

Linguagem C

Apontadores


Endereços e apontadores

Passagem de parâmetros para funções


Apontadores e vectores


Memória dinâmica

Endereços de Memória

 Podemos considerar a memória como um vector M , em que cada byte é acedido por $M[i]$
 i é o endereço do byte acedido

 Os endereços são números inteiros sem sinal

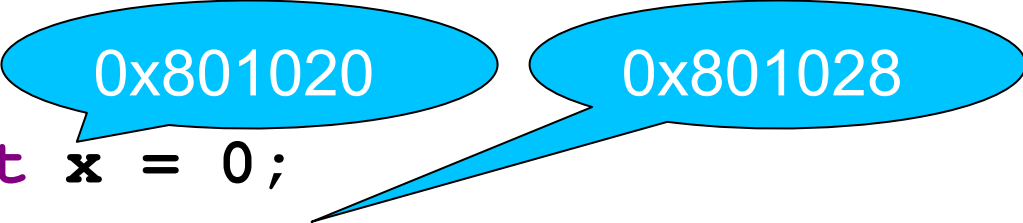
 Todos os dados (e código) são guardados na memória — logo têm um endereço

 Em C podemos obter o endereço de uma variável usando o operador `&`

`int x; // &x -> corresponde ao endereço de x!`

Apontadores

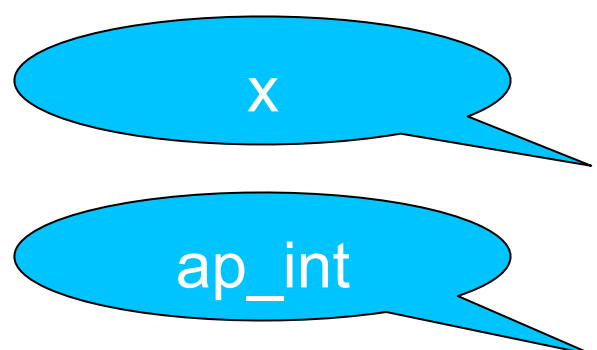
- 🖥️ Em C podemos ter variáveis que guardam endereços (em vez de dados)
- 🖥️ Essas variáveis chamam-se **apontadores**
- 🖥️ Os apontadores apontam para tipos de dados específicos



```
int x = 0;
int *ap_int;    //apontador para inteiros
                // * significa apontador
ap_int = &x;    //ap_int tem o endereço de x
// diz-se que ap_int aponta para x!
```

Apontadores — memória

 Na memória:




Endereço	Valor
0x80101C	??
0x801020	0
0x801024	??
0x801028	0x801020
0x80102C	??

Desreferenciação

 Através de um apontador podemos aceder ao valor da variável apontada

✓ A isto se chama **desreferenciar** o apontador

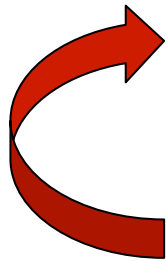
 É usado o operador ***** para desreferenciar o apontador

```
int x;  
int * ap_int;  
ap_int = &x;  
*ap_int = -1;    //desreferenciação de ap_int  
                  //o mesmo que fazer x = -1!  
// *ap_int passa a ser um aliás de x  
// uma nova designação para x!
```

Desreferenciação (2)

 Na memória:

Desreferenciar



Variável	Endereço	Valor
	0x80101C	??
x	0x801020	-1
	0x801024	??
ap_int	0x801028	0x801020
	0x80102C	??

Apontadores — propriedades

 Os apontadores são variáveis como as outras!

 Podemos atribuir valores entre apontadores

```
char c, *ap1, *ap2;
```

```
ap1 = &c;
```




```
ap2 = ap1; // ap2 tem o endereço de c
```

```
           // *ap1 e *ap2 são aliases de c!
```

 Os valores dos apontadores são **endereços de memória**





- ✓ São números sem sinal!
- ✓ O seu tamanho corresponde ao número de bits para endereçar a memória — em sistemas de 32 bits temos `sizeof(ap1) == 4`

O apontador NULL

-  A constante **NULL** (definida em `stdlib.h`) serve para indicar que o apontador não aponta para nenhum endereço válido (semelhante ao Java)
-  O valor de **NULL** corresponde ao inteiro 0
-  Desreferenciar um apontador que vale **NULL** é um erro
 - ✓ A consequência é o programa terminar com um erro de acesso à memória (*segmentation fault* ou *bus error*)

```
if( ap_int != NULL )  
    x = *ap_int; //podemos desreferenciar
```


Passagem de parâmetros — escalares

-  Em C os parâmetros escalares são sempre passados por valor!
-  Os valores são copiados para a função no momento da chamada
-  Se quisermos ter parâmetros de “saída” temos que utilizar apontadores!
-  Simulamos passagem de parâmetros por referência!

```
void incrementar( int *x )  
{  
    (*x)++;  
}  
  
int i = 0;  
incrementar(&i); // o mesmo que i++
```

Exemplo – Troca (errado)

```
void troca( int x, int y )
{
    int tmp = x;
    x = y;
    y = tmp;
}

...

int a = 1, b = 2;
troca( a, b );
printf( "a=%d b=%d\n", a, b );
```

Exemplo – Troca (errado)

```
void troca( int x, int y )  
{  
    int tmp = x;  
    x = y;  
    y = tmp;  
}
```

...

```
int a = 1, b = 2;  
troca( a, b );  
printf( "a=%d b=%d\n", a, b );
```

output: a=1 b=2

Exemplo – Troca (certo)

```
void troca( int *x, int *y )
{
    // solução – usar apontadores
    int tmp = *x;
    *x = *y;
    *y = tmp;
}

...

int a = 1, b = 2;
troca( &a, &b );
printf( "a=%d b=%d\n", a, b );
```

Exemplo – Troca (certo)




```
void troca( int *x, int *y )  
{  
    // solução – usar apontadores  
    int tmp = *x;  
    *x = *y;  
    *y = tmp;  
}
```

...


```
int a = 1, b = 2;  
troca( &a, &b );  
printf( "a=%d b=%d\n", a, b );
```

output: a=2 b=1

Apontadores e vectores

-  Em C um vector é um apontador constante para o seu primeiro elemento
-  Podemos considerar que qualquer apontador aponta para uma sequência de valores do mesmo tipo
 - ✓ Por isso, o apontador pode ser usado da mesma forma que o vector para aceder aos elementos seguintes
-  O vector é um apontador constante – não pode ser alterado para apontar para outra posição de memória

Apontadores e vectores — semelhanças

 Para muitas utilizações as seguintes declarações têm o mesmo efeito:

 `char *s1 = "xpto";`


- ✓ s1 é um apontador, inicializado com o endereço da string “xpto”, que ocupa 5 bytes

 `char s2[] = "xpto";`




- ✓ s2 é um vector de 5 bytes, inicializado para conter a string “xpto”

```
printf( "%s\n%s\n", s1, s2 );
```

```
printf( "%c\n%c\n", s1[1], s2[1] );
```

 Repare que aqui não há diferença no uso de s1 e s2!

Aritmética de apontadores

-  Podemos somar inteiros a apontadores
-  Somar n a um apontador corresponde a avançar n elementos no vector correspondente
 - ✓ $p[0]$ é o mesmo que $*p$
 - ✓ $p[3]$ é o mesmo que $*(p+3)$
 - ✓ Incrementar o apontador corresponde a fazê-lo apontar para a próxima posição do vector
 - ✓ Corresponde a somar `sizeof` do tipo apontado ao valor numérico do apontador (endereço)
 - ✓ Pode ser utilizado para percorrer vectores
-  Também podemos:
 - ✓ Subtrair inteiros a apontadores
 - ✓ Fazer a diferença (e comparação) entre apontadores

Aritmética de apontadores — exemplo

```
int v[20];
int *p;

v[16] = 1;           // 16 = 0x10
v[17] = 2;           // 17 = 0x11
p = v;
//%p — mostrar valor do apontador (endereço)
printf("%p %d\n", p, p-v);
p += 16;              0x7fff5fbff9c0 0
printf("%p %d %d\n", p, p-v, *p);
p++;                  0x7fff5fbffa00 16 1
printf("%p %d %d\n", p, p-v, *p);
                        0x7fff5fbffa04 17 2
```

Somar os elementos de um vector

 Somar todos os elementos de um vector

```
#define MAX_VEC 5

int v[MAX_VEC], *p, s, i;

p = v;
s = 0;
for( i = 0; i < MAX_VEC; i++ )
{
    s += *p;
    p++;
}
```

Somar os elementos de um vector

```
int v[MAX_VEC], *p, i;  
→ p = v;  
s = 0;  
for( i = 0; i < MAX_VEC; i++ )  
{  
    s += *p;  
    p++;  
}
```

5	4	1	2	5
---	---	---	---	---

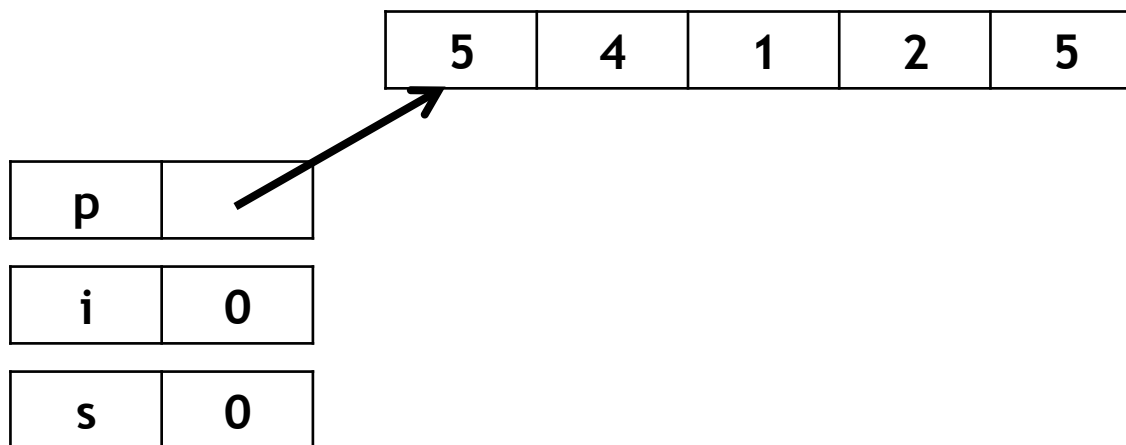
p	??
---	----

i	??
---	----

s	??
---	----

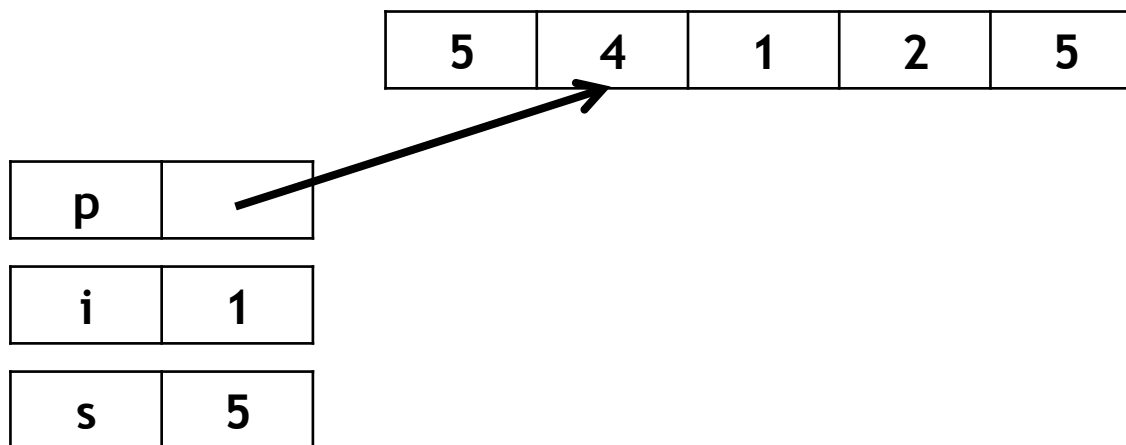
Somar os elementos de um vector

```
int v[MAX_VEC], *p, i;  
  
p = v;  
s = 0;  
→ for( i = 0; i < MAX_VEC; i++ )  
{  
    s += *p;  
    p++;  
}
```



Somar os elementos de um vector

```
int v[MAX_VEC], *p, i;  
  
p = v;  
s = 0;  
→ for( i = 0; i < MAX_VEC; i++ )  
{  
    s += *p;  
    p++;  
}
```



Somar os elementos de um vector

```
int v[MAX_VEC], *p, i;  
  
p = v;  
s = 0;  
→ for( i = 0; i < MAX_VEC; i++ )  
{  
    s += *p;  
    p++;  
}
```



p	
i	2
s	9



Somar os elementos de um vector

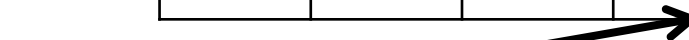
```
int v[MAX_VEC], *p, i;  
  
p = v;  
s = 0;  
→ for( i = 0; i < MAX_VEC; i++ )  
{  
    s += *p;  
    p++;  
}
```

5	4	1	2	5
---	---	---	---	---

p	
---	--

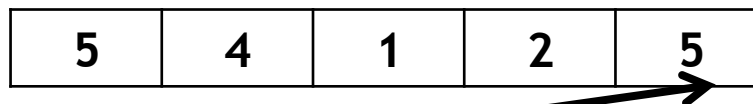
i	3
---	---

s	10
---	----



Somar os elementos de um vector


```
int v[MAX_VEC], *p, i;  
  
p = v;  
s = 0;  
→ for( i = 0; i < MAX_VEC; i++ )  
{  
    s += *p;  
    p++;  
}
```



p	
i	4
s	12

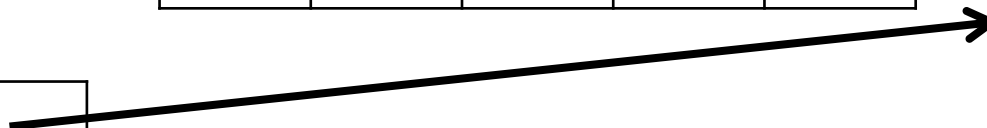
Somar os elementos de um vector

```
int v[MAX_VEC], *p, i;  
  
p = v;  
s = 0;  
for( i = 0; i < MAX_VEC; i++ )  
{  
    s += *p;  
    p++;  
}
```



5	4	1	2	5
---	---	---	---	---

p	
i	5
s	17



Pré-incremento e pós-incremento

 ***p++** incrementa **p**, mas devolve o valor apontado por **p** antes de ser incrementado


 Podemos usar para compactar o código anterior:

```
for( i = 0; i < MAX_VEC; i++ )  
    s += *p++; // o mesmo que s += *(p++);
```

 Também existe o pré-incremento

 **++p** incrementa **p**, e devolve o valor apontado por **p** após este ser incrementado

Passagem de parâmetros — vectores

 Quando uma função tem um parâmetro que é um vector — **esse parâmetro corresponde a um apontador!**

```
void f( char s[] )
```

 É exactamente o mesmo que:

```
void f( char *s )
```


 Ou seja — **s é um apontador** que vai apontar para o **vector original** com o qual foi chamada a função!

 Daí que os vectores em C sejam passados por referência!


Passagem de vectores — exemplo

```
void f( int v[] )  
{ ... }
```

```
int vec[100];  
f( vec );
```

 Quando `f` é chamada, o **apontador** `v` é inicializado para apontar para o primeiro elemento do vector `vec`

 Ao alterarmos os elementos de `v`, em `f`, estamos a alterar os elementos de `vec`!

 Repare que, em `f`, `sizeof(v)` é o tamanho da variável apontador `v`, não do vector `vec`!

Somar os elementos de um vector

 Determinar o tamanho de uma string

```
int strlen1 (char v[])  
{ int i, len=0;  
  for (i=0; v[i]!='\0'; i++)  
    len++;  
  return len;  
}
```


Notação de
vectores!

Notação de
apontadores!

```
int strlen2 (char *v)  
{ int len=0;  
  while (*v++ != '\0')  
    len++;  
  return len;  
}
```

Memória dinâmica – malloc

 É possível reservar memória para o programa sempre que necessário

 A função **malloc** reserva um bloco de memória dinâmica de n bytes e retorna um apontador para ele

 Retorna NULL se não houver memória suficiente

// alocar espaço

// para um vector de 100 inteiros

```
int *v = malloc(100*sizeof(int));
```


// alocar espaço para a copia duma string s

```
char *new_s = malloc(strlen(s)+1);
```

// alocar espaço para uma estrutura s

```
struct s *ps = malloc(sizeof(struct s));
```

calloc, realloc e free

 **calloc** reserva espaço para um vector de n posições e inicializa tudo a zeros

```
int *v = calloc(100, sizeof(int));
```


 **realloc** altera o tamanho de um bloco de memória dinâmica

✓ O bloco pode ter ser movido na memória, pelo que realloc retorna o seu novo endereço

```
v = realloc(v, 200 * sizeof(int));
```

 **free** liberta um bloco de memória dinâmica

```
free(v); // liberta memória alocada acima
```

 Protótipos destas funções estão em **stdlib.h**

Free — avisos

 A função **free** liberta blocos de memória dinâmica

- ✓ Só serve para memória alocada por malloc (e calloc)
- nunca para vectores estáticos!

 free só pode ser usado **uma vez** para cada bloco

- ✓ Fazer free duas vezes do mesmo bloco tem consequências imprevisíveis

 Depois de fazer free é boa política por o apontador a NULL

- ✓ assim se evitam erros não detectados de acessos a posições de memória inválidas

```
free(v); v = NULL;
```