



APONTADORES

CAP ENGEL Diogo Silva
dasilva@academiafa.edu.pt

Um **apontador** aponta para uma determinada zona de memória.

Um **apontador** é uma variável cujo conteúdo é um endereço de memória.

```
tipo * nomeDaVariavel;
```

declaração de
apontador

```
tipo * nomeDaVariavel;
```

declaração de
apontador

```
char a, b, *p;
```

```
int idade, *p_idade;
```

```
tipo * nomeDaVariavel;
```

declaração de
apontador

```
char a, b, *p;
```

```
int idade, *p_idade;
```

Apontadores

Variáveis
“normais”

```
tipo * nomeDaVariavel;
```

declaração de
apontador

```
char a, b, *p;  
int idade, *p_idade;
```

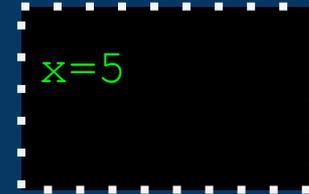
Apontadores

```
p = &a;  
p_idade = &idade;
```

Variáveis
“normais”

O operador **&** devolve o endereço de uma variável.

```
int x = 5;  
int *p = &x;  
  
printf("x=%d\n", x);
```

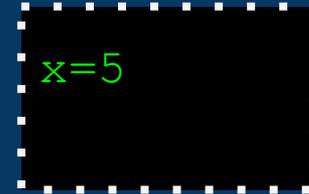


x=5

```
int x = 5;
int *p = &x;

printf("x=%d\n", x);

*p = 6;
```



O operador `*` obtém o valor armazenado na posição de memória indicada pelo apontador.

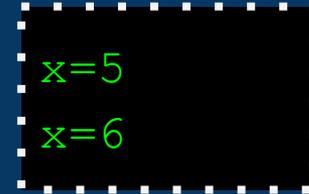
Neste exemplo usar `x` ou `*p` tem o mesmo efeito.

```
int x = 5;
int *p = &x;

printf("x=%d\n", x);

*p = 6;

printf("x=%d\n", x);
```



O operador `*` obtém o valor armazenado na posição de memória indicada pelo apontador.

Neste exemplo usar `x` ou `*p` tem o mesmo efeito.

```
int x = 5, y=7;
```

```
int *p;
```

variável	p		x		y	
valor	-----		5		7	
endereço	1000	1001	1002	1003	1004	1005

```
int x = 5, y=7;
```

```
int *p;
```

variável	p		x		y	
valor	-----		5		7	
endereço	1000	1001	1002	1003	1004	1005

```
p = &x;
```

variável	p		x		y	
valor	1002		7		7	
endereço	1000	1001	1002	1003	1004	1005

```
int x = 5, y=7;
```

```
int *p;
```

variável	p		x		y	
valor	-----		5		7	
endereço	1000	1001	1002	1003	1004	1005

```
p = &x;
```

variável	p		x		y	
valor	1002		5		7	
endereço	1000	1001	1002	1003	1004	1005

```
*p = y;
```

variável	p		x		y	
valor	1002		7		7	
endereço	1000	1001	1002	1003	1004	1005

```
int x = 5, y=7;
```

```
int *p;
```

variável	p		x		y	
valor	-----		5		7	
endereço	1000	1001	1002	1003	1004	1005

```
p = &x;
```

```
*p = y;
```

variável	p		x		y	
valor	1002		7		7	
endereço	1000	1001	1002	1003	1004	1005

```
p = &y;
```

```
*p = 20;
```

variável	p		x		y	
valor	1004		7		20	
endereço	1000	1001	1002	1003	1004	1005

variável	p		x		y	
valor	1004		7		20	
endereço	1000	1001	1002	1003	1004	1005

expressão	valor
x	
&x	
y	
&y	
p	
&p	
*p	

variável	p		x		y	
valor	1004		7		20	
endereço	1000	1001	1002	1003	1004	1005

expressão	valor
x	7
&x	1002
y	20
&y	1004
p	1004
&p	1000
*p	20

p = &x	atribuição
p = p + 1	incrementa o valor do apontador em 1 * sizeof(tipo)
p = p - 10	Decrementa o valor do apontador em 10 * sizeof(tipo)
*p	Valor existente na posição cujo endereço está armazenado em p.
&p	Endereço de memória de um apontador/variável.

```
char p[] = "hello world";
```

'h'	'e'	'l'	'l'	'o'	' '	'w'	'o'	'r'	'l'	'd'	'\0'
1000	1001	1002	1003	1004	1005	1006	1007	1008	1009	1010	1011
p	p+1	p+2	p+3	p+4	p+5	p+6	p+7	p+8	p+9	p+10	p+11

```
int p[] = {42, 87, 314159};
```

```
// na verdade um int ocupa 4 posições de memoria
```

42				87				314158			
1000	1001	1002	1003	1004	1005	1006	1007	1008	1009	1010	1011
p				p+1				p+2			

scanf

Agora já percebem o porquê de usar o `&` no `scanf`.
Nós damos ao `scanf` um endereço onde guardar os dados introduzidos pelo utilizador.

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

vetores e apontadores

Quando se declara um vector `int v[3]`, `v` é um apontador de inteiros (`int *v`) e aponta para a primeira posição de memória que foi reservada para armazenar 3 inteiros.

```
int v[3] = {42, 24, 87};  
// assume-se que v começa no  
// endereço de memória 1000;  
// cada inteiro ocupa 4 bytes  
// em memória
```

expressão	valor
<code>*v</code>	42
<code>v[0]</code>	42
<code>*(v+1)</code>	24
<code>v[1]</code>	24
<code>&v[1]</code>	1004
<code>v</code>	1000

scanf

Agora já percebem o porquê de não usar o & no `scanf` quando recebemos uma string.

Nós damos ao `scanf` um endereço onde guardar os dados introduzidos pelo utilizador, mas um vetor já é um apontador para uma zona de memória.

```
scanf ("%s", palavra);
```

apontadores de apontadores

variável	p		x		pp	
valor						
endereço	1000	1001	1002	1003	1004	1005

```
int x = 5;  
int *p = &x;  
int **pp = &p;
```

x	
p	
*p	
pp	
*pp	
**pp	

apontadores de apontadores

variável	p		x		pp	
valor	1002		5		1000	
endereço	1000	1001	1002	1003	1004	1005

```
int x = 5;  
int *p = &x;  
int **pp = &p;
```

x	5
p	1004
*p	5
pp	1000
*pp	1002
**pp	5

apontadores de apontadores

um vector é um apontador para uma zona de memória
com um conjunto de elementos seguidos

um vector bidimensional é um
apontador de apontadores para uma zona de ...

um vector tridimensional é um apontador
de apontadores de apontadores para uma zona de ...,

...

Argumentos por **valor** vs **referência**

```
int soma(int x, int y){  
    return x+y;  
}
```

x, y passados por **valor**;
função recebe um cópia dos
dados que fica armazenada
numa posição de memória
diferente da original (fora
da função).

Argumentos por **valor** vs **referência**

```
int soma(int x, int y){  
    return x+y;  
}
```

```
int somaVec(int *v, int n){  
    int i, total;  
    for(i=0, total=0; i<n; i++)  
        total += v[i];  
    return total;  
}
```

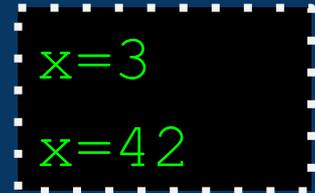
x, y, n passados por **valor**;
função recebe um cópia dos dados que fica armazenada numa posição de memória diferente da original (fora da função).

v passado por **referência**;
função recebe cópia do endereço de memória memória original dos dados;
função pode alterar dados originais porque está a aceder à zona de memória original.

Agora já podemos escrever uma função que altere o valor de uma variável local de outro bloco de código, e.g. outra função.

```
void altera(int * p, int val){  
    // instrucoes  
}
```

```
int main(void){  
    int x = 3;  
    printf("x=%d\n", x);  
    altera(&x, 42);  
    printf("x=%d\n", x);  
}
```



x=3
x=42

Escreva a função altera.