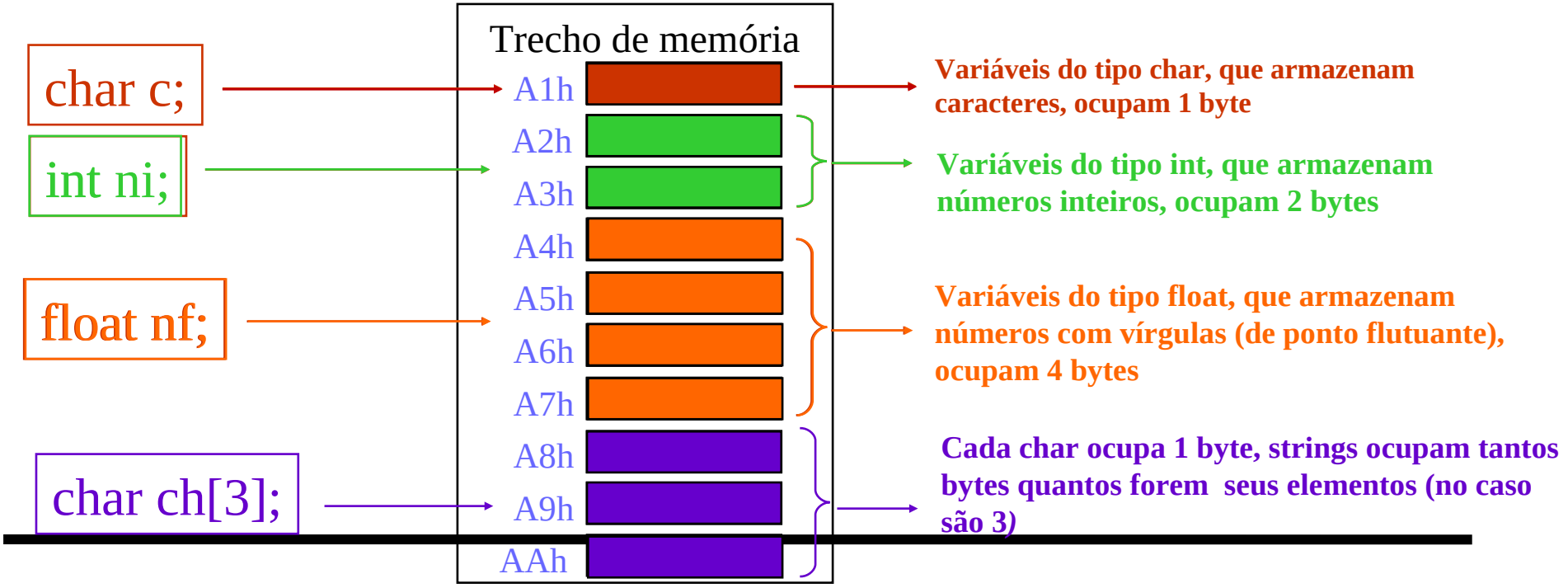
Variáveis x memória

- Quando uma variável é declarada, o compilador reserva uma região de memória para ela
  - a reserva depende do tamanho do tipo de dado da variável (que depende da arquitetura)



Ponteiro

- variável que armazena o endereço de uma variável de um determinado tipo
- declaração: **`<tipo> *<identificador>;`**
  - tipo: pode ser qualquer tipo válido em C

```
int *nota;
char *texto;
```

```
float *peso;
double *media;
```

- ponteiro tem 2 informações:
  - armazena um endereço
  - endereço aponta para um valor

Inicialização de ponteiros

- Após a declaração e antes da inicialização → valor do ponteiro é desconhecido
- Para um ponteiro não apontar para um local inválido (lixo), deve ser inicializado ou receber o valor NULL

```
int x = 10;
int *ptr;
ptr = &x;
```

Alternativa: `ptr = NULL;`

Ponteiros: operadores & e *

```
int x, q, *ptr;
x = 100;
ptr = &x;   // ptr recebe o end. de x ("aponta para")
q = *ptr;   // q recebe o conteúdo do end. de ptr
```

Operador &

→ obtém o endereço de memória de uma variável

Se a variável **x** está armazenada na posição de memória 2000, o valor de **ptr** será 2000

Operador *

→ obtém o conteúdo da variável apontada

→ usado p/ declarar um ponteiro

Se a variável **ptr** aponta p/ um endereço que contém o valor 100, o valor de **q** será 100

Ponteiros: exemplo

```c
#include <stdio.h>

int main(){
  int i = 9, *p, *q;

  p = &i;  // p aponta para i
  q = p;   // q também aponta para i

  printf("i = %ld bytes\t *p = %ld bytes\t *q = %ld bytes\n", sizeof(i), sizeof(p), sizeof(q));

  printf(" i = %d \n",i);     // valor de i
  printf("&i = %p\n\n", &i); // end de i
  printf(" p = %p \n", p);    // end apontado
  printf("*p = %d\n", *p);    // cont. do end apontado por p
  printf("&p = %p\n\n", &p); // end de p
  printf(" q = %p \n", q);    // end apontado
  printf("*q = %d\n", *q);    // cont. do end apontado por q
  printf("&q = %p\n", &q);    // end de q

  return 0;
}
```

```
i = 4 bytes      *p = 8 bytes    *q = 8 bytes

 i = 9
&i = 0x7fff72d60354

 p = 0x7fff72d60354
*p = 9
&p = 0x7fff72d60358

 q = 0x7fff72d60354
*q = 9
&q = 0x7fff72d60360
```

Aritmética de Ponteiros

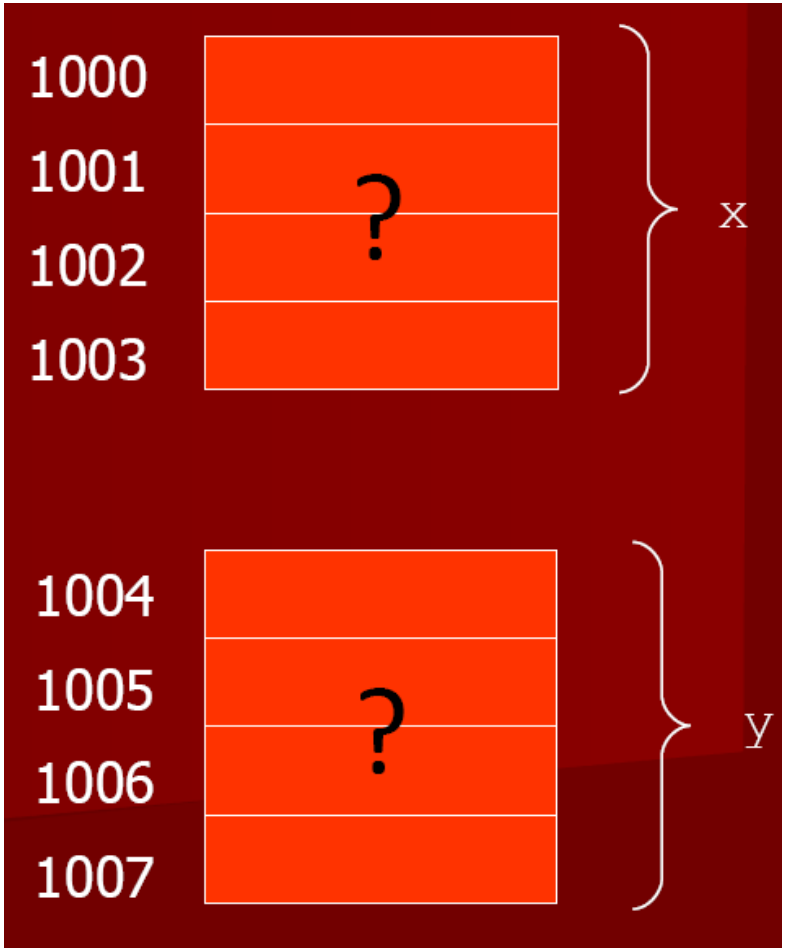
- O valor do ponteiro aumenta/diminui dependendo do nro de bytes do tipo base
  - ao ser incrementado, o ponteiro passa a apontar p/ a posição de memória do próximo elemento do seu tipo base
  - ao ser decrementado, o ponteiro passa a apontar p/ a posição de memória do elemento anterior do seu tipo base
  - Se ptr é um ponteiro p/ inteiro com valor inicial 2000 (fictício)
    - em uma arquitetura com inteiros de 4 bytes (p/ saber o tamanho de um tipo: `sizeof(tipo);`)

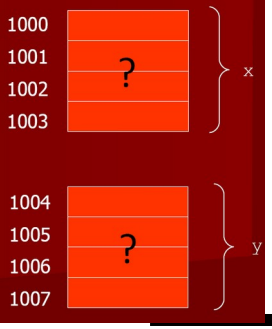
```
ptr++;          // ptr passa a ser 2004

ptr = ptr + 3;   // ptr passa a apontar p/ 2012: 3º
                 // elemento do tipo ptr, adiante do atual
```

Aritmética de Ponteiros (1a)

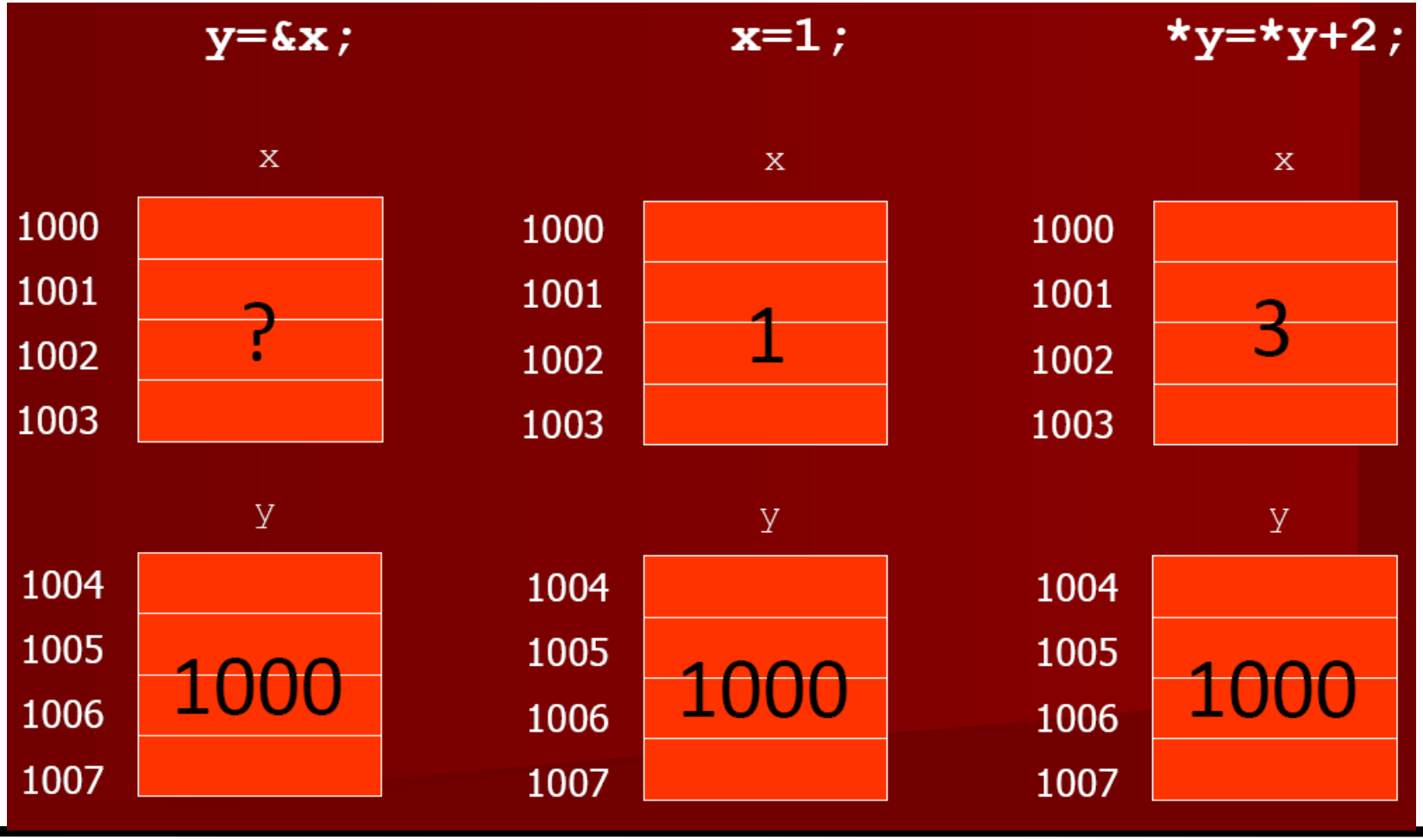
```
int x,*y;
```

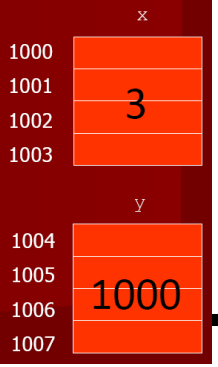
- “x” e “y” ocupam 4 bytes consecutivos cada
- “x” (“y”) inicia no endereço 1000 (1004) e termina no endereço 1003 (1007)
- O endereço de “x” (“y”) é 1000 (1004)
- Os valores iniciais de “x” e de “y” são indeterminados
- “x” é do tipo inteiro e “y” é do tipo ponteiro para inteiro



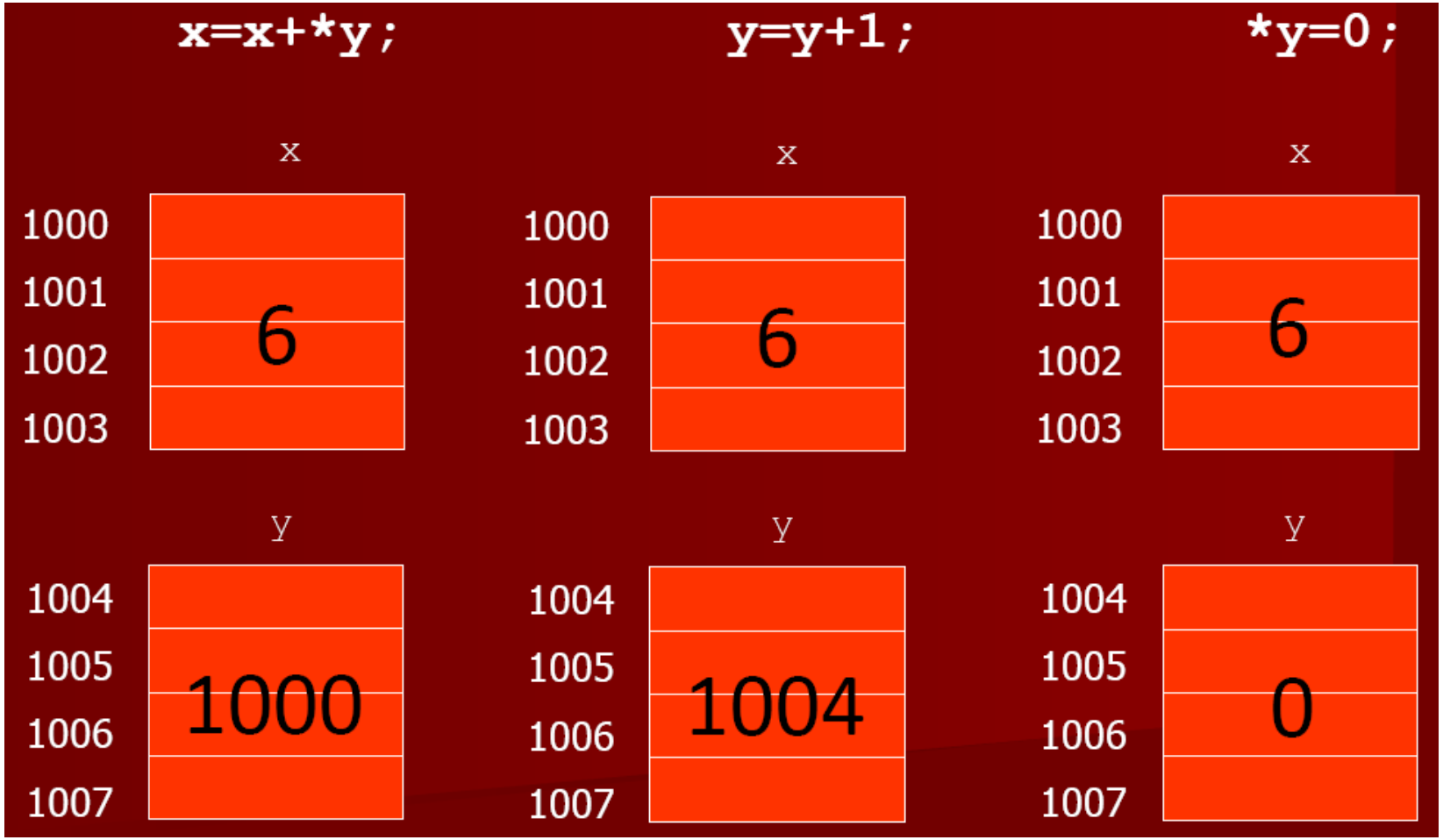


Aritmética de Ponteiros (1b)





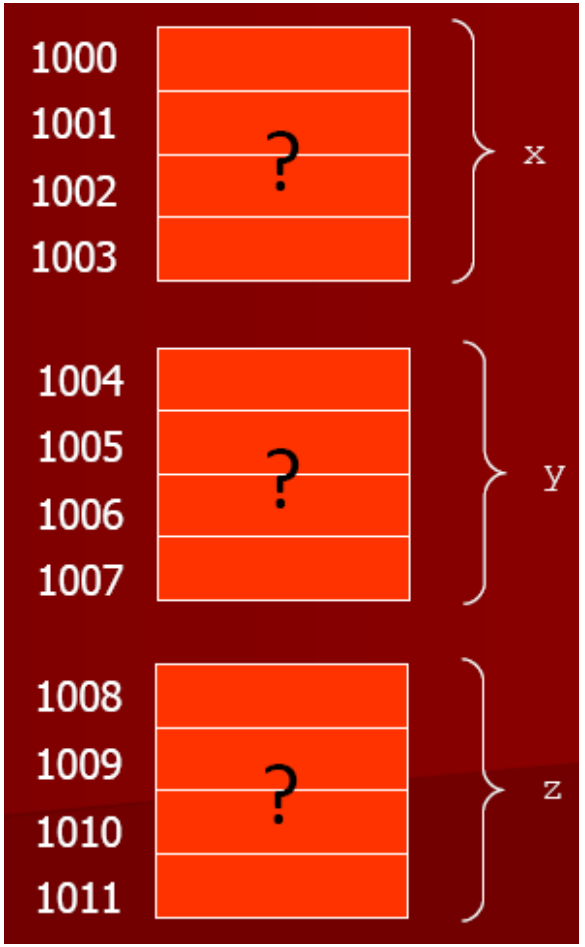
Aritmética de Ponteiros (1c)

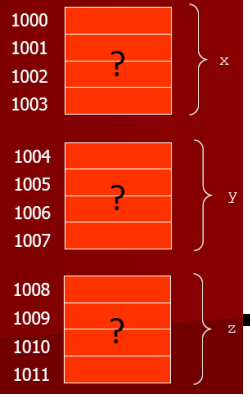


Aritmética de Ponteiros (2a)

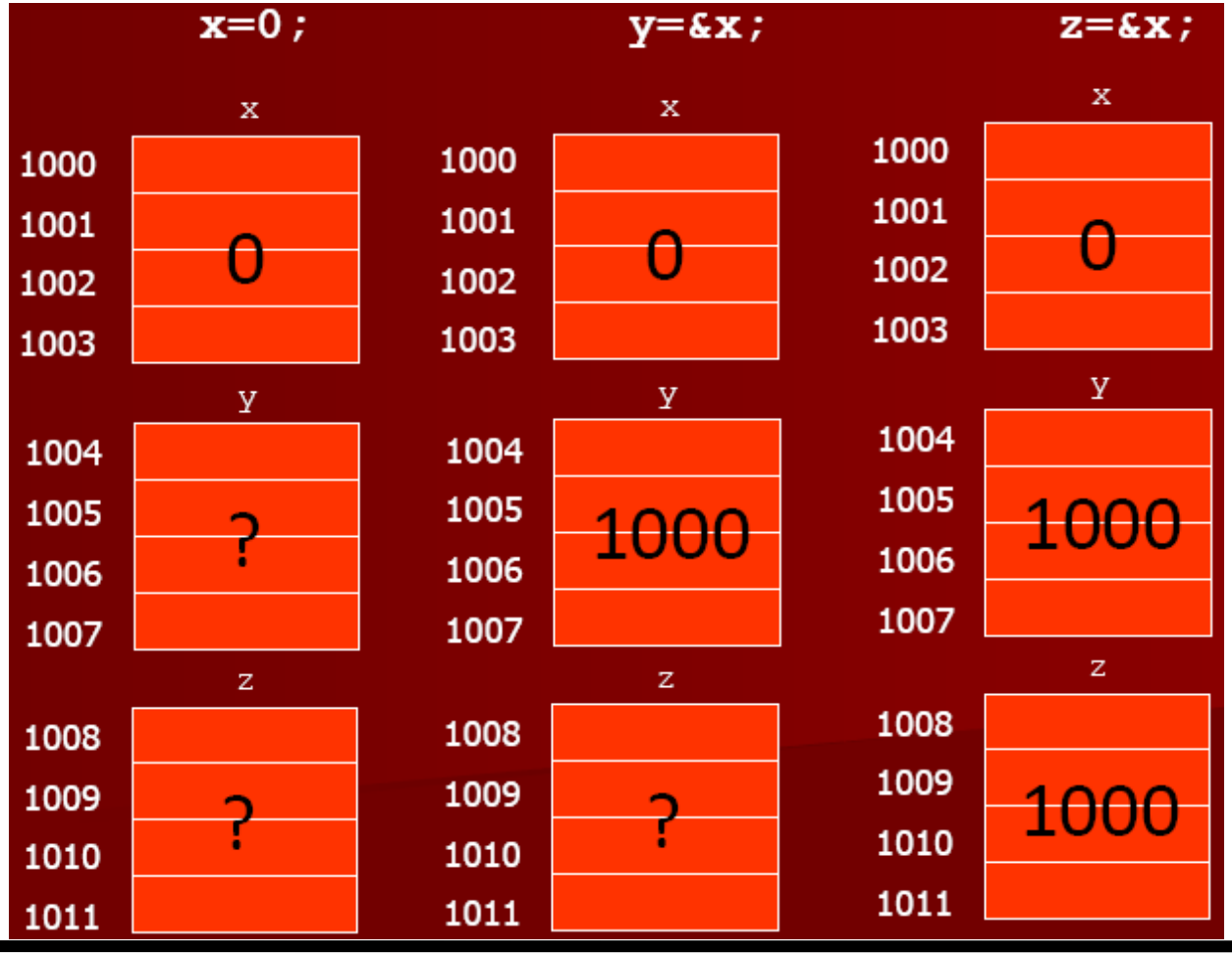
```
int x,*y,*z;
```

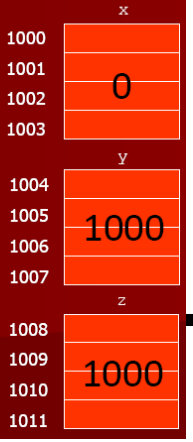
- “x” é do tipo inteiro
- “y” é do tipo ponteiro para inteiro
- “z” é do tipo ponteiro para inteiro



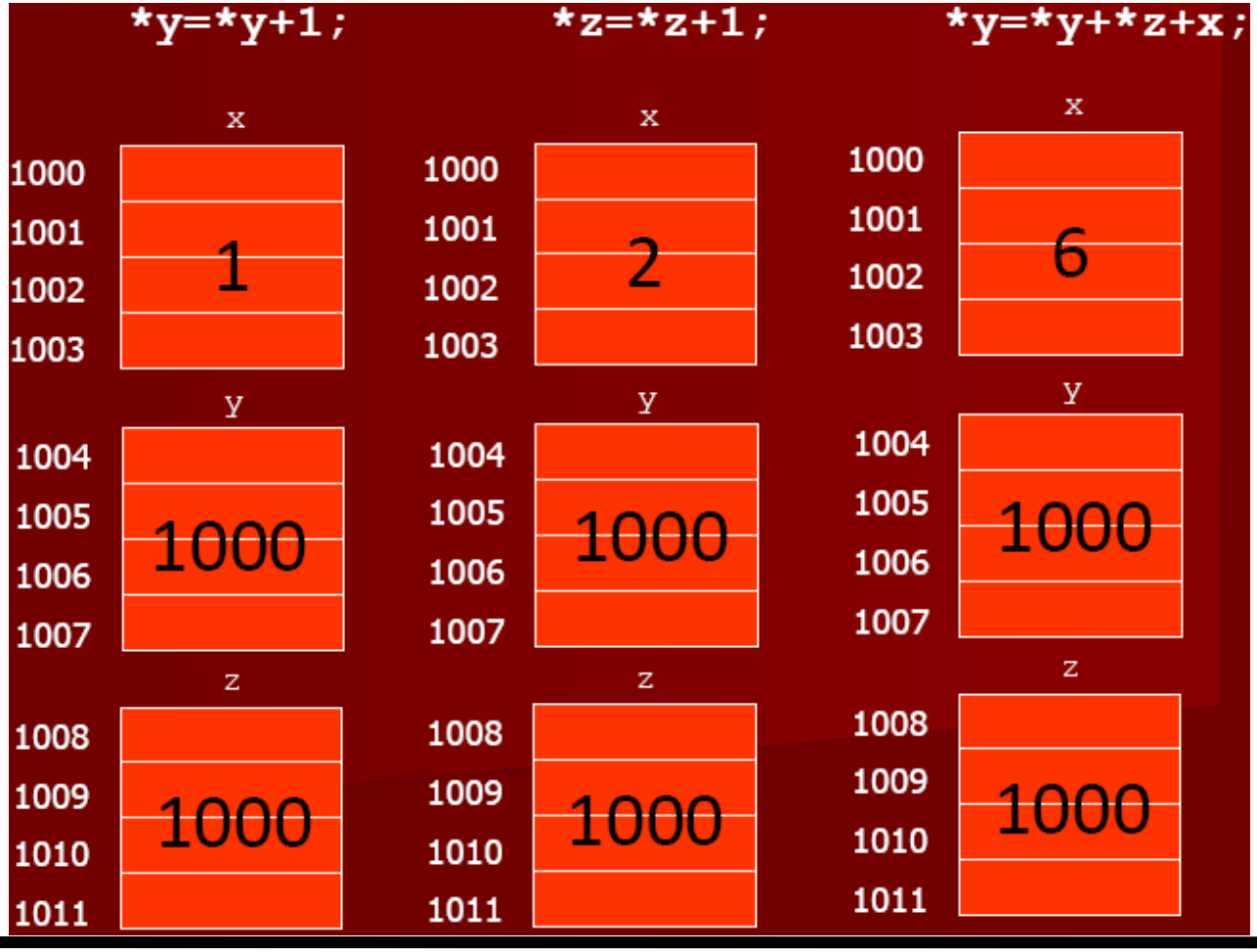


Aritmética de Ponteiros (2b)





Aritmética de Ponteiros (2c)



Vetores e ponteiros: ex. na memória

```
int x[4],*p;
```

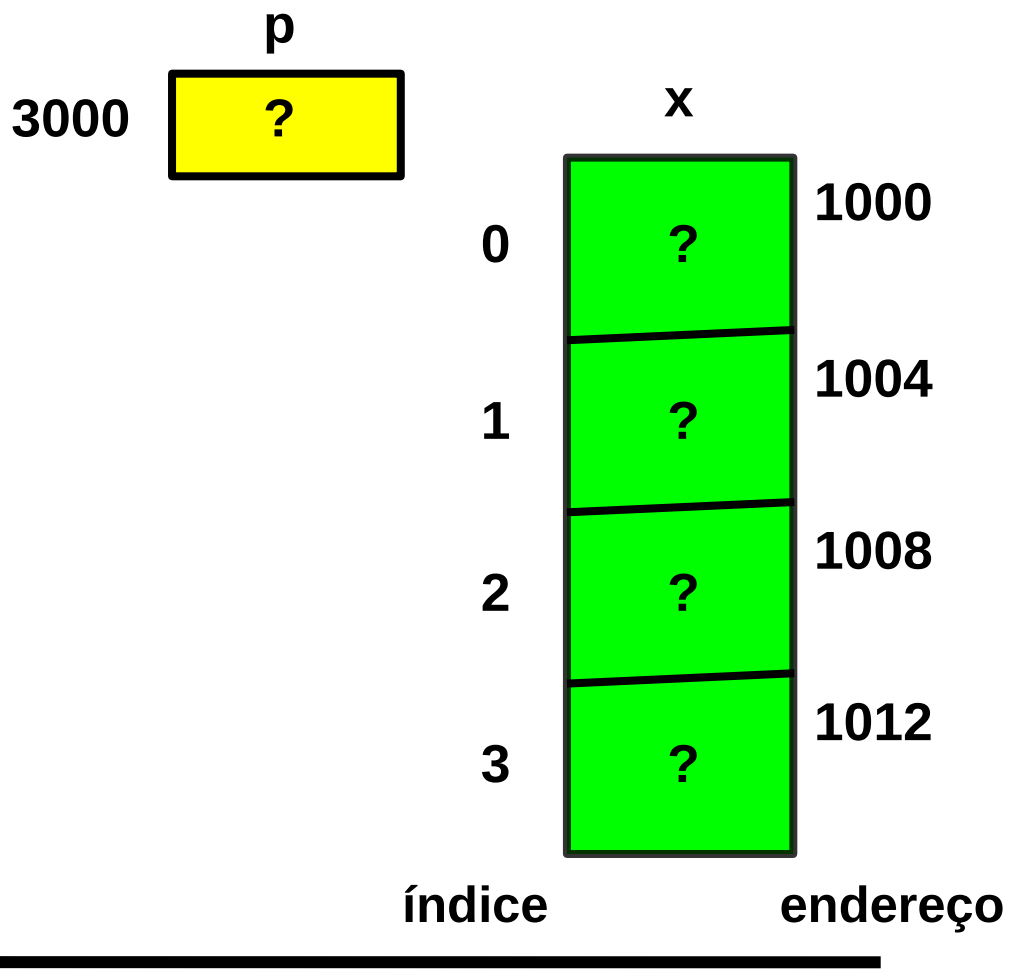
“p” é um ponteiro para inteiro

Como “x” é um vetor de inteiros, então “p” pode receber o valor armazenado em “x”

Logo, “p” aponta para o mesmo vetor que “x”

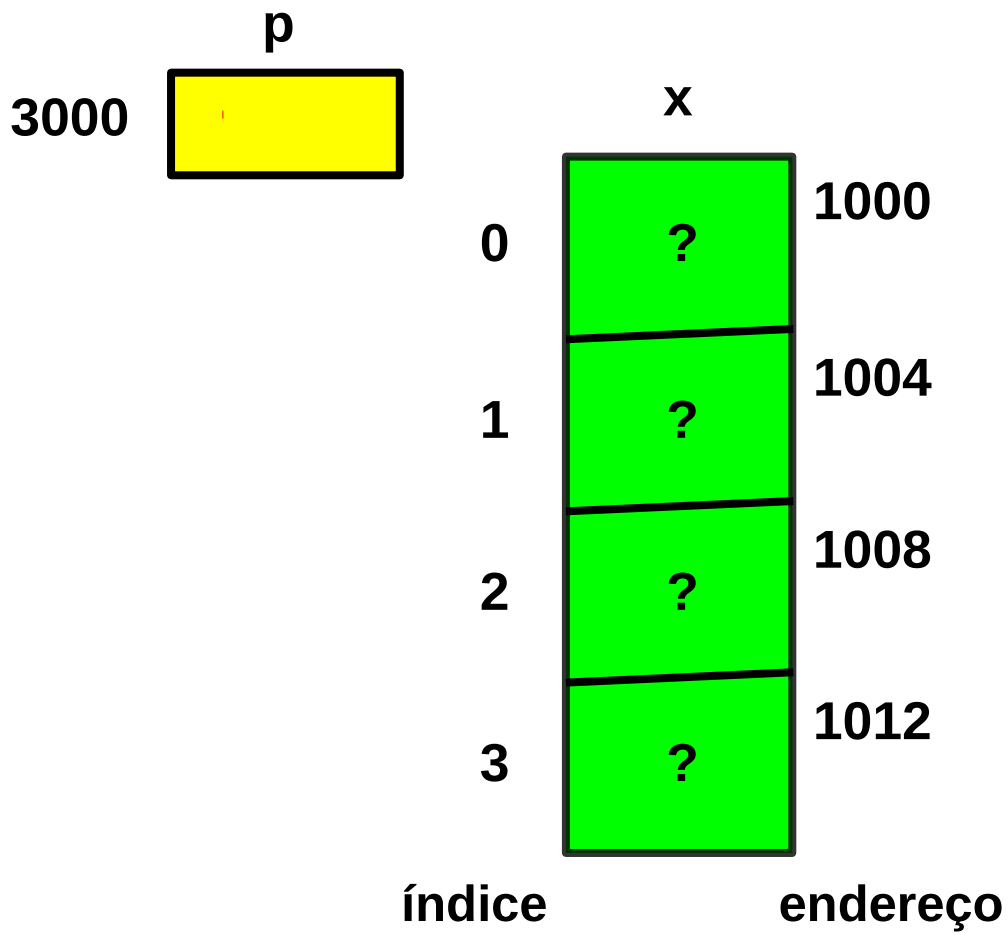
O vetor pode ser acessado através do ponteiro também

x[i] é o mesmo que *(p+i)



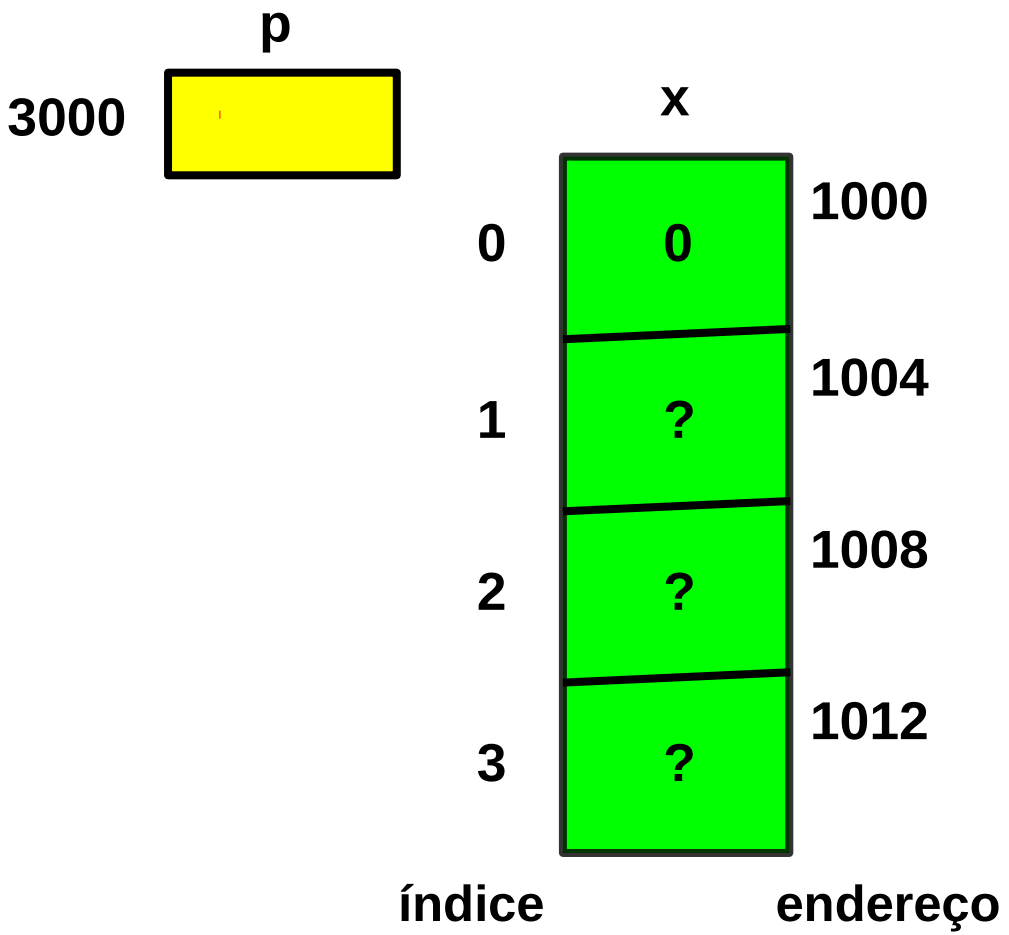
Vetores e ponteiros

```
int x[4],*p;
p=x;
```



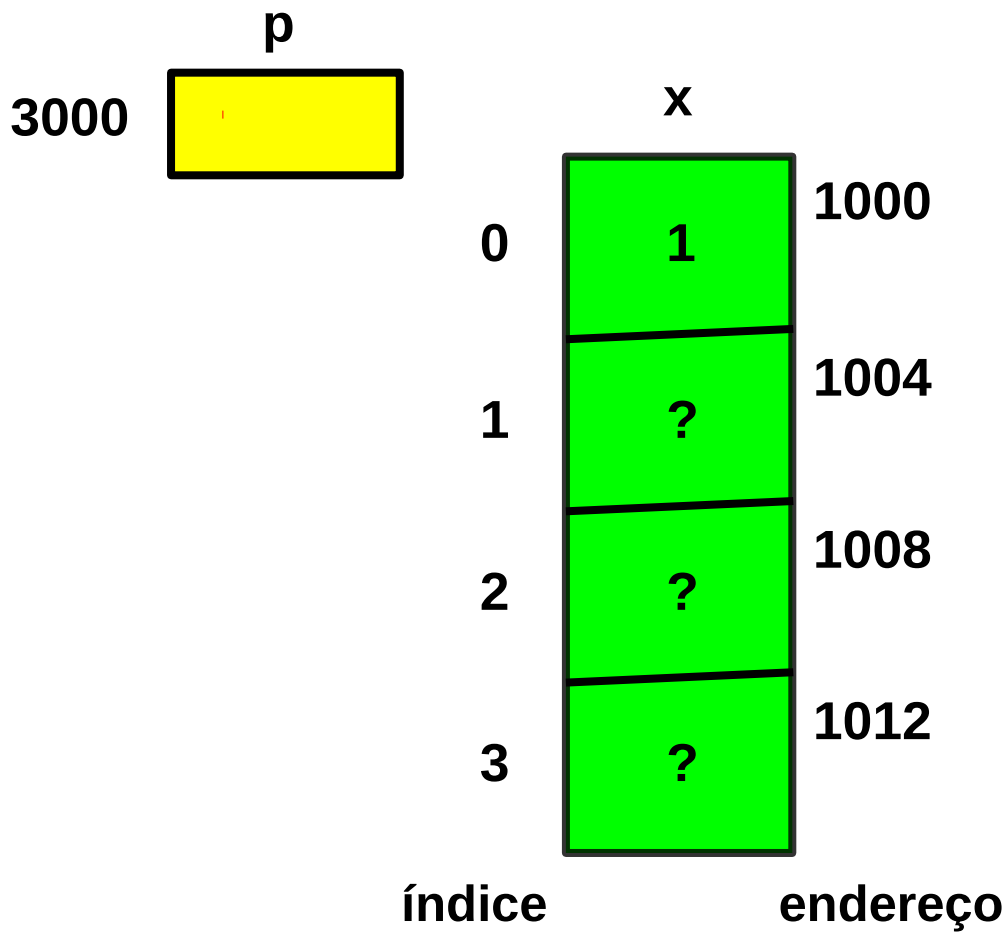
Vetores e ponteiros

```
int x[4],*p;
p=x;
*p=0;
```



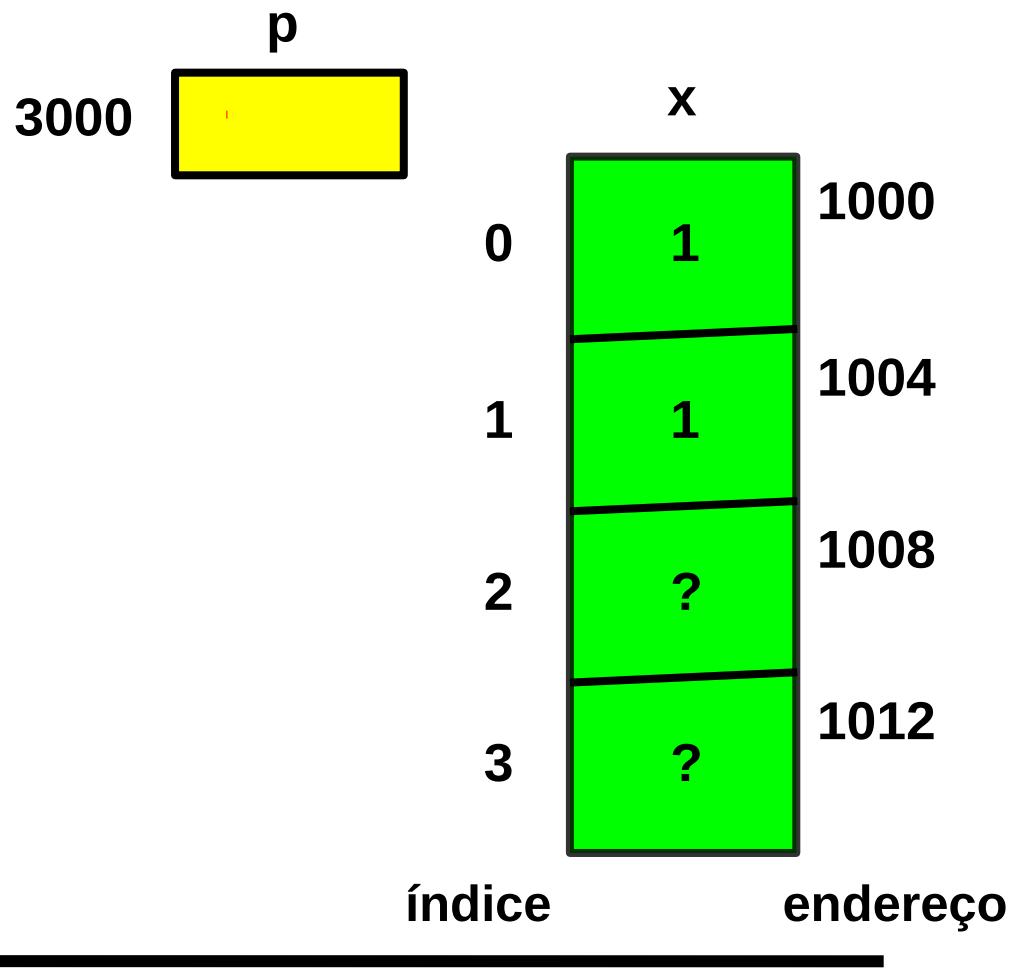
Vetores e ponteiros

```
int x[4],*p;
p=x;
*p=0;
*p=*p+1;
```



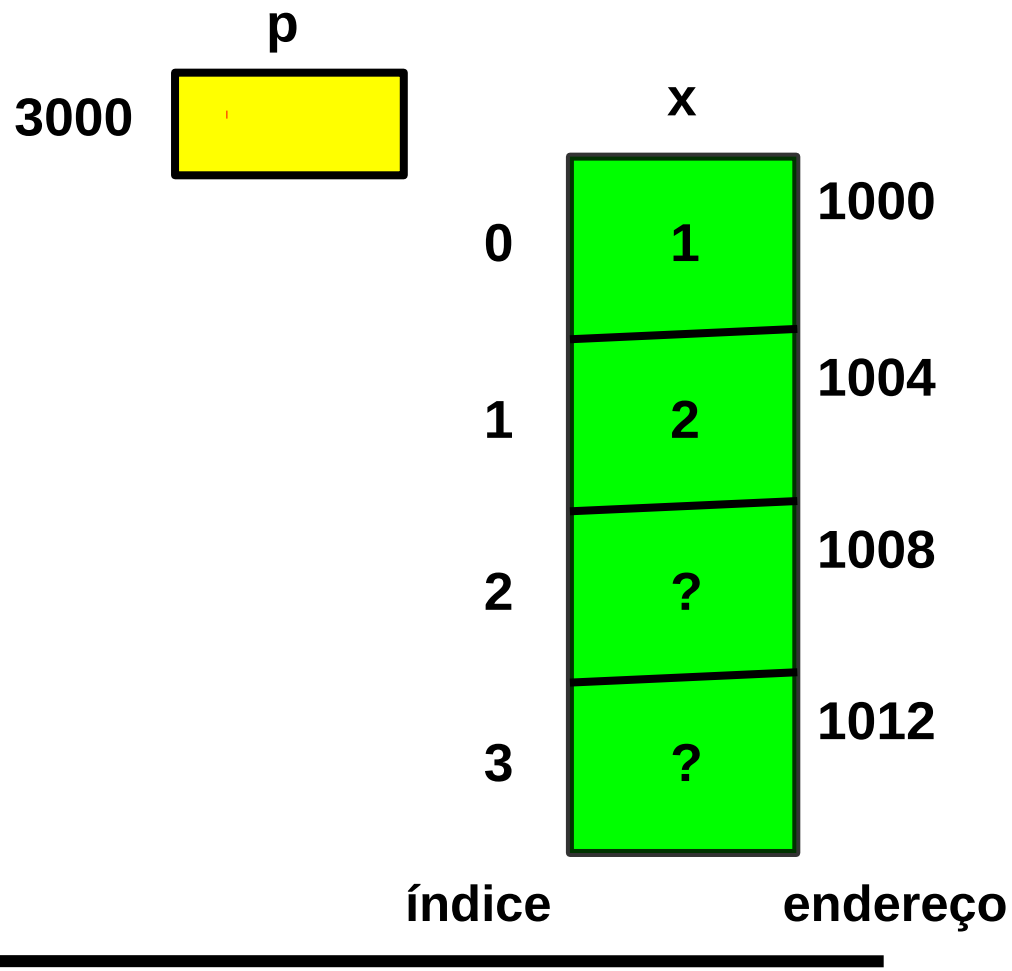
Vetores e ponteiros

```
int x[4],*p;
p=x;
*p=0;
*p=*p+1;
*(p+1)=1;
```



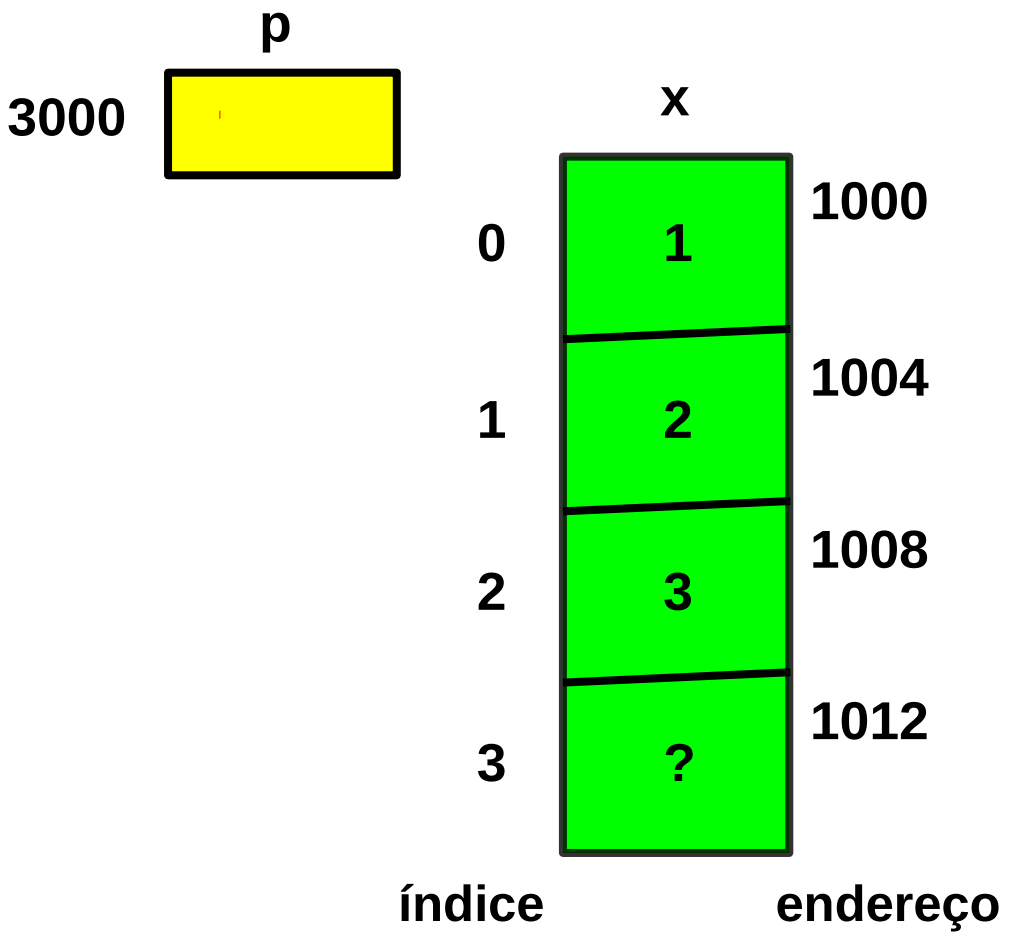
Vetores e ponteiros

```
int x[4],*p;
p=x;
*p=0;
*p=*p+1;
*(p+1)=1;
*(p+1)=*(p+1)+1;
```



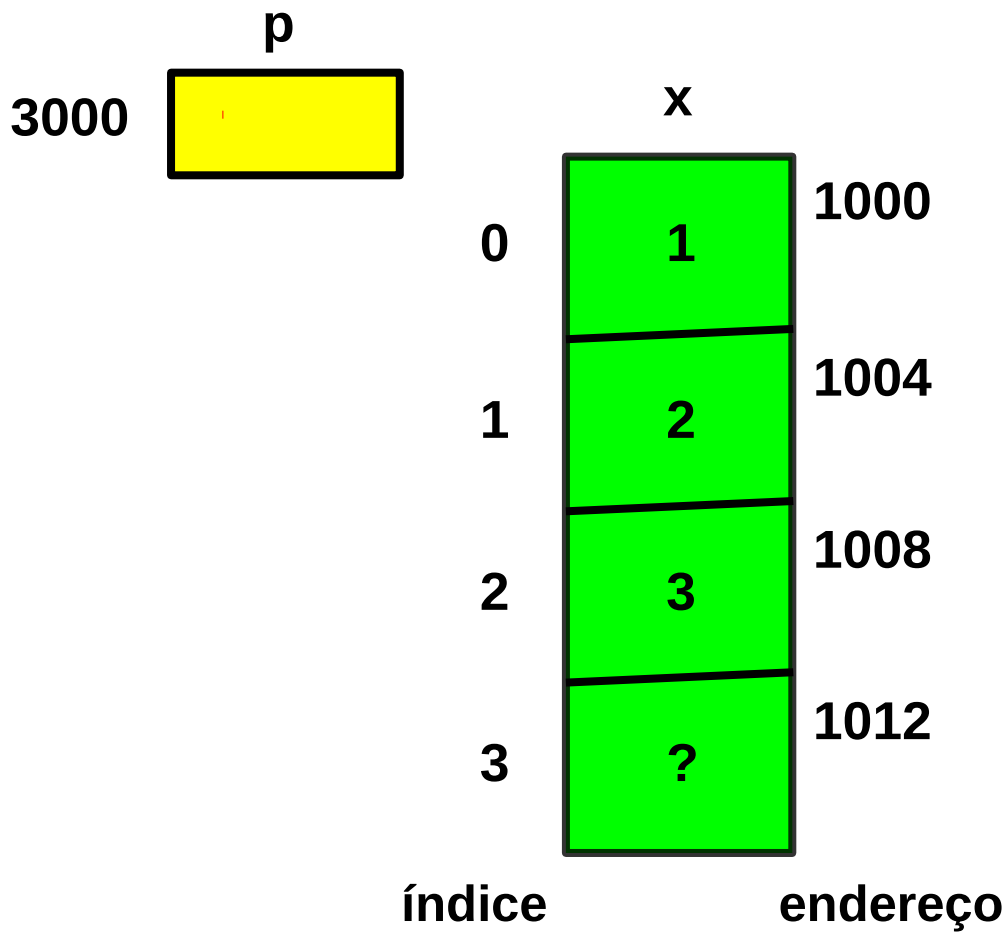
Vetores e ponteiros

```
int x[4],*p;
p=x;
*p=0;
*p=*p+1;
*(p+1)=1;
*(p+1)=*(p+1)+1;
*(p+2)=*(p+1)+1;
```



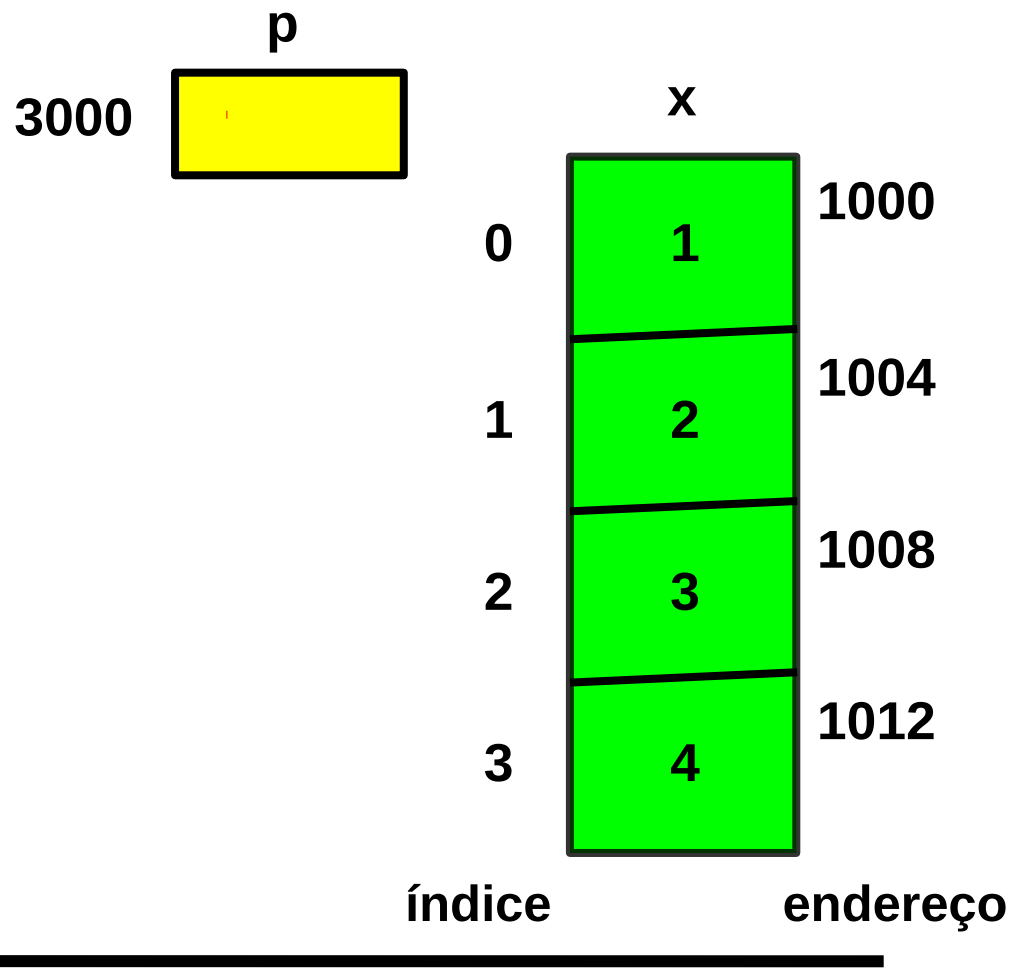
Vetores e ponteiros

```
int x[4],*p;
p=x;
*p=0;
*p=*p+1;
*(p+1)=1;
*(p+1)=*(p+1)+1;
*(p+2)=*(p+1)+1;
p=p+3;
```



Vetores e ponteiros

```
int x[4],*p;
p=x;
*p=0;
*p=*p+1;
*(p+1)=1;
*(p+1)=*(p+1)+1;
*(p+2)=*(p+1)+1;
p=p+3;
*p=4;
```



Vetores e ponteiros: equivalências

	vetor	ponteiro
Declaração	`int v[5];`	`int *v`
Manipulação (conteúdo)	`v[i]`	`*(v+i)`
Manipulação (endereço)	`&v[i]`	`v+i`

Acesso aos elementos de um vetor com ponteiro

```
int v[5]={5, 10, 15, 20, 25};
int *p = v;
```

Formas de acesso ao elemento de valor 15:

`v[2]` – 3ª posição do vetor v (índice 2)

`p[2]` – 3ª posição do vetor apontado por p

`*(p+2)` – 2º elem. adiante de p (p aponta p/ 1º elem.)

`*(v+2)` – idem anterior, pois "p = v = &v[0]"

Strings e ponteiros

```c
#include <stdio.h>

void minhaStrcpy(char *destino, char *origem){

  while(*origem){
     *destino = *origem;
     origem++;
     destino++;
  }
  *destino = '\0';
}

int main(){
  char s1[30], s2[30];

  printf("digite uma string: ");
  fgets(s1, sizeof(s1), stdin);
  minhaStrcpy(s2, s1);
  puts(s1);
  puts(s2);

  return 0;
}
```

```c
void minhaStrcpy(char *destino, char *origem){

  while(*origem)
     *(destino++) = *(origem++);

  *destino = '\0';
}
```

```
digite uma string: programa
programa

programa
```