  
24    }
```

Endereço de variáveis

- ▶ Para descobrir o endereço de uma variável em C, use o operador &



```
"C:\Users\trave_000\Dropbox\Aulas\2014-... - [X]
Valor de a: 5
Endereco de a: 2686744
Endereco de a (em hexadecimal): 0028FF18

Valor de b: 10
Endereco de b: 2686740
Endereco de b (em hexadecimal): 0028FF14

Valor de c[0]: a
Endereco de c[0]: 2686735
Endereco de c[0] (em hexadecimal): 0028FF0F

Valor de c[1]: b
Endereco de c[1]: 2686736
Endereco de c[1] (em hexadecimal): 0028FF10

Valor de c[2]: 8
Endereco de c[2]: 2686737
Endereco de c[2] (em hexadecimal): 0028FF11

Valor de c[3]: d
Endereco de c[3]: 2686738
Endereco de c[3] (em hexadecimal): 0028FF12

Valor de c[4]: !
Endereco de c[4]: 2686739
Endereco de c[4] (em hexadecimal): 0028FF13
```

Exercício: faça o mapa de memória para este programa, usando os endereços reais apresentados ao lado

-
- ▶ Relembrando as primeiras aulas – comando `scanf()`



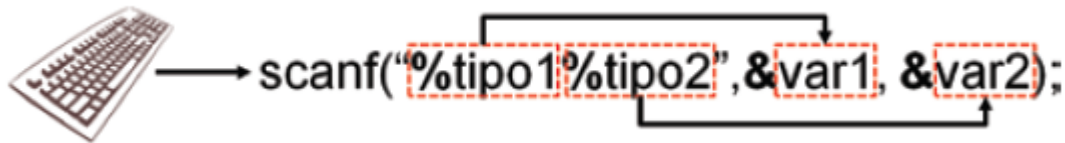
Comando de entrada

- ▶ Em C, o comando que permite ler dados da entrada padrão (no caso o teclado) é o `scanf()`
- ▶ **`scanf()`**
- ▶ Sintaxe: **`scanf("format",&name1,...)`**
 - ▶ `format` – especificador de formato da entrada que será lida
 - ▶ `&name1, &name2, ...` – endereços das variáveis que irão receber os valores lidos



Comando de entrada

- ▶ Temos, igual ao comando printf, que especificar o tipo (formato) do dado que será lido
- ▶ scanf("tipo de entrada", lista de variáveis)



- ▶ Alguns “tipos de entrada”
 - ▶ %c – leitura de **um** caractere
 - ▶ %d – leitura de números inteiros
 - ▶ %f – leitura de número reais
 - ▶ %s – leitura de **vários** caracteres – **ainda não usamos no curso**

Comando scanf() - Exemplo

```
// declaração das variáveis
```

```
float peso;
```

```
float altura;
```

```
float IMC;
```

```
// Obtendo os dados do usuário
```

```
printf("Informe o peso: ");
```

```
scanf("%f",&peso);
```

```
printf("Informe a altura: ");
```

```
scanf("%f",&altura);
```

```
// calculando o ICM e mostrando o resultado
```

```
IMC = peso / (altura*altura);
```

```
printf("Peso: %f, Altura: %f, IMC: %f", peso, altura, IMC);
```



Comando scanf() - Exemplo

▶ Como “ler em voz alta”

`scanf("%f",&peso);` // leia um valor real (float ou double) e armazene no endereço reservado para a variável peso

▶ O símbolo & indica qual é o endereço da variável que vai receber os dados lidos

- ▶ peso – variável peso
- ▶ &peso – endereço da variável peso



Endereços de variáveis

- ▶ Se no comando `scanf()` precisamos passar um endereço de variável, será que é possível passar diretamente o endereço, sem usar o símbolo `&`?
- ▶ Veja o código a seguir que faz isso



```
int k;
unsigned int endereco_de_k;

// inicializando k
k = 10;
printf("\n Valor da variavel 'k': %d \n",k);

// obtendo o endereço da variável 'k'
// vamos usar o operador &
endereco_de_k = &k;

printf("\n Endereco da variavel 'k': %u \n",endereco_de_k);
printf("\n Endereco da variavel 'k': %u \n",&k);
printf("\n Endereco da variavel 'k': %p (em hexadecimal)\n",&k);

// sabemos que o scanf pede um endereço de memória
// o que acontece se passarmos o endereço da
// variável k?
printf("\n Digite o valor novo valor, a ser armazenado no endereço da variavel k: ");
// OBSERVE que não estamos usando & no scanf! Isso porque já temos o endereço
scanf("%d",endereco_de_k);

// mostrando o novo valor de 'k'
printf("\n\n Valor da variavel k, apos scanf de 'endereco_de_k': %d \n",k);

// mostrando o endereço de 'k' que deve permanecer o mesmo de antes
printf("\n Endereco da variavel 'k': %u \n",endereco_de_k);
printf("\n Endereco da variavel 'k': %u \n",&k);
printf("\n Endereco da variavel 'k': %p (em hexadecimal)\n",&k);
```



```
int k;  
unsigned int endereco_de_k;
```

```
// inicializando k  
k = 10;  
printf("\n Valor da variavel 'k': %d \n",k);
```

```
// obtendo o endereço da variável 'k'  
// vamos usar o operador &  
endereco_de_k = &k;
```

```
printf("\n Endereco da variavel 'k': %u \n",endereco_de_k);  
printf("\n Endereco da variavel 'k': %u \n",&k);  
printf("\n Endereco da variavel 'k': %p (em hexadecimal)\n",&k);
```

```
// sabemos que o scanf pede um endereço de memória  
// o que acontece se passarmos o endereço da  
// variável k?  
printf("\n Digite o valor novo valor, a ser armazenado no endereço da variavel k: ");  
// OBSERVE que não estamos usando & no scanf! Isso porque já temos o endereço  
scanf("%d",endereco_de_k);
```

```
// mostrando o novo valor de 'k'  
printf("\n\n Valor da variavel k, apos scanf de 'endereco_de_k': %d \n",k);
```

```
// mostrando o endereço de 'k' que deve permanecer o mesmo de antes  
printf("\n Endereco da variavel 'k': %u \n",endereco_de_k);  
printf("\n Endereco da variavel 'k': %u \n",&k);  
printf("\n Endereco da variavel 'k': %p (em hexadecimal)\n",&k);
```

Endereço	Blocos (1 byte)	Nome variável	Tipo
0/NULL	indefinido	----	----
1	lx	k	int
2			
3			
4			
5	lx		
6		endereco_de_k	unsigned int
7			
8			
9			
10			
11			

```
int k;
unsigned int endereco_de_k;

// inicializando k
k = 10;
printf("\n Valor da variavel 'k': %d \n",k);
```

```
// obtendo o endereço da variável 'k'
// vamos usar o operador &
endereco_de_k = &k;
```

```
printf("\n Endereco da variavel 'k': %u \n",endereco_de_k);
printf("\n Endereco da variavel 'k': %u \n",&k);
printf("\n Endereco da variavel 'k': %p (em hexadecimal)\n",&k);
```

```
// sabemos que o scanf pede um endereço de memória
// o que acontece se passarmos o endereço da
// variável k?
printf("\n Digite o valor novo valor, a ser armazenado no endereço da variavel k: ");
// OBSERVE que não estamos usando & no scanf! Isso porque já temos o endereço
scanf("%d",endereco_de_k);
```

```
// mostrando o novo valor de 'k'
printf("\n\n Valor da variavel k, apos scanf de 'endereco_de_k': %d \n",k);
```

```
// mostrando o endereço de 'k' que deve permanecer o mesmo de antes
printf("\n Endereco da variavel 'k': %u \n",endereco_de_k);
printf("\n Endereco da variavel 'k': %u \n",&k);
printf("\n Endereco da variavel 'k': %p (em hexadecimal)\n",&k);
```

1		k	int
2	10		
3			
4			
5			
6		endereco_de_k	unsigned int
7	lx		
8			
9			

```

int k;
unsigned int endereco_de_k;

// inicializando k
k = 10;
printf("\n Valor da variavel 'k': %d \n",k);

// obtendo o endereço da variável 'k'
// vamos usar o operador &
endereco_de_k = &k;

```

1		k	int
2	10		
3			
4			
5			
6		endereco_de_k	unsigned int
7	1		
8			
9			

```

printf("\n Endereco da variavel 'k': %u \n",endereco_de_k);
printf("\n Endereco da variavel 'k': %u \n",&k);
printf("\n Endereco da variavel 'k': %p (em hexadecimal)\n",&k);

// sabemos que o scanf pede um endereço de memória
// o que acontece se passarmos o endereço da
// variável k?
printf("\n Digite o valor novo valor, a ser armazenado no endereço da variavel k: ");
// OBSERVE que não estamos usando & no scanf! Isso porque já temos o endereço
scanf("%d",endereco_de_k);

// mostrando o novo valor de 'k'
printf("\n\n Valor da variavel k, apos scanf de 'endereco_de_k': %d \n",k);

// mostrando o endereço de 'k' que deve permanecer o mesmo de antes
printf("\n Endereco da variavel 'k': %u \n",endereco_de_k);
printf("\n Endereco da variavel 'k': %u \n",&k);
printf("\n Endereco da variavel 'k': %p (em hexadecimal)\n",&k);

```

```

int k;
unsigned int endereco_de_k;

// inicializando k
k = 10;
printf("\n Valor da variavel 'k': %d \n",k);

// obtendo o endereço da variável 'k'
// vamos usar o operador &
endereco_de_k = &k;

```

1		k	int
2	10		
3			
4			
5			
6		endereco_de_k	unsigned int
7	1		
8			
9			

```

printf("\n Endereco da variavel 'k': %u \n",endereco_de_k);

```

Valor 1

```

printf("\n Endereco da variavel 'k': %u \n",&k);
printf("\n Endereco da variavel 'k': %p (em hexadecimal)\n",&k);

// sabemos que o scanf pede um endereço de memória
// o que acontece se passarmos o endereço da
// variável k?
printf("\n Digite o valor novo valor, a ser armazenado no endereço da variavel k: ");
// OBSERVE que não estamos usando & no scanf! Isso porque já temos o endereço
scanf("%d",endereco_de_k);

// mostrando o novo valor de 'k'
printf("\n\n Valor da variavel k, apos scanf de 'endereco_de_k': %d \n",k);

// mostrando o endereço de 'k' que deve permanecer o mesmo de antes
printf("\n Endereco da variavel 'k': %u \n",endereco_de_k);
printf("\n Endereco da variavel 'k': %u \n",&k);
printf("\n Endereco da variavel 'k': %p (em hexadecimal)\n",&k);

```

```
int k;
unsigned int endereco_de_k;

// inicializando k
k = 10;
printf("\n Valor da variavel 'k': %d \n",k);

// obtendo o endereço da variável 'k'
// vamos usar o operador &
endereco_de_k = &k;
```

1		k	int
2	10		
3			
4			
5			
6		endereco_de_k	unsigned int
7	1		
8			
9			

```
printf("\n Endereco da variavel 'k': %u \n",endereco_de_k);
printf("\n Endereco da variavel 'k': %u \n",&k);
```

Endereço l

```
printf("\n Endereco da variavel 'k': %p (em hexadecimal)\n",&k);

// sabemos que o scanf pede um endereço de memória
// o que acontece se passarmos o endereço da
// variável k?
printf("\n Digite o valor novo valor, a ser armazenado no endereço da variavel k: ");
// OBSERVE que não estamos usando & no scanf! Isso porque já temos o endereço
scanf("%d",endereco_de_k);

// mostrando o novo valor de 'k'
printf("\n\n Valor da variavel k, apos scanf de 'endereco_de_k': %d \n",k);

// mostrando o endereço de 'k' que deve permanecer o mesmo de antes
printf("\n Endereco da variavel 'k': %u \n",endereco_de_k);
printf("\n Endereco da variavel 'k': %u \n",&k);
printf("\n Endereco da variavel 'k': %p (em hexadecimal)\n",&k);
```

```

int k;
unsigned int endereco_de_k;

// inicializando k
k = 10;
printf("\n Valor da variavel 'k': %d \n",k);

// obtendo o endereço da variável 'k'
// vamos usar o operador &
endereco_de_k = &k;

```

```

printf("\n Endereco da variavel 'k': %u \n",endereco_de_k);
printf("\n Endereco da variavel 'k': %u \n",&k);
printf("\n Endereco da variavel 'k': %p (em hexadecimal)\n",&k);

```

1		k	int
2	10		
3			
4			
5			
6		endereco_de_k	unsigned int
7	1		
8			
9			

Endereço 0x1

```

// sabemos que o scanf pede um endereço de memória
// o que acontece se passarmos o endereço da
// variável k?
printf("\n Digite o valor novo valor, a ser armazenado no endereço da variavel k: ");
// OBSERVE que não estamos usando & no scanf! Isso porque já temos o endereço
scanf("%d",endereco_de_k);

// mostrando o novo valor de 'k'
printf("\n\n Valor da variavel k, apos scanf de 'endereco_de_k': %d \n",k);

// mostrando o endereço de 'k' que deve permanecer o mesmo de antes
printf("\n Endereco da variavel 'k': %u \n",endereco_de_k);
printf("\n Endereco da variavel 'k': %u \n",&k);
printf("\n Endereco da variavel 'k': %p (em hexadecimal)\n",&k);

```



```
int k;
unsigned int endereco_de_k;

// inicializando k
k = 10;
printf("\n Valor da variavel 'k': %d \n",k);
```

```
// obtendo o endereço da variável 'k'
// vamos usar o operador &
endereco_de_k = &k;
```

```
printf("\n Endereco da variavel 'k': %u \n",endereco_de_k);
printf("\n Endereco da variavel 'k': %u \n",&k);
printf("\n Endereco da variavel 'k': %p (em hexadecimal)\n",&k);
```

1		k	int
2	10		
3			
4			
5			
6		endereco_de_k	unsigned int
7	1		
8			
9			

```
// sabemos que o scanf pede um endereço de memória
// o que acontece se passarmos o endereço da
// variável k?
printf("\n Digite o valor novo valor, a ser armazenado no endereço da variavel k: ");
// OBSERVE que não estamos usando & no scanf! Isso porque já temos o endereço
scanf("%d",endereco_de_k);
```

```
// mostrando o novo valor de 'k'
printf("\n\n Valor da variavel k, apos scanf de 'endereco_de_k': %d \n",k);
```

```
// mostrando o endereço de 'k' que deve permanecer o mesmo de antes
printf("\n Endereco da variavel 'k': %u \n",endereco_de_k);
printf("\n Endereco da variavel 'k': %u \n",&k);
printf("\n Endereco da variavel 'k': %p (em hexadecimal)\n",&k);
```



```
int k;
unsigned int endereco_de_k;

// inicializando k
k = 10;
printf("\n Valor da variavel 'k': %d \n",k);

// obtendo o endereço da variável 'k'
// vamos usar o operador &
endereco_de_k = &k;
```

1		k	int
2	10		
3			
4			
5			
6		endereco_de_k	unsigned int
7	1		
8			
9			

```
printf("\n Endereco da variavel 'k': %u \n",endereco_de_k);
printf("\n Endereco da variavel 'k': %u \n",&k);
printf("\n Endereco da variavel 'k': %p (em hexadecimal)\n",&k);
```

```
// sabemos que o scanf vai ler o valor da variavel k:
// o que acontece se o usuário digitar um valor diferente do esperado?
// variável k?
printf("\n Digite o valor da variavel k: ");
// OBSERVE que não é necessário especificar o tipo da variavel k:
scanf("%d",endereco_de_k);
```

```
scanf("%d",endereco_de_k);
```

Leia um valor inteiro e armazene no endereço "endereco_de_k"

>> suponha que o usuário digite o número 50

```
// mostrando o novo valor da variavel k:
printf("\n\n Valor da variavel k: %d \n",k);

// mostrando o endereço de 'k' que deve permanecer o mesmo de antes
printf("\n Endereco da variavel 'k': %u \n",endereco_de_k);
printf("\n Endereco da variavel 'k': %u \n",&k);
printf("\n Endereco da variavel 'k': %p (em hexadecimal)\n",&k);
```

variavel k: ");
endereço

```
int k;
unsigned int endereco_de_k;

// inicializando k
k = 10;
printf("\n Valor da variavel 'k': %d \n",k);

// obtendo o endereço da variável 'k'
// vamos usar o operador &
endereco_de_k = &k;
```

1		k	int
2	10		
3			
4			
5			
6		endereco_de_k	unsigned int
7	1		
8			
9			

```
printf("\n Endereco da variavel k: ");
printf("\n Endereco da variavel k: ");
printf("\n Endereco da variavel k: ");
```

```
scanf("%d",endereco_de_k);
```

Observe que não foi utilizado o operador "&" para pegar o endereço da variável "endereco_de_k"

```
// sabemos que o scanf vai ler o valor da variavel k
// o que acontece se o usuário digitar um valor diferente de 10?
// variável k?
printf("\n Digite o valor da variavel k: ");
// OBSERVE que não é necessário o operador & para o scanf
scanf("%d",endereco_de_k);
```

```
printf("\n Valor da variavel k: ");
scanf("%d",endereco_de_k);
```

```
// mostrando o novo valor da variavel k
printf("\n\n Valor da variavel k: %d \n",k);
```

```
// mostrando o endereço de 'k' que deve permanecer o mesmo de antes
printf("\n Endereco da variavel 'k': %u \n",endereco_de_k);
printf("\n Endereco da variavel 'k': %u \n",&k);
printf("\n Endereco da variavel 'k': %p (em hexadecimal)\n",&k);
```

```
int k;
unsigned int endereco_de_k;

// inicializando k
k = 10;
printf("\n Valor da variavel 'k': %d \n",k);

// obtendo o endereço da variável 'k'
// vamos usar o operador &
endereco_de_k = &k;
```

1		k	int
2	10		
3			
4			
5			
6		endereco_de_k	unsigned int
7	1		
8			
9			

```
printf("\n Endereco da variavel k: ");
printf("\n Endereco da variavel k: ");
printf("\n Endereco da variavel k: ");
```

```
scanf("%d",&endereco_de_k);
```

Assim, o valor a ser digitado será armazenado no endereço de memória que está escrito na variável "endereco_de_k"

```
// sabemos que o scanf vai ler o valor que está escrito na variável k?
// o que acontece se o usuário digitar um valor diferente do endereço de memória?
// OBSERVE que não é o endereço de memória que é digitado, é o valor da variável k!
printf("\n Digite o valor da variavel k: ");
scanf("%d",&endereco_de_k);
```

```
printf("\n Digite o valor da variavel k: ");
scanf("%d",&endereco_de_k);
```

```
// mostrando o novo valor da variavel k
printf("\n\n Valor da variavel k: %d \n",k);

// mostrando o endereço de 'k' que deve permanecer o mesmo de antes
printf("\n Endereco da variavel 'k': %u \n",endereco_de_k);
printf("\n Endereco da variavel 'k': %u \n",&k);
printf("\n Endereco da variavel 'k': %p (em hexadecimal)\n",&k);
```

```
int k;
unsigned int endereco_de_k;

// inicializando k
k = 10;
printf("\n Valor da variavel 'k': %d \n",k);
```

```
// obtendo o endereço da variável 'k'
// vamos usar o operador &
endereco_de_k = &k;
```

```
printf("\n Endereco da variavel 'k': %u \n",endereco_de_k);
printf("\n Endereco da variavel 'k': %u \n",&k);
printf("\n Endereco da variavel 'k': %p (em hexadecimal)\n",&k);
```

```
// sabemos que o scanf pede um endereço de memória
// o que acontece se passarmos o endereço da
// variável k?
printf("\n Digite o valor novo valor, a ser armazenado no endereço da variavel k: ");
// OBSERVE que não estamos usando & no scanf! Isso porque já temos o endereço
scanf("%d",endereco_de_k);
```

```
// mostrando o novo valor de 'k'
printf("\n\n Valor da variavel k, apos scanf de 'endereco_de_k': %d \n",k);
```

```
// mostrando o endereço de 'k' que deve permanecer o mesmo de antes
printf("\n Endereco da variavel 'k': %u \n",endereco_de_k);
printf("\n Endereco da variavel 'k': %u \n",&k);
printf("\n Endereco da variavel 'k': %p (em hexadecimal)\n",&k);
```

1		k	int
2	50		
3			
4			
5			
6		endereco_de_k	unsigned int
7	1		
8			
9			



```
int k;
unsigned int endereco_de_k;

// inicializando k
k = 10;
printf("\n Valor da variavel 'k': %d \n",k);
```

```
// obtendo o endereço da variável 'k'
// vamos usar o operador &
endereco_de_k = &k;
```

```
printf("\n Endereco da variavel 'k': %u \n",endereco_de_k);
printf("\n Endereco da variavel 'k': %u \n",&k);
printf("\n Endereco da variavel 'k': %p (em hexadecimal)\n",&k);
```

```
// sabemos que o scanf pede um endereço de memória
// o que acontece se passarmos o endereço da
// variável k?
```

```
printf("\n Digite o valor novo valor, a ser armazenado no endereço da variavel k: ");
// OBSERVE que não estamos usando & no scanf! Isso porque já temos o endereço
scanf("%d",endereco_de_k);
```

```
// mostrando o novo valor de 'k'
printf("\n\n Valor da variavel k, apos scanf de 'endereco_de_k': %d \n",k);
```

```
// mostrando o endereço de 'k' que deve permanecer o mesmo de antes
```

```
printf("\n Endereco da variavel 'k': %u \n",endereco_de_k);
```

```
printf("\n Endereco da variavel 'k': %u \n",&k);
```

```
printf("\n Endereco da variavel 'k': %p (em hexadecimal)\n",&k);
```

1		k	int
2	50		
3			
4			
5			
6		endereco_de_k	unsigned int
7	1		
8			
9			

Valor l

```
int k;
unsigned int endereco_de_k;

// inicializando k
k = 10;
printf("\n Valor da variavel 'k': %d \n",k);
```

```
// obtendo o endereço da variável 'k'
// vamos usar o operador &
endereco_de_k = &k;
```

```
printf("\n Endereco da variavel 'k': %u \n",endereco_de_k);
printf("\n Endereco da variavel 'k': %u \n",&k);
printf("\n Endereco da variavel 'k': %p (em hexadecimal)\n",&k);
```

```
// sabemos que o scanf pede um endereço de memória
// o que acontece se passarmos o endereço da
// variável k?
printf("\n Digite o valor novo valor, a ser armazenado no endereço da variavel k: ");
// OBSERVE que não estamos usando & no scanf! Isso porque já temos o endereço
scanf("%d",endereco_de_k);
```

```
// mostrando o novo valor de 'k'
printf("\n\n Valor da variavel k, apos scanf de 'endereco_de_k': %d \n",k);
```

```
// mostrando o endereço de 'k' que deve permanecer o mesmo de antes
printf("\n Endereco da variavel 'k': %u \n",endereco_de_k);
```

```
printf("\n Endereco da variavel 'k': %u \n",&k);
```

```
printf("\n Endereco da variavel 'k': %p (em hexadecimal)\n",&k);
```

1		k	int
2	50		
3			
4			
5			
6		endereco_de_k	unsigned int
7	1		
8			
9			

Endereço l

```
int k;
unsigned int endereco_de_k;

// inicializando k
k = 10;
printf("\n Valor da variavel 'k': %d \n",k);
```

```
// obtendo o endereço da variável 'k'
// vamos usar o operador &
endereco_de_k = &k;
```

```
printf("\n Endereco da variavel 'k': %u \n",endereco_de_k);
printf("\n Endereco da variavel 'k': %u \n",&k);
printf("\n Endereco da variavel 'k': %p (em hexadecimal)\n",&k);
```

```
// sabemos que o scanf pede um endereço de memória
// o que acontece se passarmos o endereço da
// variável k?
```

```
printf("\n Digite o valor novo valor, a ser armazenado no endereco da variavel k: ");
// OBSERVE que não estamos usando & no scanf! Isso porque já temos o endereço
scanf("%d",endereco_de_k);
```

```
// mostrando o novo valor de 'k'
printf("\n\n Valor da variavel k, apos scanf de 'endereco_de_k': %d \n",k);
```

```
// mostrando o endereço de 'k' que deve permanecer o mesmo de antes
printf("\n Endereco da variavel 'k': %u \n",endereco_de_k);
printf("\n Endereco da variavel 'k': %u \n",&k);
```

```
printf("\n Endereco da variavel 'k': %p (em hexadecimal)\n",&k);
```

1		k	int
2	50		
3			
4			
5			
6		endereco_de_k	unsigned int
7	1		
8			
9			

Endereço 0x1


```
int k;
unsigned int endereco_de_k;

// inicializando k
k = 10;
printf("\n Valor da variavel 'k': %d \n",k);

// obtendo o endereço da variável 'k'
// vamos usar o operador &
endereco_de_k = &k;

printf("\n Endereco da variavel 'k': %u \n",endereco_de_k);
printf("\n Endereco da variavel 'k': %u \n",&k);
printf("\n Endereco da variavel 'k': %p (em hexadecimal)\n",&k);

// sabemos que o scanf pede um endereço de memória
// o que acontece se passarmos o endereço da
// variável k?
printf("\n Digite o valor novo valor, a ser armazenado no endereço da variavel k: ");
// OBSERVE que não estamos usando & no scanf! Isso porque já temos o endereço
scanf("%d",endereco_de_k);

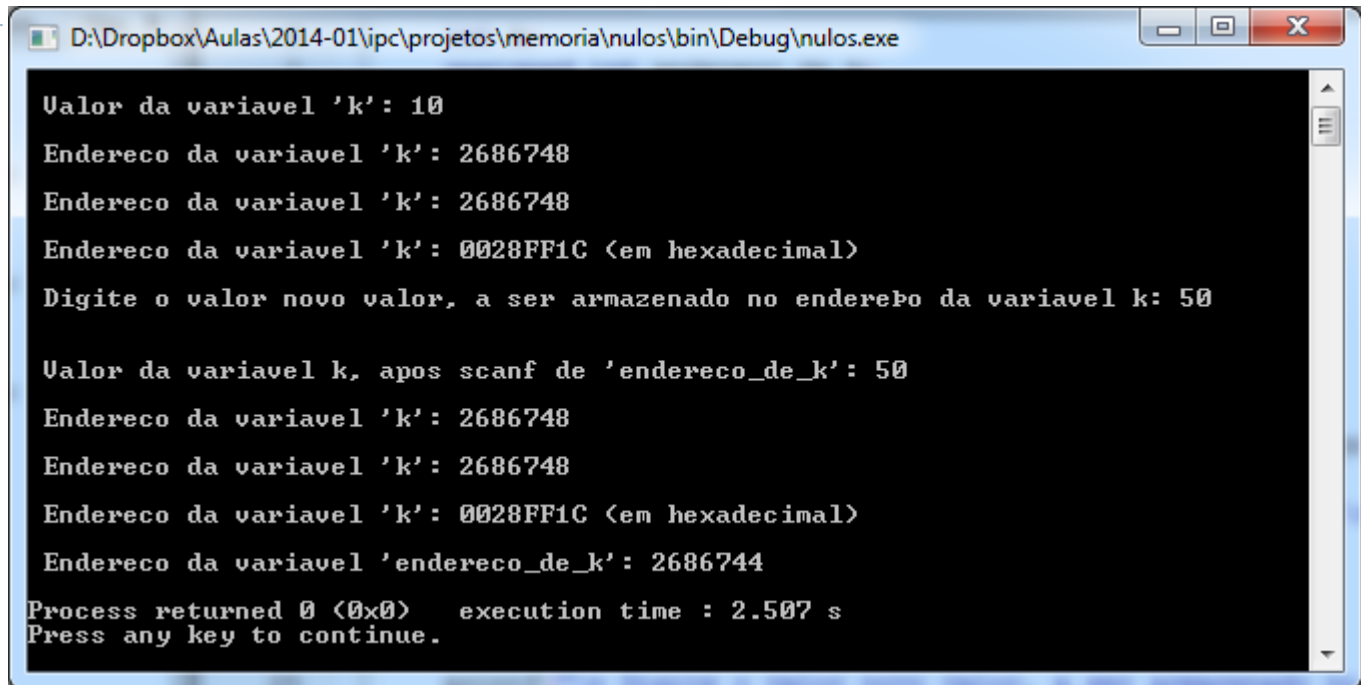
// mostrando o novo valor de 'k'
printf("\n\n Valor da variavel k, apos scanf de 'endereco_de_k': %d \n",k);

// mostrando o endereço de 'k' que deve permanecer o mesmo de antes
printf("\n Endereco da variavel 'k': %u \n",endereco_de_k);
printf("\n Endereco da variavel 'k': %u \n",&k);
printf("\n Endereco da variavel 'k': %p (em hexadecimal)\n",&k);
```



Execução real

► Saída



```
D:\Dropbox\Aulas\2014-01\ipc\projetos\memoria\nulos\bin\Debug\nulos.exe

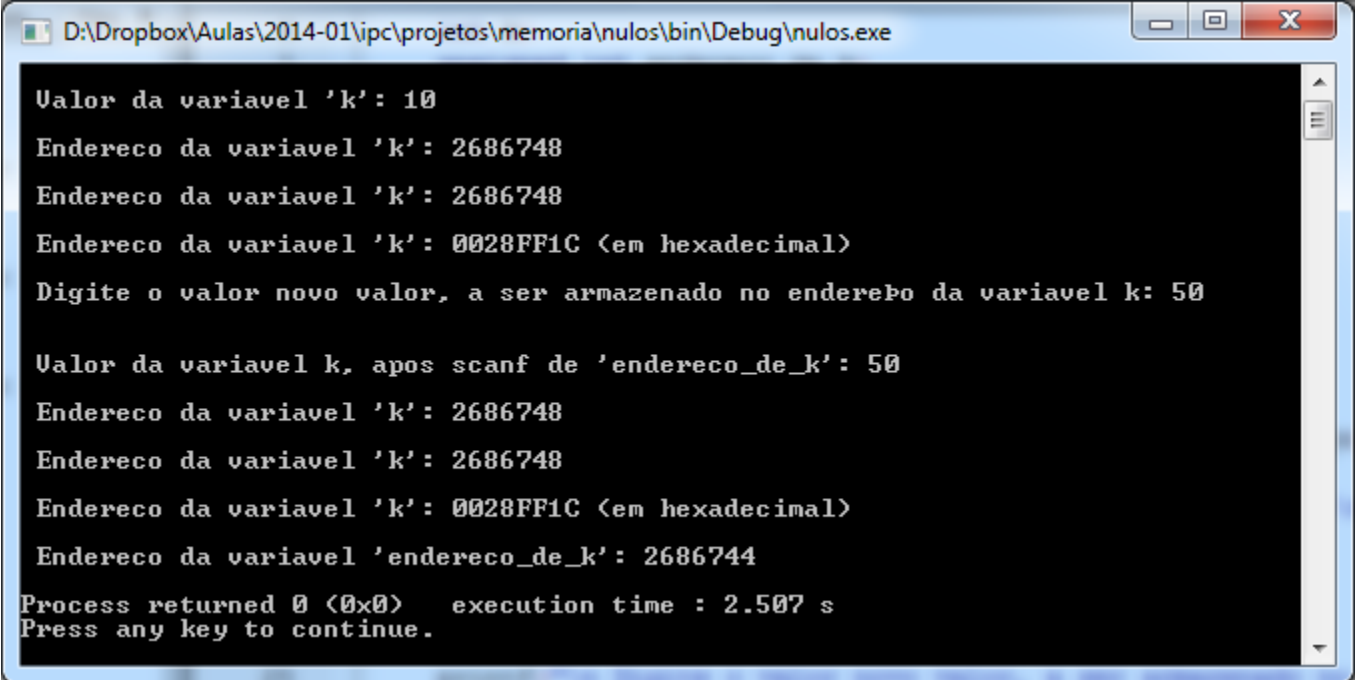
Valor da variavel 'k': 10
Endereco da variavel 'k': 2686748
Endereco da variavel 'k': 2686748
Endereco da variavel 'k': 0028FF1C <em hexadecimal>
Digite o valor novo valor, a ser armazenado no endereco da variavel k: 50

Valor da variavel k, apos scanf de 'endereco_de_k': 50
Endereco da variavel 'k': 2686748
Endereco da variavel 'k': 2686748
Endereco da variavel 'k': 0028FF1C <em hexadecimal>
Endereco da variavel 'endereco_de_k': 2686744
Process returned 0 (0x0)   execution time : 2.507 s
Press any key to continue.
```

- Observe no código que não usamos & para passar o endereço da variável `scanf("%d", endereco_de_k);`

Execução real

► Saída



```
D:\Dropbox\Aulas\2014-01\ipc\projetos\memoria\nulos\bin\Debug\nulos.exe

Valor da variavel 'k': 10
Endereco da variavel 'k': 2686748
Endereco da variavel 'k': 2686748
Endereco da variavel 'k': 0028FF1C <em hexadecimal>
Digite o valor novo valor, a ser armazenado no endereco da variavel k: 50

Valor da variavel k, apos scanf de 'endereco_de_k': 50
Endereco da variavel 'k': 2686748
Endereco da variavel 'k': 2686748
Endereco da variavel 'k': 0028FF1C <em hexadecimal>
Endereco da variavel 'endereco_de_k': 2686744
Process returned 0 (0x0)   execution time : 2.507 s
Press any key to continue.
```

- Observe que o valor da variável k mudou (era 10 e virou 50), mas seu endereço não (continua 2686748 ou 0028FF1C em hexadecimal)

Execução real

▶ Saída



```
D:\Dropbox\Aulas\2014-01\ipc\projetos\memoria\nulos\bin\Debug\nulos.exe

Valor da variavel 'k': 10
Endereco da variavel 'k': 2686748
Endereco da variavel 'k': 2686748
Endereco da variavel 'k': 0028FF1C <em hexadecimal>
Digite o valor novo valor, a ser armazenado no endereco da variavel k: 50

Valor da variavel k, apos scanf de 'endereco_de_k': 50
Endereco da variavel 'k': 2686748
Endereco da variavel 'k': 2686748
Endereco da variavel 'k': 0028FF1C <em hexadecimal>
Endereco da variavel 'endereco_de_k': 2686744

Process returned 0 (0x0)   execution time : 2.507 s
Press any key to continue.
```

- ▶ Também foi acrescentada a seguinte linha no código, para mostrar o endereço da variável `endereco_de_k`
 - ▶ `printf("\n Endereco da variavel 'endereco_de_k': %u \n",&endereco_de_k);`
- ▶ Usamos a variável `endereco_de_k` para guardar o endereço da variável `k`, mas lembre-se que `endereco_de_k` é uma variável e também ocupa espaço de memória e possui um endereço

Execução real

► Saída

```
D:\Dropbox\Aulas\2014-01\ipc\projetos\memoria\nulos\bin\Debug\nulos.exe

Valor da variavel 'k': 10
Endereco da variavel 'k': 2686748
Endereco da variavel 'k': 2686748
Endereco da variavel 'k': 0028FF1C <em hexadecimal>
Digite o valor novo valor, a ser armazenado no endereco da var

Valor da variavel k, apos scanf de 'endereco_de_k': 50
Endereco da variavel 'k': 2686748
Endereco da variavel 'k': 2686748
Endereco da variavel 'k': 0028FF1C <em hexadecimal>
Endereco da variavel 'endereco_de_k': 2686744

Process returned 0 (0x0)   execution time : 2.507 s
Press any key to continue.
```

2686742			
2686743			
2686744	2686748	endereco_de_k	unsigned int
2686745			
2686746			
2686747			
2686748	10	k	int
2686749			
2686750			
2686751			
2686752			
2686753			

2686742			
2686743			
2686744	2686748	endereco_de_k	unsigned int
2686745			
2686746			
2686747			
2686748	50	k	int
2686749			
2686750			
2686751			
2686752			
2686753			

Exercício

- ▶ Crie um programa que contenha a seguinte variável
 - ▶ `char nome[] = "Maria"`
- ▶ Utilizando a função `scanf`, altere o nome de Maria para Mario. Não use o operador `&` no `scanf`.



Ponteiros

- ▶ Vimos alguns tipos de dados em C:
 - ▶ int, float, double, char, unsigned int
 - ▶ Structs
- ▶ Na slides anteriores, usamos o operador `&` para trabalhar com endereços de memória, e usamos também variáveis `unsigned int` para guardar esses endereços
 - ▶ Essa não é a forma adequada para trabalhar com endereços. Em C podemos usar ponteiros



Ponteiros

- ▶ Vimos alguns tipos de dados em C:
 - ▶ int, float, double, char, unsigned int
 - ▶ Structs
- ▶ Ponteiro é um tipo de dado que serve para armazenar endereços de memória.
- ▶ Um ponteiro é uma variável como qualquer outra do programa – sua diferença é que ela não armazena um valor inteiro, real, caractere ou booleano. Ela serve para armazenar **endereços de memória** (que, no fundo, são valores inteiros sem sinal, como um unsigned int ou unsigned long long int).



Ponteiros

- ▶ Para declarar uma variável do tipo ponteiro, use a seguinte sintaxe:

`tipo_de_dado *nome_da_variável`

- ▶ Note que não existe um tipo de dado chamado *pointer*
- ▶ O que define o ponteiro é o sinal de `*` juntamente com um outro tipo de dado do programa
- ▶ Esse `tipo_de_dado` deve ser definido pois o ponteiro armazena um endereço de memória, e devemos especificar qual o tipo de dado que existe naquele endereço que ele armazena



Ponteiros

`tipo_de_dado *nome_da_variável`

- ▶ Exemplo de Ponteiros que são usados para guardar endereços de variáveis inteiras.

```
int *p;
```

```
int *proximo;
```

```
int *anterior;
```

```
int *abacaxi;
```

- ▶ Note que, a única diferença na declaração de uma variável do tipo ponteiro é a adição de um símbolo *****.



Ponteiros

- ▶ `int *p;`
- ▶ O espaço ocupado por um ponteiro em um sistema 32 bits é 4 bytes (o mesmo que um unsigned int)
- ▶ Sistemas 64 bits os ponteiros ocupam 8 bytes (o mesmo que um unsigned long long int)
- ▶ Lembre que o ponteiro também é uma variável, e portanto ocupa um espaço de memória



Ponteiros

► Mais exemplos:

```
double *valores;
```

```
double *estoque;
```

```
char *nome;
```

```
char *endereco;
```

```
float *temperatura;
```



Ponteiros – Espaço Ocupado

```
// armazena endereço de variáveis double  
double *valores;  
double *estoque;
```

```
// armazena endereço de variáveis char  
char *nome;  
char *endereco;
```

```
// armazena endereço de variáveis float  
float *temperatura;
```



Ponteiros – Espaço Ocupado

- ▶ Como o ponteiro armazena um endereço, ele ocupa uma quantidade de memória independente do tipo que ele aponta

`double *valores;` → 4 bytes
`double *estoque;` → 4 bytes

`char *nome;` → 4 bytes
`char *endereco;` → 4 bytes

`float *temperatura;` → 4 bytes

Ponteiros

- ▶ Como um ponteiro serve para receber um endereço de memória, podemos fazer, por exemplo, a seguinte operação:

```
int a = 40; // cria uma variável do tipo inteiro, chamada a,  
           //e inicializa com valor 40  
int *p; // cria uma variável do tipo ponteiro para inteiro,  
        //chamada p, e o conteúdo inicial é lixo  
p = &a; // faz p receber o endereço de a.  
        //Dizemos que p aponta para a
```



Ponteiros

45			
46			
47		a	int
48	40		
49			
50			
51			
52			
53			
54			
55			
56			

► Mapa de memória

- Sabendo que um ponteiro ocupa 4 bytes (sistema de 32 bits)

```
int a = 40; // cria uma variável do tipo inteiro, chamada a,  
           //e inicializa com valor 40
```

```
int *p; // cria uma variável do tipo ponteiro para inteiro,  
        //chamada p, e o conteúdo inicial é lixo  
p = &a; // faz p receber o endereço de a.  
        //Dizemos que p aponta para a
```



Ponteiros

► Mapa de memória

- Sabendo que um ponteiro ocupa 4 bytes (sistema de 32 bits)

45			
46			
47		a	int
48	40		
49			
50			
51		p	int *
52	lx		
53			
54			
55			
56			

```
int a = 40; // cria uma variável do tipo inteiro, chamada a,  
           //e inicializa com valor 40
```

```
int *p; // cria uma variável do tipo ponteiro para inteiro,  
        //chamada p, e o conteúdo inicial é lixo
```

```
p = &a; // faz p receber o endereço de a.  
        //Dizemos que p aponta para a
```



Ponteiros

► Mapa de memória

- Sabendo que um ponteiro ocupa 4 bytes (sistema de 32 bits)

45			
46			
47	40	a	int
48			
49			
50			
51	47	p	int *
52			
53			
54			
55			
56			

```
int a = 40; // cria uma variável do tipo inteiro, chamada a,
            //e inicializa com valor 40

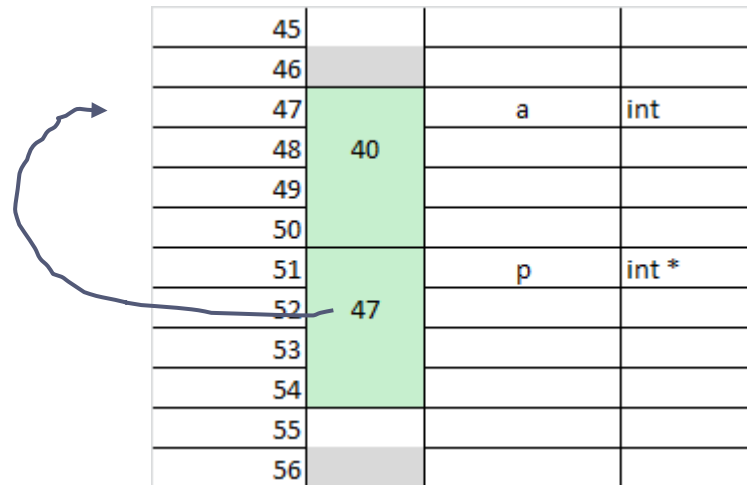
int *p; // cria uma variável do tipo ponteiro para inteiro,
        //chamada p, e o conteúdo inicial é lixo

p = &a; // faz p receber o endereço de a.
        //Dizemos que p aponta para a
```



Ponteiros

- ▶ Mapa de memória
 - ▶ “p aponta para a”



The diagram illustrates a memory map with a table of memory addresses and their contents. The table has four columns: Address, Value, Variable Name, and Data Type. The first column (Address) ranges from 45 to 56. The second column (Value) contains the value stored at each address. The third column (Variable Name) contains the name of the variable stored at each address. The fourth column (Data Type) contains the data type of the variable. The variable 'a' is of type 'int' and is located at address 47. The variable 'p' is of type 'int*' and is located at address 51. A blue arrow points from the value 47 in the 'Value' column at address 51 to the variable 'a' in the 'Variable Name' column at address 47, indicating that 'p' points to 'a'.

45			
46			
47		a	int
48	40		
49			
50			
51		p	int *
52	47		
53			
54			
55			
56			

Desreferenciar (*dereferencing*) um ponteiro

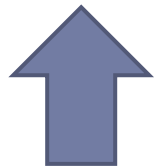
- ▶ Existem operadores específicos em C para trabalhar com ponteiros.
- ▶ Um desses operadores é o símbolo *
- ▶ Note que o mesmo símbolo é usado para declarar um ponteiro e também para multiplicação – mas essas operações não são relacionadas.
- ▶ O operador * serve para desreferenciar (*dereferencing*) um ponteiro – ou seja, ele retorna o **conteúdo** do endereço de memória que ele referencia/aponta.
- ▶ Ao usar o operador *, o tipo retornado será o mesmo tipo apontado pelo ponteiro



Desreferenciar (*dereferencing*) um ponteiro

► Exemplo

```
int a = 40; // cria uma variável do tipo inteiro, chamada a,  
           //e inicializa com valor 40  
int *p; // cria uma variável do tipo ponteiro para inteiro,  
        //chamada p, e o conteúdo inicial é lixo  
p = &a; // faz p receber o endereço de a.  
        //Dizemos que p aponta para a  
  
printf("\n O valor da variavel 'a' eh: %d", *p);
```



Saída:
O valor da variavel 'a' eh: 40

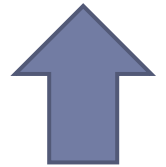


Desreferenciar (*dereferencing*)

▶ Exemplo

45			
46			
47	40	a	int
48			
49			
50			
51	47	p	int *
52			
53			
54			
55			
56			

```
printf("\n O valor da variavel 'a' eh: %d", *p);
```



- ▶ O programa vai até o ponteiro **p** e verifica para qual endereço ele aponta.
 - ▶ No exemplo, é o endereço 47
- ▶ Em seguida, o programa vai até o endereço 47 e busca a informação que está lá.
 - ▶ No caso, o valor contido no endereço 47 é o valor inteiro 40, que é mostrado como resposta no printf



Diferentes operadores

► Exemplo

45			
46			
47	40	a	int
48			
49			
50			
51	47	p	int *
52			
53			
54			
55			
56			

```
printf("\n O valor da variavel 'a' eh: %d", *p);
```

```
printf("\n O valor da variavel 'p' eh: %p", p);
```

```
printf("\n O endereço da var. 'p' eh: %p", &p);
```



Diferentes operadores

► Exemplo

45			
46			
47		a	int
48	40		
49			
50			
51		p	int *
52	47		
53			
54			
55			
56			

```
printf("\n O valor da variavel 'a' eh: %d", *p);
```

Saída: 0 valor da variavel 'a' eh: 40

```
printf("\n O valor da variavel 'p' eh: %p", p);
```

```
printf("\n O endereço da var. 'p' eh: %p", &p);
```



Diferentes operadores

► Exemplo

45			
46			
47		a	int
48	40		
49			
50			
51		p	int *
52	47		
53			
54			
55			
56			

```
printf("\n O valor da variavel 'a' eh: %d", *p);
```

```
printf("\n O valor da variavel 'p' eh: %p", p);
```

Saída: 0 valor da variavel 'p' eh: 47

```
printf("\n O endereço da var. 'p' eh: %p", &p);
```



Diferentes operadores

► Exemplo

45			
46			
47		a	int
48	40		
49			
50			
51		p	int *
52	47		
53			
54			
55			
56			

```
printf("\n O valor da variavel 'a' eh: %d", *p);
```

```
printf("\n O valor da variavel 'p' eh: %p", p);
```

```
printf("\n O endereço da var. 'p' eh: %p", &p);
```

Saída: 0 endereço da var. 'p' eh: 51



Desreferenciar (*dereferencing*) um ponteiro

- ▶ O desreferenciamento `*` pode ser usado para atribuição de valores às variáveis apontadas pelos ponteiros,

```
int a = 40;  
int *p;  
p = &a; // faz p receber o endereço de a  
*p = 59; // altera o conteúdo do endereço apontado por p
```

```
printf("\n O valor da variavel a eh: %d", a);
```

Saída:
O valor da variavel 'a' eh: 59



Desreferenciar (*dereferencing*) um ponteiro

- ▶ O desreferenciamento `*` pode ser usado para acessar valores às variáveis apontadas pelo ponteiro

45			
46			
47		a	int
48	40		
49			
50			
51		p	int *
52	47		
53			
54			
55			
56			

```
int a = 40;
```

```
int *p;
```

```
p = &a; // faz p receber o endereço de a
```

```
*p = 59; // altera o conteúdo do endereço apontado por p
```

```
printf("\n O valor da variavel a eh: %d", a);
```

Saída:

O valor da variavel 'a' eh: 59

Desreferenciar (*dereferencing*) um ponteiro

- ▶ O desreferenciamento `*` pode ser usado para alterar valores às variáveis apontadas pelo ponteiro

45			
46			
47		a	int
48	59		
49			
50			
51		p	int *
52	47		
53			
54			
55			
56			

```
int a = 40;
```

```
int *p;
```

```
p = &a; // faz p receber o endereço de a
```

```
*p = 59; // altera o conteúdo do endereço apontado por p
```

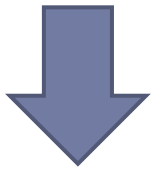
```
printf("\n O valor da variavel a eh: %d", a);
```

Saída:

O valor da variavel 'a' eh: 59

Desreferenciar (*dereferencing*)

▶ Exemplo



```
*p = 59; // altera o conteúdo do endereço apontado por p
```

45			
46			
47		a	int
48	59		
49			
50			
51		p	int *
52	47		
53			
54			
55			
56			

- ▶ O programa vai até o ponteiro **p** e verifica para qual endereço ele aponta.
 - ▶ No exemplo, é o endereço 47
- ▶ Em seguida, o programa vai até o endereço 47 e faz a atribuição do valor 59 neste local



Exemplos

```
// declarando as variáveis
```

```
double val;
```

```
float k;
```

```
// declarando os ponteiros
```

```
double *pval;
```

```
float *pk;
```

```
// atribuindo os valores das variáveis aos ponteiros
```

```
pval = &val;
```

```
pk = &k;
```

```
// alterando o conteúdo das variáveis via ponteiros
```

```
*pval = 33.45;
```

```
*pk = 2.4;
```



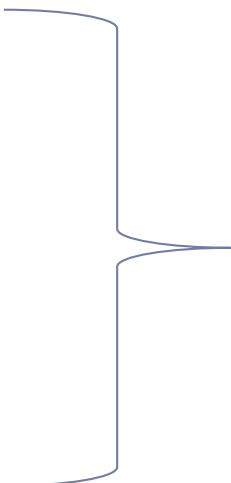
Exemplos

```
// declarando as variáveis  
double val;  
float k;
```

```
// declarando os ponteiros  
double *pval;  
float *pk;
```

```
// atribuindo os valores das variáveis ao  
pval = &val;  
pk = &k;
```

```
// alterando o conteúdo das variáveis via  
*pval = 33.45;  
*pk = 2.4;
```



4			
5	lx	val	double
6			
7			
8			
9			
10			
11			
12			
13	lx	k	float
14			
15			
16			
17	lx	pval	*double
18			
19			
20			
21	lx	pk	*float
22			
23			
24			
25			

Exemplos

```
// declarando as variáveis
```

```
double val;
```

```
float k;
```

```
// declarando os ponteiros
```

```
double *pval;
```

```
float *pk;
```

```
// atribuindo os valores das variáveis aos ponteiros
```

```
pval = &val;
```

```
pk = &k;
```

```
// alterando o conteúdo das variáveis via ponteiros
```

```
*pval = 33.45;
```

```
*pk = 2.4;
```

4			
5	lx	val	double
6			
7			
8			
9			
10			
11			
12			
13	lx	k	float
14			
15			
16			
17	5	pval	*double
18			
19			
20			
21	13	pk	*float
22			
23			
24			
25			

Exemplos

```
// declarando as variáveis
```

```
double val;
```

```
float k;
```

```
// declarando os ponteiros
```

```
double *pval;
```

```
float *pk;
```

```
// atribuindo os valores das variáveis ao
```

```
pval = &val;
```

```
pk = &k;
```

```
// alterando o conteúdo das variáveis via
```

```
*pval = 33.45;
```

```
*pk = 2.4;
```

4			
5	33.45	val	double
6			
7			
8			
9			
10			
11			
12			
13	2.4	k	float
14			
15			
16			
17	5	pval	*double
18			
19			
20			
21	13	pk	*float
22			
23			
24			
25			

Está correto?

```
double *preco;
```

```
*preco = 50.0;
```



Está correto?

```
double *preco;
```

```
*preco = 50.0;
```

- ▶ Não houve alocação para guardar um número 'double'
- ▶ houve somente alocação para guardar um ponteiro para double



Está correto?

```
double *preco;
```

```
*preco = 50.0;
```

68			
69	lx	preco	*double
70			
71			
72			
73			
74			

- ▶ Ao declarar uma variável, o conteúdo dela é lixo

Está correto?

```
double *preco;
```

```
*preco = 50.0;
```

67			
68			
69		preco	*double
70	13		
71			
72			
73			
74			

- ▶ Ao declarar uma variável, o conteúdo dela é lixo
- ▶ Suponha que o lixo seja o valor 13

Está correto?

```
double *preco;  
*preco = 50.0;
```

- ▶ O programa tentará alterar o que está no endereço 13, e poderá travar
- ▶ Pode alterar outras variáveis

67			
68			
69		preco	*double
70	13		
71			
72			
73			
74			

11			
12			
13		k	float
14	2.4		
15			
16			
17		pval	*double
18	5		
19			
20			
21		pk	*float
22	13		
23			
24			
25			

Está correto?

```
double *preco;  
*preco = 50.0;
```

- ▶ O programa tentará alterar o que está no endereço 13, e poderá travar

Memória invadida (8 bytes pois preço é *double)

- ▶ Pode alterar outras variáveis

67			
68			
69		preco	*double
70	13		
71			
72			
73			
74			

11			
12			
13		k	float
14	2.4		
15			
16			
17		pval	*double
18	5		
19			
20			
21		pk	*float
22	13		
23			
24			
25			

Inicialização

- ▶ Um ponteiro pode ter o valor especial **NULL** que é o endereço de nenhum lugar.
 - ▶ Ex: `int *p = NULL;`
- ▶ Pode-se usar o valor 0 (zero) ao invés de **NULL**

Endereço	Blocos (1 byte)	Nome variável	Tipo
0 / NULL	indefinido	----	----
1			
2			

28			
29			
30	NULL	p	*int
31			
32			
33			
34			

```
int k;  
unsigned int endereco_de_k;
```

```
// inicializando k  
k = 10;  
printf("\n Valor da variavel 'k': %d \n",k);
```

```
// obtendo o endereço da variável 'k'  
// vamos usar o operador &  
endereco_de_k = &k;
```

```
printf("\n Endereco da variavel 'k': %u \n",endereco_de_k);  
printf("\n Endereco da variavel 'k': %u \n",&k);  
printf("\n Endereco da variavel 'k': %p (em hexadecimal)\n",&k);
```

```
// sabemos que o scanf pede um endereço de memória  
// o que acontece se passarmos o endereço da  
// variável k?  
printf("\n Digite o valor novo valor, a ser armazenado no endereço da variavel k: ");  
// OBSERVE que não estamos usando & no scanf! Isso porque já temos o endereço  
scanf("%d",endereco_de_k);
```

```
// mostrando o novo valor de 'k'  
printf("\n\n Valor da variavel k, apos scanf de 'endereco_de_k': %d \n",k);
```

```
// mostrando o endereço de 'k' que deve permanecer o mesmo de antes  
printf("\n Endereco da variavel 'k': %u \n",endereco_de_k);  
printf("\n Endereco da variavel 'k': %u \n",&k);  
printf("\n Endereco da variavel 'k': %p (em hexadecimal)\n",&k);
```

Altere esse programa
utilizando ponteiros (que é a
forma correta de se fazer)



```
int k;  
int *endereco_de_k;
```



```
// inicializando k  
k = 10;  
printf("\n Valor da variavel 'k': %d \n",k);
```

```
// obtendo o endereço da variável 'k'  
// vamos usar o operador &  
endereco_de_k = &k;
```

```
printf("\n Endereco da variavel 'k': %u \n",endereco_de_k);  
printf("\n Endereco da variavel 'k': %u \n",&k);  
printf("\n Endereco da variavel 'k': %p (em hexadecimal)\n",&k);
```

```
// sabemos que o scanf pede um endereço de memória  
// o que acontece se passarmos o endereço da  
// variável k?  
printf("\n Digite o valor novo valor, a ser armazenado no endereço da variavel k: ");  
// OBSERVE que não estamos usando & no scanf! Isso porque já temos o endereço  
scanf("%d",endereco_de_k);
```

```
// mostrando o novo valor de 'k'  
printf("\n\n Valor da variavel k, apos scanf de 'endereco_de_k': %d \n",k);
```

```
// mostrando o endereço de 'k' que deve permanecer o mesmo de antes  
printf("\n Endereco da variavel 'k': %u \n",endereco_de_k);  
printf("\n Endereco da variavel 'k': %u \n",&k);  
printf("\n Endereco da variavel 'k': %p (em hexadecimal)\n",&k);
```

Forma correta



Operações com ponteiros

▶ Atribuição

- ▶ p1 aponta para o mesmo lugar que p2;

p1 = p2;

- ▶ a variável apontada por p1 recebe o mesmo conteúdo da variável apontada por p2;

***p1 = *p2;**



Operações com ponteiros

- ▶ Apenas duas operações aritméticas podem ser utilizadas com o endereço armazenado pelo ponteiro: adição e subtração
- ▶ podemos apenas somar e subtrair valores INTEIROS
 - ▶ `p++`; soma +1 no endereço armazenado no ponteiro.
 - ▶ `p--`; subtrai 1 no endereço armazenado no ponteiro.
 - ▶ `p = p+15`; soma +15 no endereço armazenado no ponteiro.



Operações com ponteiros

- ▶ As operações de adição e subtração no endereço dependem do tipo de dado que o ponteiro aponta.
- ▶ Considere um ponteiro para inteiro, **int ***. O tipo **int** ocupa um espaço de 4 bytes na memória.
- ▶ Assim, nas operações de adição e subtração são adicionados/subtraídos 4 bytes por incremento/decremento, pois esse é o tamanho de um inteiro na memória e, portanto, é também o valor mínimo necessário para sair dessa posição reservada de memória



Operações com ponteiros

- ▶ **Operações ilegais com ponteiros**
 - ▶ Dividir ou multiplicar ponteiros;
 - ▶ Somar o endereço de dois ponteiros;
 - ▶ Não se pode adicionar ou subtrair float ou double de ponteiros.



Operações com ponteiros

- ▶ Já sobre seu conteúdo apontado, valem todas as operações
 - ▶ $(*p)++$; incrementar o conteúdo da variável apontada pelo ponteiro p ;
 - ▶ $*p = (*p) * 15$; multiplica o conteúdo da variável apontada pelo ponteiro p por 15;



Operações com ponteiros

▶ Operações relacionais

- ▶ `==` e `!=` para saber se dois ponteiros são iguais ou diferentes.

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  int main(){
04      int *p, *p1, x, y;
05      p = &x;
06      p1 = &y;
07      if(p == p1)
08          printf("Ponteiros iguais\n");
09      else
10          printf("Ponteiros diferentes\n");
11      system("pause");
12      return 0;
13  }
```



Operações com ponteiros

► Operações relacionais

- $>$, $<$, $>=$ e $<=$ para saber qual ponteiro aponta para uma posição mais alta na memória.

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  int main(){
04      int *p, *p1, x, y;
05      p = &x;
06      p1 = &y;
07      if(p > p1)
08          printf("O ponteiro p aponta para uma posicao a
09             frente de p1\n");
10      else
11          printf("O ponteiro p NAO aponta para uma posicao
12             a frente de p1\n");
13      system("pause");
14      return 0;
15  }
```



Ponteiros Genéricos

- ▶ Normalmente, um ponteiro aponta para um tipo específico de dado.
 - ▶ Um ponteiro genérico é um ponteiro que pode apontar para qualquer tipo de dado.
- ▶ Declaração

```
void *nome_ponteiro;
```



Exemplo

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int a = 10;
    double d = 30;

    void *p;

    // atribuindo o endereço de 'a' ao ponteiro void
    p = &a;

    // mostrando o conteúdo do endereço apontado por p (no caso, a var. 'a')
    printf("Valor de a: %d", *p);
}
```



- ▶ Erro em tempo de compilação:
 - ▶ **ERROR: invalid use of void expression**



Exemplo


```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int a = 10;
    double d = 30;

    void *p;

    // atribuindo o endereço de 'a' ao ponteiro void
    p = &a;

    printf("Valor de a: %d", *(int *)p);
}
```



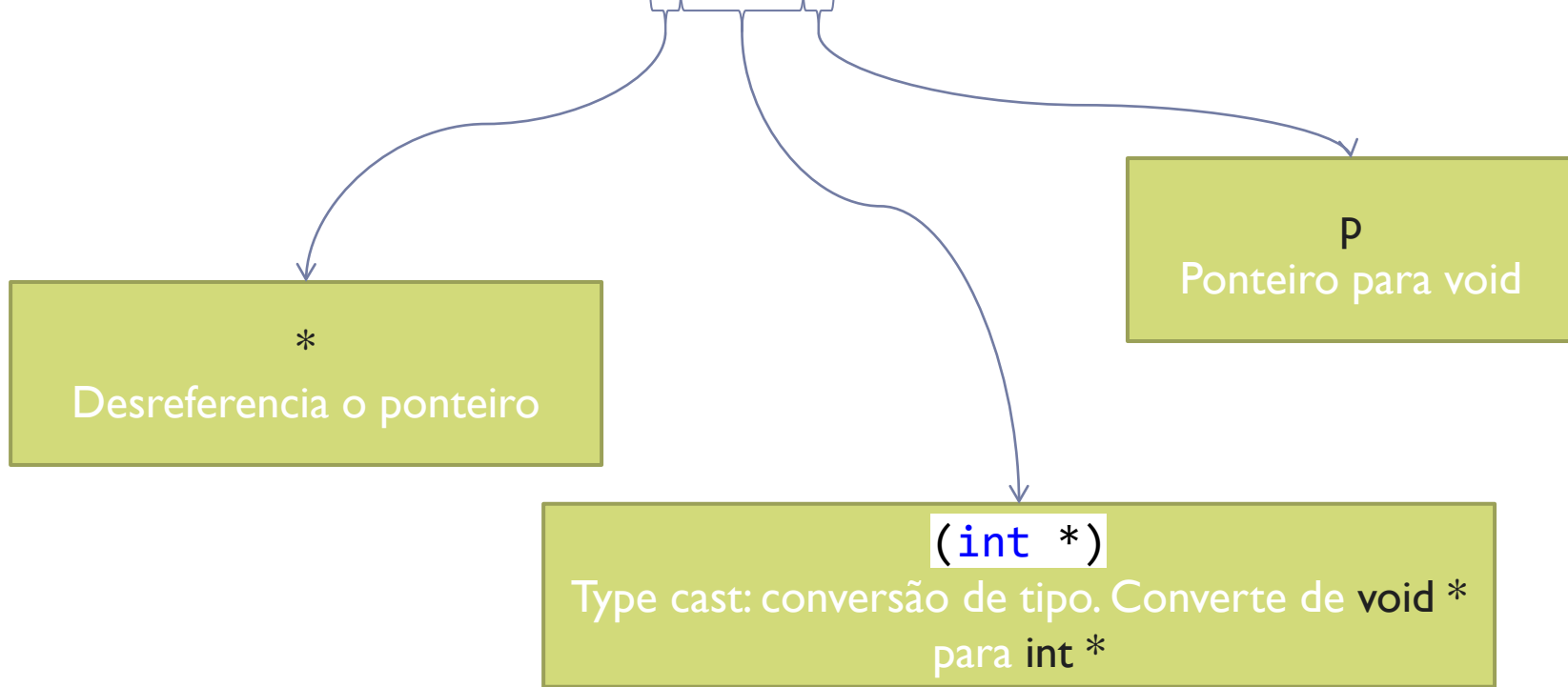
- ▶ Para usar *void, é necessário fazer a conversão para o tipo do ponteiro que o void aponta. No caso, é um ponteiro para inteiro
 - ▶ Use para converter: (int *)



Exemplo

► Observe com atenção

```
printf("Valor de a: %d", *(int *)p);
```



Exemplo

- Mudando o apontamento de p de um inteiro para um double

```
int a = 10;
double d = 30;

void *p;

// atribuindo o endereço de 'a' (inteiro) ao ponteiro void
p = &a;

// mostrando o conteúdo do endereço apontado por p (no caso, a var. 'a')
// printf("Valor de a: %d", *p); // errado!

// mostrando o conteúdo do endereço apontado por p (no caso, a var. 'a')
printf("Valor de a: %d", *(int *)p);

// atribuindo o endereço de 'd' (double) ao ponteiro void
p = &d;

// mostrando o conteúdo do endereço apontado por p (no caso, a var. 'd')
printf("Valor de b: %f", *(double *)p);
```



Ponteiros Genéricos

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  int main(){
04      void *pp;
05      int *p1, p2 = 10;
06      p1 = &p2;
07      //recebe o endereço de um inteiro
08      pp = &p2;
09      printf("Endereco em pp: %p \n",pp);
10      //recebe o endereço de um ponteiro para inteiro
11      pp = &p1;
12      printf("Endereco em pp: %p \n",pp);
13      //recebe o endereço guardado em p1 (endereço de p2)
14      pp = p1;
15      printf("Endereco em pp: %p \n",pp);
16      system("pause");
17      return 0;
18  }
```



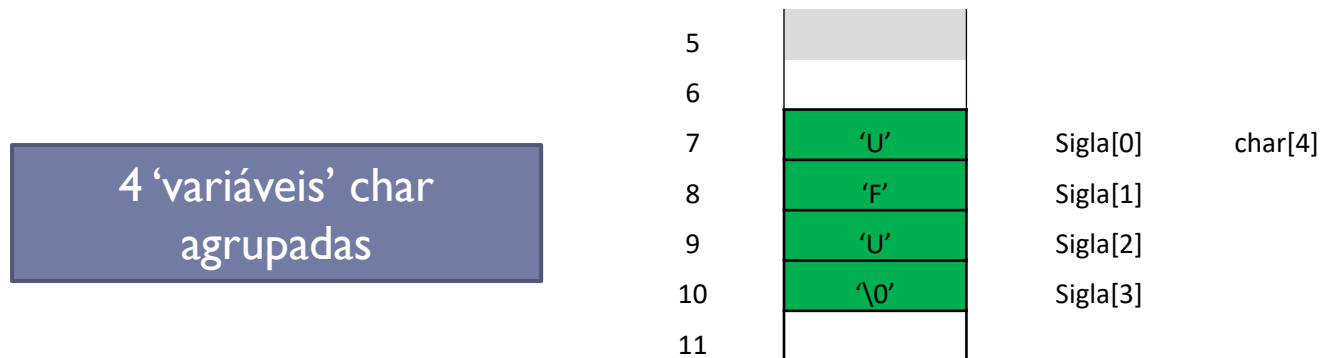
Ponteiros Genéricos

- ▶ Para acessar o conteúdo de um ponteiro genérico é preciso antes convertê-lo para o tipo de ponteiro com o qual se deseja trabalhar via **type cast** – *conversão de tipo*

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  int main(){
04      void *pp;
05      int p2 = 10;
06      // ponteiro genérico recebe o endereço de um
    inteiro
07      pp = &p2;
08      //enta acessar o conteúdo do ponteiro genérico
09      printf("Conteúdo: %d\n",*pp); //ERRO
10      //converte o ponteiro genérico pp para (int *)
    antes de acessar seu conteúdo.
11      printf("Conteúdo: %d\n",*(int*)pp); //CORRETO
12      system("pause");
13      return 0;
14  }
```

Ponteiros e Arrays

- ▶ **Ponteiros e arrays possuem uma ligação muito forte.**
 - ▶ Arrays são agrupamentos de dados do mesmo tipo na memória.



Ponteiros e Arrays

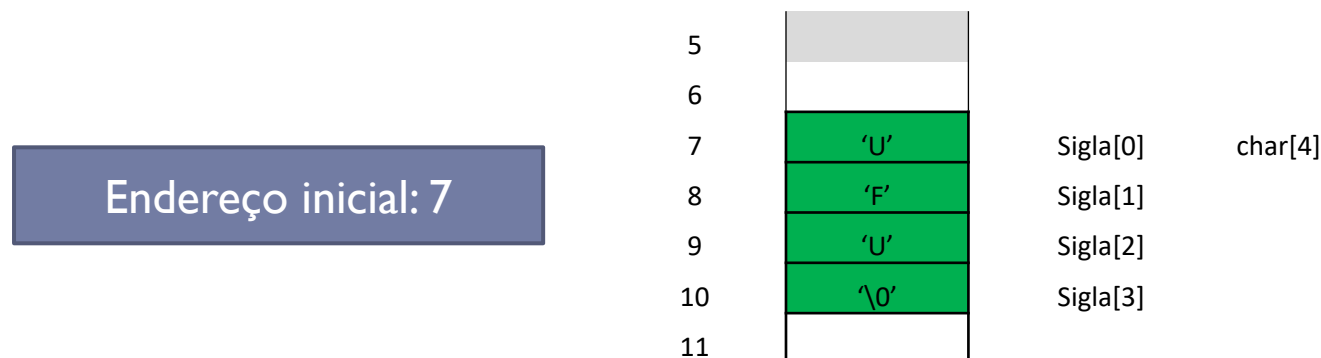
- ▶ **Ponteiros e arrays possuem uma ligação muito forte.**
 - ▶ Quando declaramos um array, informamos ao computador para reservar uma certa quantidade de memória a fim de armazenar os elementos do array de forma sequencial.

4 x 1 byte = 4 bytes

5			
6			
7	'U'	Sigla[0]	char
8	'F'	Sigla[1]	char
9	'U'	Sigla[2]	char
10	'\0'	Sigla[3]	char
11			

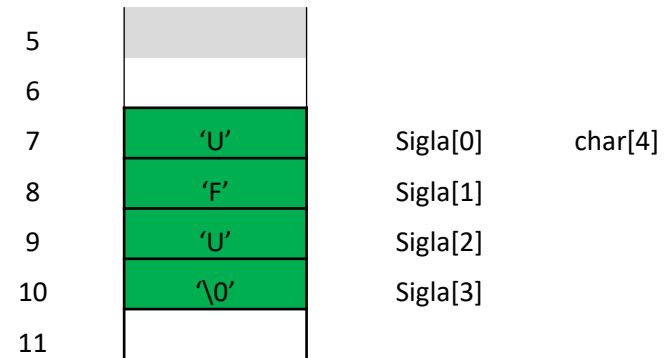
Ponteiros e Arrays

- ▶ **Ponteiros e arrays possuem uma ligação muito forte.**
 - ▶ Como resultado dessa operação, o computador nos devolve um ponteiro que aponta para o começo dessa sequência de bytes na memória.



Ponteiros e Arrays

- ▶ Em C, o nome do array (sem índice) é apenas **um ponteiro** que aponta para o primeiro elemento do array.



```
char Sigla[4] = "UFU";
```

```
// mostrando o endereço da posição 0 do vetor
```

```
printf("\n Posicao de indice zero do vetor (Sigla[0]): %u",&Sigla[0]);
```

```
// mostrando o endereço da posição 0 do vetor
```

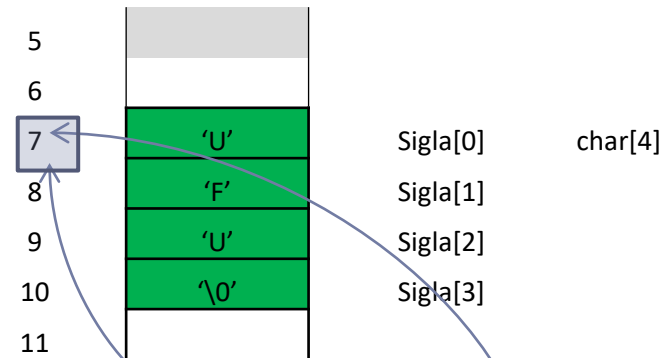
```
printf("\n Posicao de indice zero do vetor (Sigla): %u",Sigla);
```



Ponteiros e Arrays

- ▶ Em C, o nome do array (sem índice) é apenas **um ponteiro** que aponta para o primeiro elemento do array.

- ▶ **&Sigla[0]** é igual **Sigla**



```
char Sigla[4] = "UFU";
```

```
// mostrando o endereço da posição 0 do vetor
```

```
printf("\n Posicao de indice zero do vetor (Sigla[0]): %u", &Sigla[0]);
```

```
// mostrando o endereço da posição 0 do vetor
```

```
printf("\n Posicao de indice zero do vetor (Sigla): %u", Sigla);
```

Ponteiros e Arrays

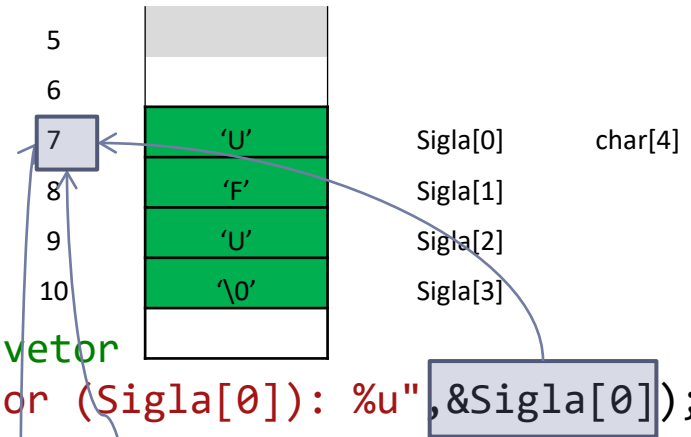
- ▶ O endereço do vetor é o próprio vetor
 - ▶ **&Sigla** é igual **Sigla**

```
char Sigla[4] = "UFU";
```

```
// mostrando o endereço da posição 0 do vetor  
printf("\n Posicao de indice zero do vetor (Sigla[0]): %u", &Sigla[0]);
```

```
// mostrando o endereço da posição 0 do vetor  
printf("\n Nome do vetor (Sigla): %u", Sigla);
```

```
// mostrando o endereço da posição 0 do vetor  
printf("\n Endereço do vetor (&Sigla): %u", &Sigla);
```



Ponteiros e Arr

```
D:\Dropbox\Aulas\2014-01\ipc\projetos\memoria\ArraysPonteiros\bin\Debug\Arra

Posicao de indice zero do vetor <Sigla[0]>: 2686748
Nome do vetor <Sigla>: 2686748
Endereco do vetor (&Sigla): 2686748
Process returned 0 (0x0)   execution time : 0.030 s
Press any key to continue.
```

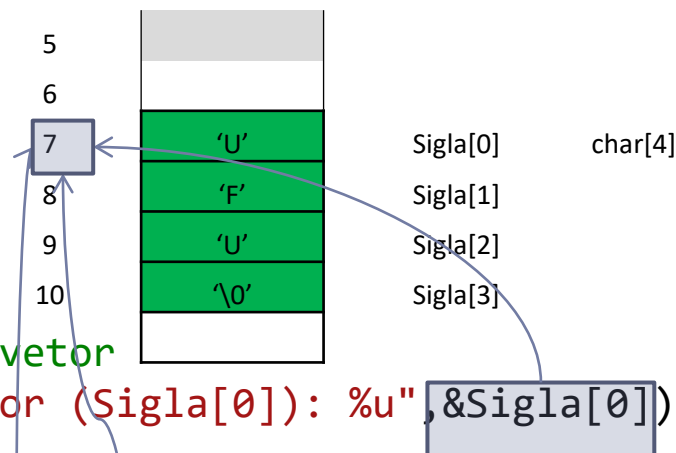
- ▶ O endereço do vetor é o próprio vetor
 - ▶ **&Sigla** é igual **Sigla**

```
char Sigla[4] = "UFU";
```

```
// mostrando o endereço da posição 0 do vetor
printf("\n Posicao de indice zero do vetor (Sigla[0]): %u", &Sigla[0]);
```

```
// mostrando o endereço da posição 0 do vetor
printf("\n Nome do vetor (Sigla): %u", Sigla);
```

```
// mostrando o endereço da posição 0 do vetor
printf("\n Endereço do vetor (&Sigla): %u", &Sigla);
```



Ponteiros e Arrays

- ▶ O nome do array (sem índice) é apenas um ponteiro que aponta para o primeiro elemento do array.

```
char Sigla[4] = "UFU";  
char *p;
```

```
p = Sigla;
```

67			
68			
69	79	p	char *
70			
71			
72			
73			
74			
75			
76			
77			
78			
79	'U'	Sigla[0]	char
80	'F'	Sigla[1]	char
81	'U'	Sigla[2]	char
82	'\0'	Sigla[3]	char
83			
84			

Ponteiros e Arrays

- ▶ Nesse exemplo

```
char Sigla[4] = "UFU";
```

```
char *p;
```

```
p = Sigla;
```

- ▶ Temos que:

- ▶ ***p** é equivalente a **Sigla[0]**;
- ▶ **Sigla[indice]** é equivalente a ***(p+indice)**;
- ▶ **Sigla** é equivalente a **&Sigla[0]**;
- ▶ **&Sigla[indice]** é equivalente a **(Sigla + indice)**;



Ponteiros e Arrays

Exemplo: acessando arrays utilizando ponteiros

Usando array		Usando ponteiro	
01	#include <stdio.h>	01	#include <stdio.h>
02	#include <stdlib.h>	02	#include <stdlib.h>
03	int main(){	03	int main(){
04	int vet[5]= {1,2,3,4,5};	04	int vet[5]= {1,2,3,4,5};
05	int *p = vet;	05	int *p = vet;
06	int i;	06	int i;
07	for (i = 0;i < 5;i++)	07	for (i = 0;i < 5;i++)
08	printf("%d\n",p[i]);	08	printf("%d\n",*(p+i));
09	system("pause");	09	system("pause");
10	return 0;	10	return 0;
11	}	11	}



Ponteiros e Arrays

- ▶ Os colchetes [] substituem o uso conjunto de operações aritméticas e de acesso ao conteúdo (operador “*”) no acesso ao conteúdo de uma posição de um array ou ponteiro.
 - ▶ O valor entre colchetes é o deslocamento a partir da posição inicial. Nesse caso, **p[2]** equivale a ***(p+2)**.

```
#include <stdio.h>
#include <stdlib.h>
int main (){
    int vet[5] = {1,2,3,4,5};
    int *p;
    p = vet;
    printf (“Terceiro elemento: %d ou %d”,p[2],*(p+2));
    system(“pause”);
    return 0;
}
```



Ponteiros e Structs

- ▶ Existe um operador específico para trabalhar com desreferenciamento de ponteiros para struct
- ▶ Operador ->
 - ▶ Como usar: ponteiro -> membro_da_struct
 - ▶ Exemplo
 - ▶ (*p).x é igual a p->x

Operators (grouped by precedence)

structure member operator	<i>name.member</i>
structure pointer	<i>pointer->member</i>
increment, decrement	<i>++, --</i>
plus, minus, logical not, bitwise not	<i>+, -, !, ~</i>
indirection via pointer, address of object	<i>*pointer, &name</i>
cast expression to type	<i>(type) expr</i>
size of an object	<i>sizeof</i>

```
int main(){  
  
    struct aluno joao;  
  
    joao.num_aluno = 10;  
    joao.nota1 = 10;  
    joao.nota2 = 4.4;  
    joao.nota3 = 7;  
    joao.media = (joao.nota1 + joao.nota2 + joao.nota3)/3.0;  
  
    struct aluno *pa;  
  
    pa = &joao;  
  
    printf("Numero aluno: %d\n", (*pa).num_aluno);  
    printf("Nota 1: %f\n", (*pa).nota1);  
    printf("Nota 2: %f\n", (*pa).nota2);  
    printf("Nota 3: %f\n", (*pa).nota3);  
    printf("Media 3: %f\n", (*pa).media);  
  
    // comandos equivalentes  
    printf("\n");  
    printf("Numero aluno: %d\n", pa->num_aluno);  
    printf("Nota 1: %f\n", pa->nota1);  
    printf("Nota 2: %f\n", pa->nota2);  
    printf("Nota 3: %f\n", pa->nota3);  
    printf("Media 3: %f\n", pa->media);  
}
```

```
int main(){

    struct aluno joao;

    joao.num_aluno = 10;
    joao.nota1 = 10;
    joao.nota2 = 4.4;
    joao.nota3 = 7;
    joao.media = (joao.nota1 + joao.nota2 + joao.nota3)/3.0;

    struct aluno *pa;

    pa = &joao;

    printf("Numero aluno: %d\n", (*pa).num_aluno);
    printf("Nota 1: %f\n", (*pa).nota1);
    printf("Nota 2: %f\n", (*pa).nota2);
    printf("Nota 3: %f\n", (*pa).nota3);
    printf("Media 3: %f\n", (*pa).media);

    // comandos equivalentes
    printf("\n");
    printf("Numero aluno: %d\n", pa->num_aluno);
    printf("Nota 1: %f\n", pa->nota1);
    printf("Nota 2: %f\n", pa->nota2);
    printf("Nota 3: %f\n", pa->nota3);
    printf("Media 3: %f\n", pa->media);
}
```

```
struct aluno {
    int num_aluno;
    float nota1, nota2, nota3;
    float media;
};
```

Desreferenciamento (*) do ponteiro pa, seguido pelo operador . (ponto) da struct, para acessar os membros

```
int main(){
```

```
    struct aluno joao;
```

```
    joao.num_aluno = 10;
```

```
    joao.nota1 = 10;
```

```
    joao.nota2 = 4.4;
```

```
    joao.nota3 = 7;
```

```
    joao.media = (joao.nota1 + joao.nota2 + joao.nota3)/3.0;
```

```
    struct aluno *pa;
```

```
    pa = &joao;
```

```
    printf("Numero aluno: %d\n", (*pa).num_aluno);
```

```
    printf("Nota 1: %f\n", (*pa).nota1);
```

```
    printf("Nota 2: %f\n", (*pa).nota2);
```

```
    printf("Nota 3: %f\n", (*pa).nota3);
```

```
    printf("Media 3: %f\n", (*pa).media);
```

```
    // comandos equivalentes
```

```
    printf("\n");
```

```
    printf("Numero aluno: %d\n", pa->num_aluno);
```

```
    printf("Nota 1: %f\n", pa->nota1);
```

```
    printf("Nota 2: %f\n", pa->nota2);
```

```
    printf("Nota 3: %f\n", pa->nota3);
```

```
    printf("Media 3: %f\n", pa->media);
```

```
struct aluno {  
    int num_aluno;  
    float nota1, nota2, nota3;  
    float media;  
};
```

Note que `*pa.nota1` estaria errado pois o ponto (.) tem precedência sobre o operador de desreferenciamento (*)


```

int main(){
    struct aluno joao;

    joao.num_aluno = 10;
    joao.nota1 = 10;
    joao.nota2 = 4.4;
    joao.nota3 = 7;
    joao.media = (joao.nota1 + joao.nota2 + joao.nota3)/3.0;

    struct aluno *pa;

    pa = &joao;

    printf("Numero aluno: %d\n", (*pa).num_aluno);
    printf("Nota 1: %f\n", (*pa).nota1);
    printf("Nota 2: %f\n", (*pa).nota2);
    printf("Nota 3: %f\n", (*pa).nota3);
    printf("Media 3: %f\n", (*pa).media);

    // comandos equivalentes
    printf("\n");
    printf("Numero aluno: %d\n", pa->num_aluno);
    printf("Nota 1: %f\n", pa->nota1);
    printf("Nota 2: %f\n", pa->nota2);
    printf("Nota 3: %f\n", pa->nota3);
    printf("Media 3: %f\n", pa->media);
}

```

```

struct aluno {
    int num_aluno;
    float nota1, nota2, nota3;
    float media;
};

```

Operador -> é equivalente aos comandos acima e mais fácil de usar, além de possuir alta precedência