

# Princípios de Desenvolvimento de Algoritmos **MAC122**

Prof. Dr. Paulo Miranda  
**IME-USP**

Alocação Dinâmica  
de Memória

# Alocação Dinâmica de Memória

- Motivação:**

- Nossos programas pré-fixavam o número de variáveis a serem utilizadas.

- No caso de vetores e matrizes o tamanho era fixado como sendo um limitante superior previsto (**constante**).

- Desperdício de memória.**

- Variáveis locais são armazenadas em uma parte da memória chamada pilha.

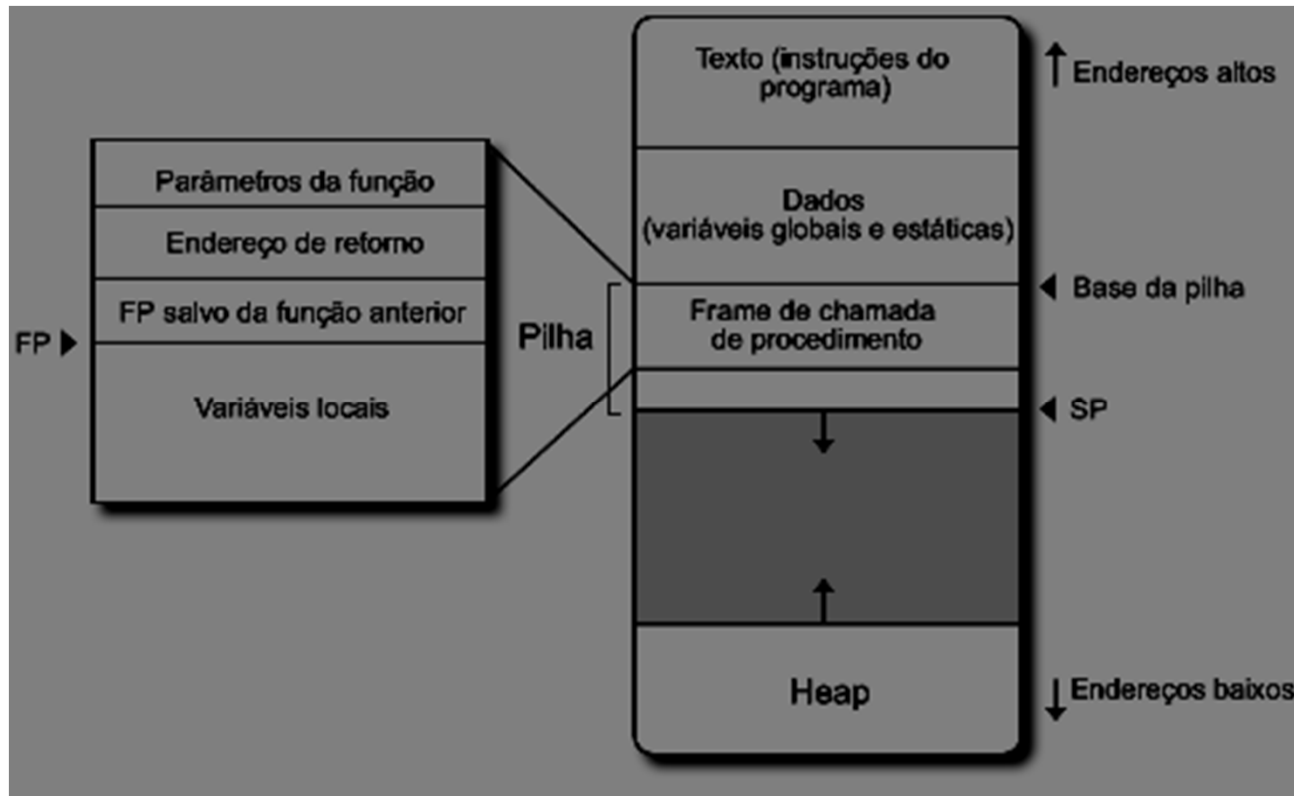
# Alocação Dinâmica de Memória

## •Motivação:

- Existe uma parte da memória para alocação dinâmica chamada heap que contém toda memória disponível, não reservada para outras finalidades.
- C permite alocar (**reservar**) espaço de memória de tamanho arbitrário no heap em tempo de execução tornando os programas mais flexíveis.
- O acesso a memória alocada é realizada por meio de ponteiros.

# Alocação Dinâmica de Memória

- A área de heap cresce em sentido oposto à pilha e em direção a esta.



# Alocação Dinâmica de Memória

- Funções da stdlib.h:

- `malloc` ou `calloc` para **alocar** memória no heap.

- `free` para **desalocar** (liberar) memória previamente alocada com `malloc` ou `calloc`.

- `realloc` para **alterar** o tamanho de um bloco de memória alocado, preservando o conteúdo já existente.

# Alocação Dinâmica de Memória

## •Função `malloc`:

```
void *malloc(unsigned int nbytes);
```

–Aloca um bloco de memória no heap com tamanho em bytes dado pelo argumento `nbytes`.

–A função retorna o endereço do primeiro byte do bloco de memória recém alocado ou o valor nulo `NULL` em caso de falta de memória.

–O endereço retornado é do tipo `void *`, o que simboliza um endereço genérico sem um tipo específico.

–Esse endereço deve ser armazenado em um ponteiro através de uma conversão (**cast**) do tipo do endereço genérico para o tipo do ponteiro particular.

# Alocação Dinâmica de Memória

- Função `calloc`:

```
void *calloc(unsigned int nmemb,  
             unsigned int size);
```

- Aloca um bloco de memória no heap com tamanho em bytes dado por **`nmemb*size`**.

- A função retorna o endereço do primeiro byte do bloco de memória recém alocado.

- Ela também inicializa todo o conteúdo do bloco com zero.

# Alocação Dinâmica de Memória

- **Exemplo:** Alocação de variável anônima (**sem nome**)

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

int main(){
    int *p;

    p = (int *)malloc(4);
    *p = 12;
    printf("%d\n", *p);
    free(p);
    system("pause");
    return 0;
}
```

- O programa inicia a execução na função principal (**main**).



# Alocação Dinâmica de Memória

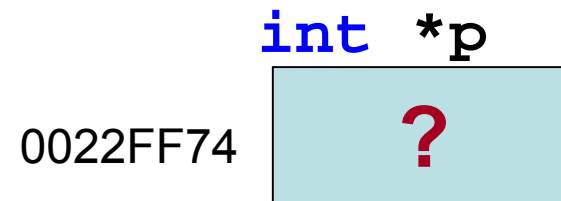
- **Exemplo:** Alocação de variável anônima (**sem nome**)

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

int main(){
    ← int *p;

    p = (int *)malloc(4);
    *p = 12;
    printf("%d\n", *p);
    free(p);
    system("pause");
    return 0;
}
```

- É alocado espaço para as variáveis locais da função principal (**main**).



# Alocação Dinâmica de Memória

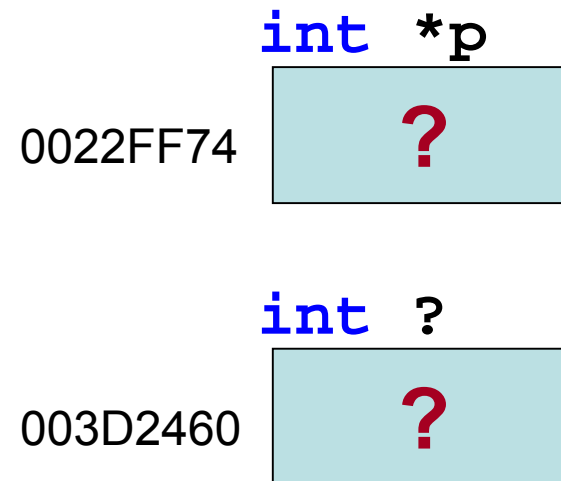
## •Exemplo: Alocação de variável anônima (sem nome)

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

int main(){
    int *p;

    ← p = (int *)malloc(4);
      *p = 12;
      printf("%d\n", *p);
      free(p);
      system("pause");
      return 0;
}
```

- O comando **malloc** aloca 4 bytes de memória de forma dinâmica, exatamente o tamanho de uma variável do tipo **int**.



# Alocação Dinâmica de Memória

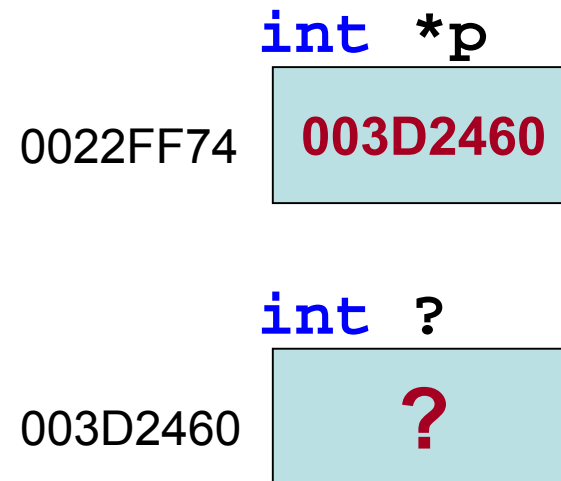
## •Exemplo: Alocação de variável anônima (**sem nome**)

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

int main(){
    int *p;

    ← p = (int *)malloc(4);
      *p = 12;
      printf("%d\n", *p);
      free(p);
      system("pause");
      return 0;
}
```

•A variável recém criada não possui nome (**anônima**). O seu endereço é retornado pela função **malloc** e atribuído ao ponteiro.



# Alocação Dinâmica de Memória

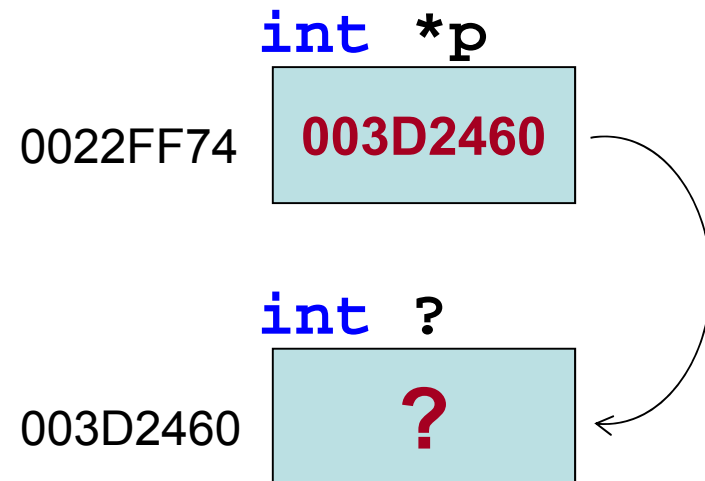
- **Exemplo:** Alocação de variável anônima (**sem nome**)

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

int main(){
    int *p;

    ← p = (int *)malloc(4);
      *p = 12;
      printf("%d\n", *p);
      free(p);
      system("pause");
      return 0;
}
```

- Dizemos que p aponta para a variável anônima (**graficamente representado por uma seta**).



# Alocação Dinâmica de Memória

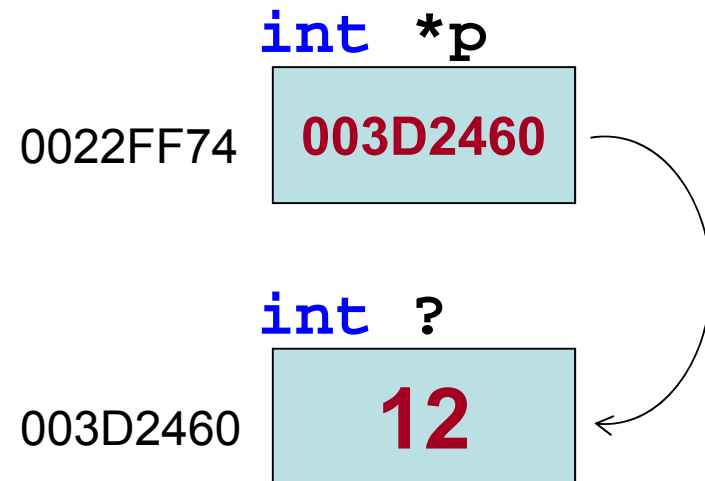
## •Exemplo: Alocação de variável anônima (**sem nome**)

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

int main(){
    int *p;

    p = (int *)malloc(4);
    ← → *p = 12;
    printf("%d\n", *p);
    free(p);
    system("pause");
    return 0;
}
```

•A partir do ponteiro é possível alterar o conteúdo da variável anônima apontada.



# Alocação Dinâmica de Memória

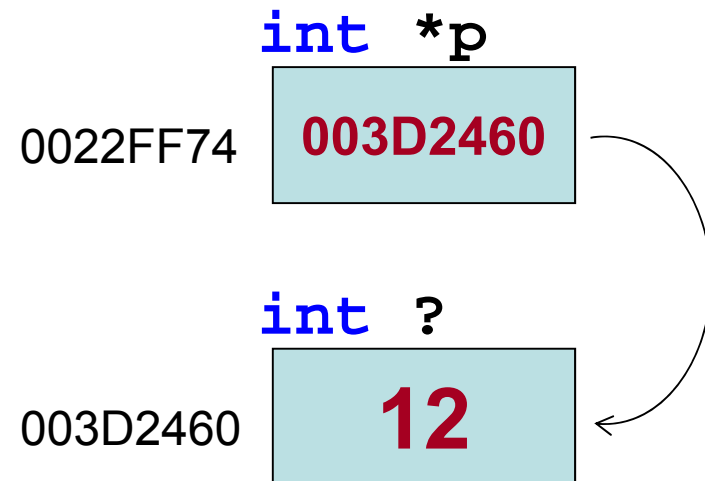
- **Exemplo:** Alocação de variável anônima (**sem nome**)

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

int main(){
    int *p;

    p = (int *)malloc(4);
    *p = 12;
    printf("%d\n", *p);
    free(p);
    system("pause");
    return 0;
}
```

- O conteúdo **12** da variável anônima é impresso na saída padrão.



# Alocação Dinâmica de Memória

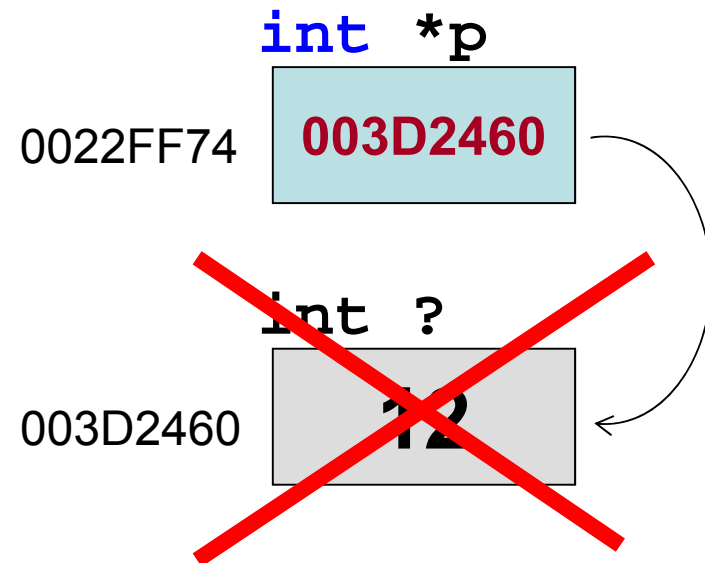
- **Exemplo:** Alocação de variável anônima (**sem nome**)

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

int main(){
    int *p;

    p = (int *)malloc(4);
    *p = 12;
    printf("%d\n", *p);
    free(p);
    system("pause");
    return 0;
}
```

• A memória da variável anônima é desalocada ficando disponível para futuras alocações.



# Alocação Dinâmica de Memória

## •Alocação dinâmica de uma variável anônima:

–Para alocar variáveis de outros tipos (ex: `float`, `char`) basta mudar o tipo do ponteiro, o **cast** de conversão e o número de bytes a serem alocados.

–Para obter o tamanho em bytes de qualquer tipo da linguagem basta chamar o operador **`sizeof()`**.

```
tipo *p;  
p = (tipo *)malloc(sizeof(tipo));
```

–Ex:

```
float *p;  
p = (float *)malloc(sizeof(float));
```

```
struct Aluno *p;  
p = (struct Aluno *)malloc(sizeof(struct Aluno));
```



# Alocação Dinâmica de Memória

- **Alocação dinâmica de vetores:**

- **Antes de falarmos sobre alocação dinâmica de vetores, vamos entender a sua organização em memória e a sua relação com ponteiros a partir de um exemplo.**

# Alocação Dinâmica de Memória

```
↔ #include <stdio.h>
   int main(){
       int A[4]={1,2,3,4};
       int *p;
       printf("&A[0]: %u\n",&A[0]);
       printf("&A[1]: %u\n",&A[1]);
       printf("&A[2]: %u\n",&A[2]);
       printf("&A[3]: %u\n",&A[3]);
       printf("A: %u\n",A);
       p = &A[0];
       printf("p: %u, *p: %d\n",p,*p);
       p = p + 2;
       printf("p: %u, *p: %d\n",p,*p);
       return 0;
   }
```

- O programa inicia a execução na função principal (**main**).

# Alocação Dinâmica de Memória

```
#include <stdio.h>
int main(){
↔ int A[4]={1,2,3,4};
  int *p;
  printf("&A[0]: %u\n",&A[0]);
  printf("&A[1]: %u\n",&A[1]);
  printf("&A[2]: %u\n",&A[2]);
  printf("&A[3]: %u\n",&A[3]);
  printf("A: %u\n",A);
  p = &A[0];
  printf("p: %u, *p: %d\n",p,*p);
  p = p + 2;
  printf("p: %u, *p: %d\n",p,*p);
  return 0;
}
```

- É declarado um vetor com **4** elementos inteiros.
- Abaixo dos elementos são mostrados os seus respectivos endereços de memória.
- Normalmente os endereços são mostrados em base hexadecimal, porém para facilitar o entendimento eles estão em base decimal.

**int** A[**4**]

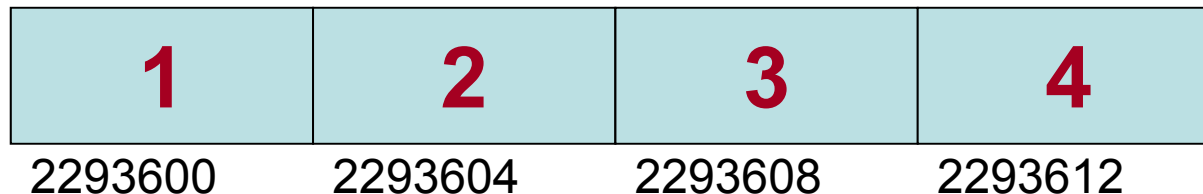
<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>
2293600	2293604	2293608	2293612

# Alocação Dinâmica de Memória

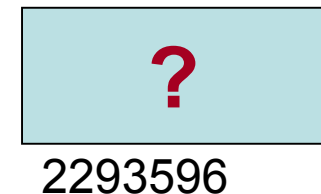
```
#include <stdio.h>
int main(){
    int A[4]={1,2,3,4};
    ⇔ int *p;
    printf("&A[0]: %u\n",&A[0]);
    printf("&A[1]: %u\n",&A[1]);
    printf("&A[2]: %u\n",&A[2]);
    printf("&A[3]: %u\n",&A[3]);
    printf("A: %u\n",A);
    p = &A[0];
    printf("p: %u, *p: %d\n",p,*p);
    p = p + 2;
    printf("p: %u, *p: %d\n",p,*p);
    return 0;
}
```

- É declarado um ponteiro **p** para um inteiro.

**int** A[4]



**int** \*p

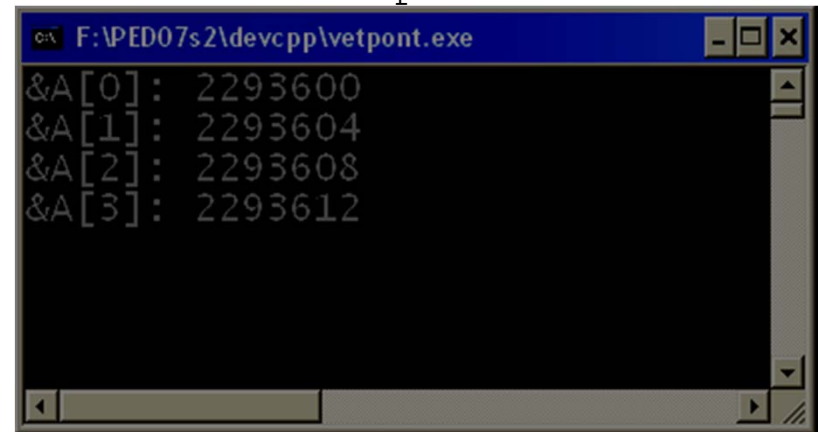


# Alocação Dinâmica de Memória

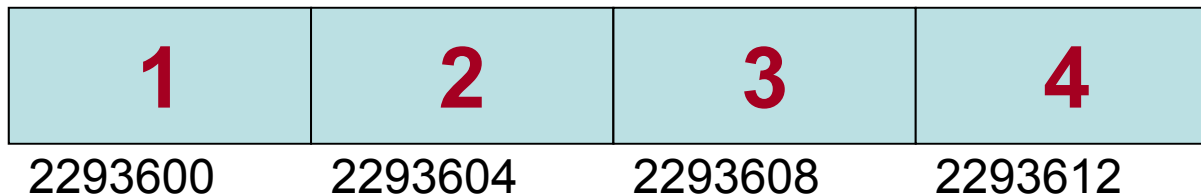
```
#include <stdio.h>
int main(){
    int A[4]={1,2,3,4};
    int *p;
    printf("&A[0]: %u\n",&A[0]);
    printf("&A[1]: %u\n",&A[1]);
    printf("&A[2]: %u\n",&A[2]);
    <=> printf("&A[3]: %u\n",&A[3]);
    printf("A: %u\n",A);
    p = &A[0];
    printf("p: %u, *p: %d\n",p,*p);
    p = p + 2;
    printf("p: %u, *p: %d\n",p,*p);
    return 0;
}
```

- São impressos os endereços de todos elementos do vetor com a formatação de decimal sem sinal (**%u**).

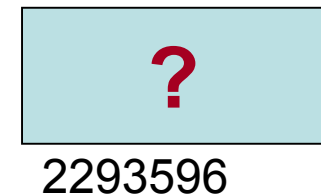
Saída padrão:



**int** A[4]



**int** \*p

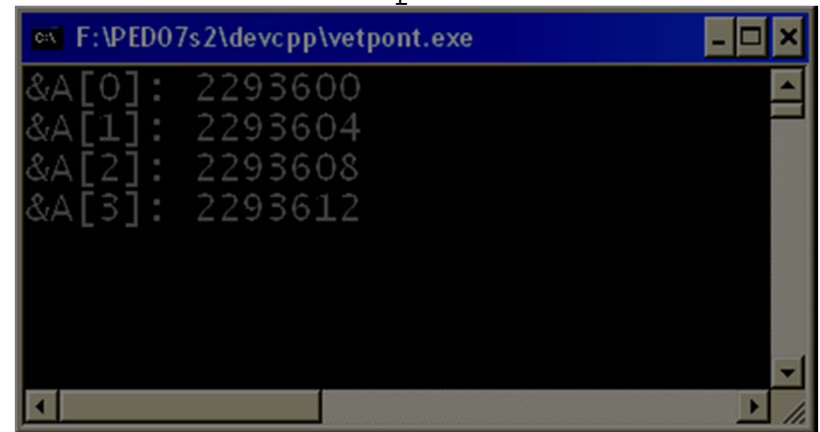


# Alocação Dinâmica de Memória

```
#include <stdio.h>
int main(){
    int A[4]={1,2,3,4};
    int *p;
    printf("&A[0]: %u\n",&A[0]);
    printf("&A[1]: %u\n",&A[1]);
    printf("&A[2]: %u\n",&A[2]);
    <=> printf("&A[3]: %u\n",&A[3]);
    printf("A: %u\n",A);
    p = &A[0];
    printf("p: %u, *p: %d\n",p,*p);
    p = p + 2;
    printf("p: %u, *p: %d\n",p,*p);
    return 0;
}
```

- Os elementos são inteiros de **4** bytes em posições consecutivas, logo os offsets dos endereços formam uma PA de razão **4**.

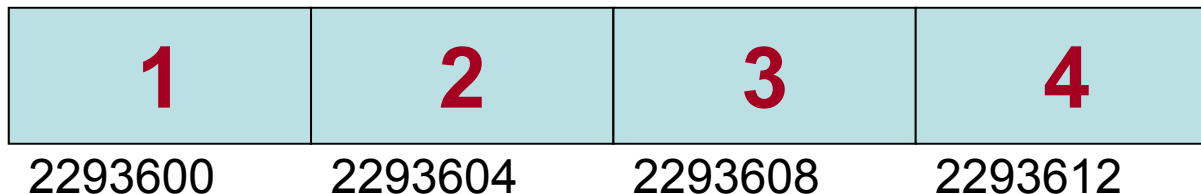
Saída padrão:



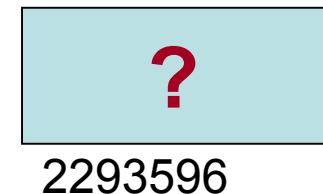
```

F:\PED07s2\devcpp\vetpont.exe
&A[0]: 2293600
&A[1]: 2293604
&A[2]: 2293608
&A[3]: 2293612
A: 2293600
p: 2293600, *p: 1
p: 2293608, *p: 3
```

**int** A[**4**]



**int** \*p



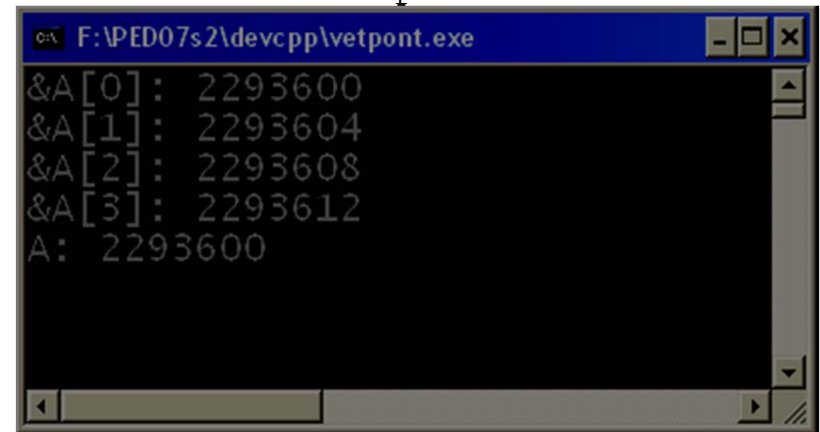
# Alocação Dinâmica de Memória

```
#include <stdio.h>
int main(){
    int A[4]={1,2,3,4};
    int *p;
    printf("&A[0]: %u\n",&A[0]);
    printf("&A[1]: %u\n",&A[1]);
    printf("&A[2]: %u\n",&A[2]);
    printf("&A[3]: %u\n",&A[3]);
    printf("A: %u\n",A);
    p = &A[0];
    printf("p: %u, *p: %d\n",p,*p);
    p = p + 2;
    printf("p: %u, *p: %d\n",p,*p);
    return 0;
}
```

• É impresso o valor do nome do vetor. Em C, o nome do vetor é o próprio endereço do primeiro elemento.

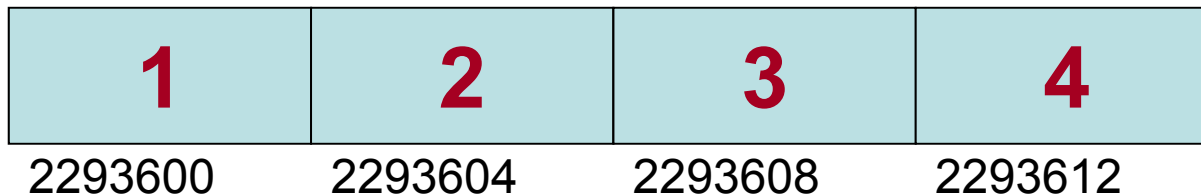
(A = &A[0] = 2293600)

Saída padrão:

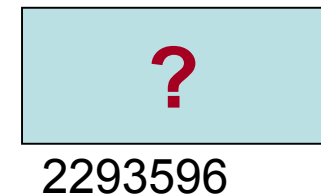


```
C:\ F:\PED07s2\devcpp\vetpont.exe
&A[0]: 2293600
&A[1]: 2293604
&A[2]: 2293608
&A[3]: 2293612
A: 2293600
```

int A[4]



int \*p

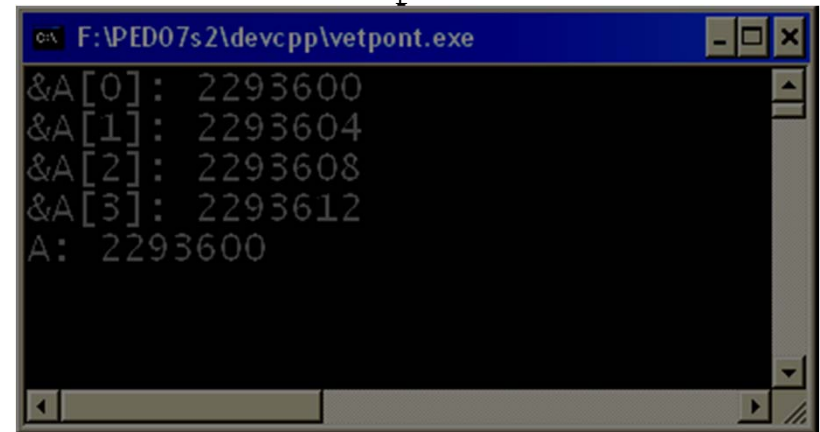


# Alocação Dinâmica de Memória

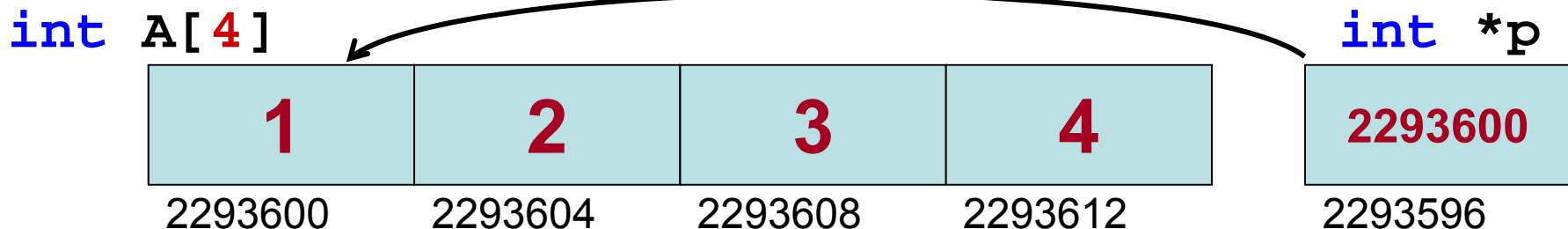
```
#include <stdio.h>
int main(){
    int A[4]={1,2,3,4};
    int *p;
    printf("&A[0]: %u\n",&A[0]);
    printf("&A[1]: %u\n",&A[1]);
    printf("&A[2]: %u\n",&A[2]);
    printf("&A[3]: %u\n",&A[3]);
    printf("A: %u\n",A);
    ←→p = &A[0];
    printf("p: %u, *p: %d\n",p,*p);
    p = p + 2;
    printf("p: %u, *p: %d\n",p,*p);
    return 0;
}
```

- O endereço do primeiro elemento é atribuído ao ponteiro p, que passa a apontar para o início do vetor.

Saída padrão:



```
&A[0]: 2293600
&A[1]: 2293604
&A[2]: 2293608
&A[3]: 2293612
A: 2293600
```



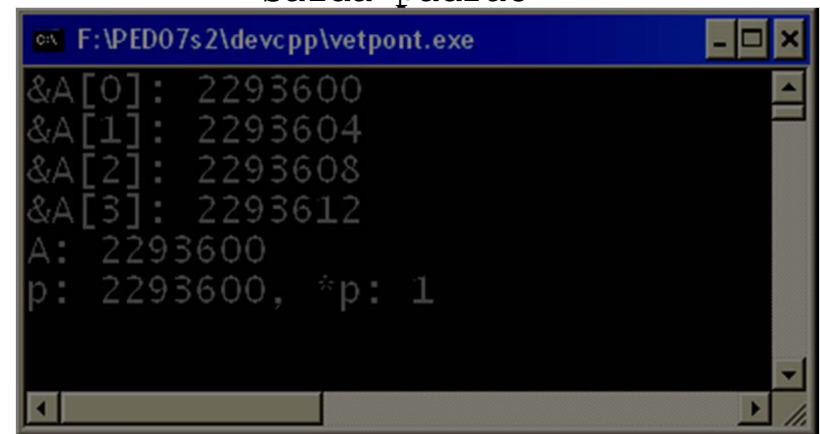


# Alocação Dinâmica de Memória

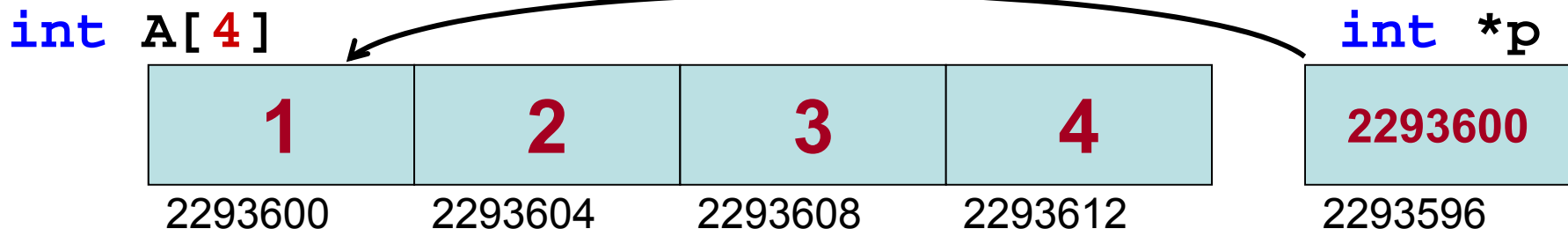
```
#include <stdio.h>
int main(){
    int A[4]={1,2,3,4};
    int *p;
    printf("&A[0]: %u\n",&A[0]);
    printf("&A[1]: %u\n",&A[1]);
    printf("&A[2]: %u\n",&A[2]);
    printf("&A[3]: %u\n",&A[3]);
    printf("A: %u\n",A);
    p = &A[0];
    printf("p: %u, *p: %d\n",p,*p);
    p = p + 2;
    printf("p: %u, *p: %d\n",p,*p);
    return 0;
}
```

- São impressos o conteúdo do ponteiro **p** (endereço de **A[0]**) e também o conteúdo da variável apontada por **p** (conteúdo de **A[0]**).

Saída padrão:



```
C:\ F:\PED07s2\devcpp\vetpont.exe
&A[0]: 2293600
&A[1]: 2293604
&A[2]: 2293608
&A[3]: 2293612
A: 2293600
p: 2293600, *p: 1
```

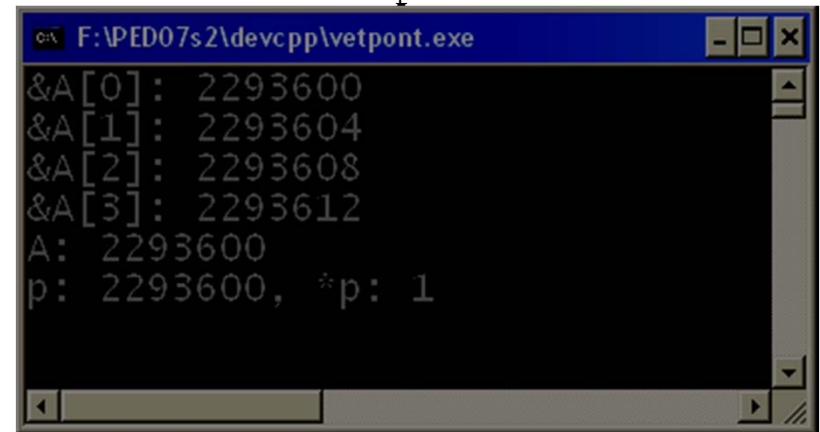


# Alocação Dinâmica de Memória

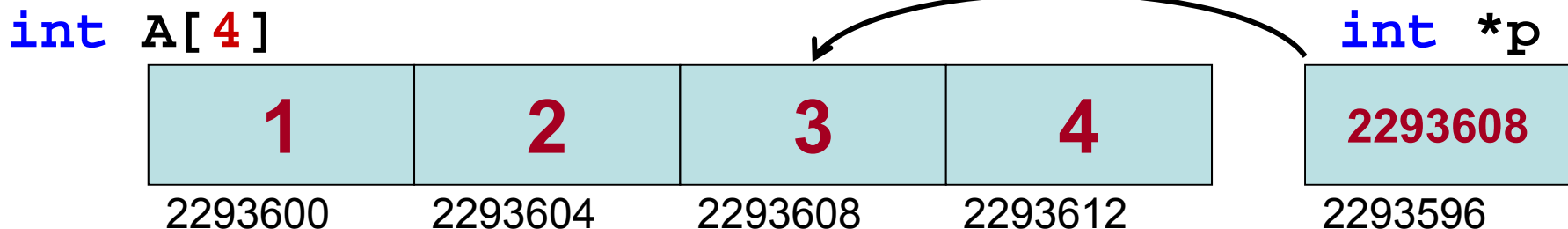
```
#include <stdio.h>
int main(){
    int A[4]={1,2,3,4};
    int *p;
    printf("&A[0]: %u\n",&A[0]);
    printf("&A[1]: %u\n",&A[1]);
    printf("&A[2]: %u\n",&A[2]);
    printf("&A[3]: %u\n",&A[3]);
    printf("A: %u\n",A);
    p = &A[0];
    printf("p: %u, *p: %d\n",p,*p);
    <-->p = p + 2;
    printf("p: %u, *p: %d\n",p,*p);
    return 0;
}
```

- Quando somamos **1** a um ponteiro, ele passa a apontar para o próximo elemento. Ou seja, sempre se desloca pelo tamanho do tipo apontado.

Saída padrão:



```
C:\ F:\PED07s2\devcpp\vetpont.exe
&A[0]: 2293600
&A[1]: 2293604
&A[2]: 2293608
&A[3]: 2293612
A: 2293600
p: 2293600, *p: 1
```

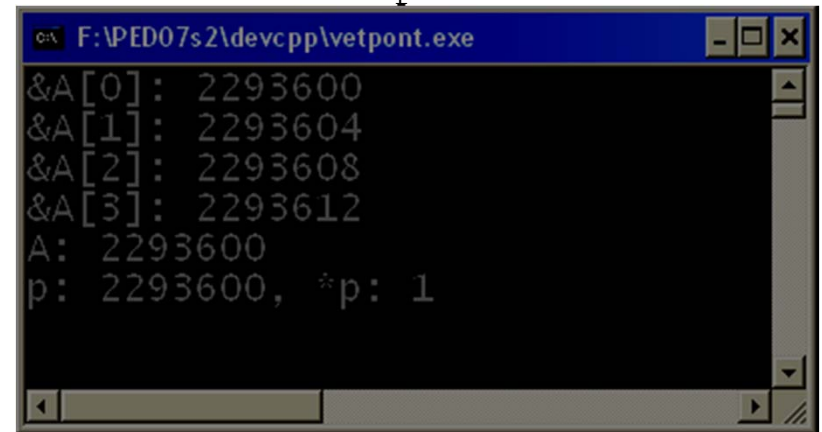


# Alocação Dinâmica de Memória

```
#include <stdio.h>
int main(){
    int A[4]={1,2,3,4};
    int *p;
    printf("&A[0]: %u\n",&A[0]);
    printf("&A[1]: %u\n",&A[1]);
    printf("&A[2]: %u\n",&A[2]);
    printf("&A[3]: %u\n",&A[3]);
    printf("A: %u\n",A);
    p = &A[0];
    printf("p: %u, *p: %d\n",p,*p);
    <=> p = p + 2;
    printf("p: %u, *p: %d\n",p,*p);
    return 0;
}
```

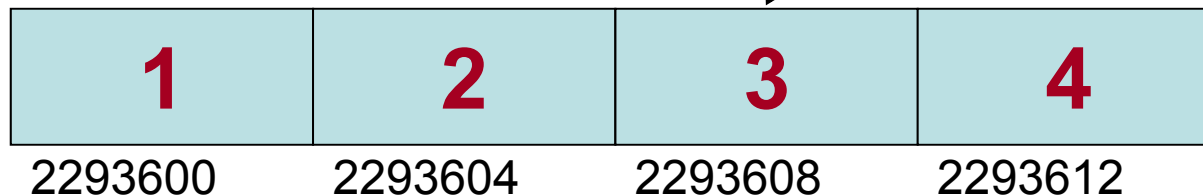
- Somando **2** obtemos o segundo elemento do tipo apontado após **A[0]**.
- No exemplo cada **1** somado vale por **4** (tamanho do **int**).

Saída padrão:

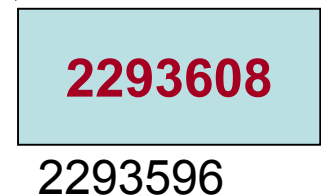


```
C:\ F:\PED07s2\devcpp\vetpont.exe
&A[0]: 2293600
&A[1]: 2293604
&A[2]: 2293608
&A[3]: 2293612
A: 2293600
p: 2293600, *p: 1
```

**int** A[4]



**int** \*p

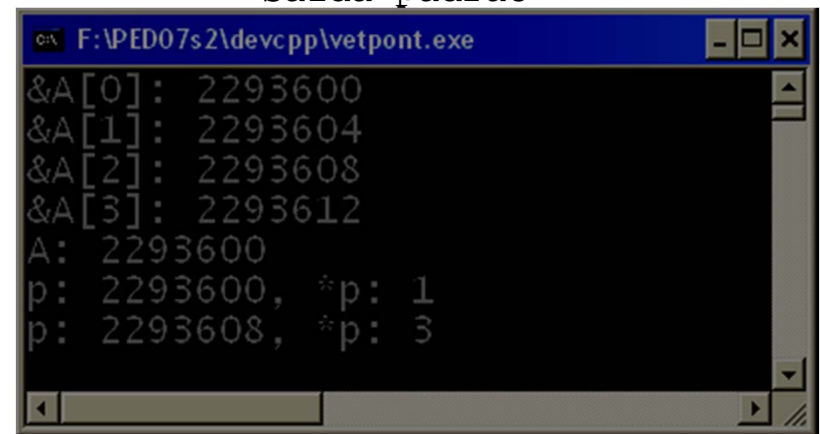


# Alocação Dinâmica de Memória

```
#include <stdio.h>
int main(){
    int A[4]={1,2,3,4};
    int *p;
    printf("&A[0]: %u\n",&A[0]);
    printf("&A[1]: %u\n",&A[1]);
    printf("&A[2]: %u\n",&A[2]);
    printf("&A[3]: %u\n",&A[3]);
    printf("A: %u\n",A);
    p = &A[0];
    printf("p: %u, *p: %d\n",p,*p);
    p = p + 2;
    printf("p: %u, *p: %d\n",p,*p);
    return 0;
}
```

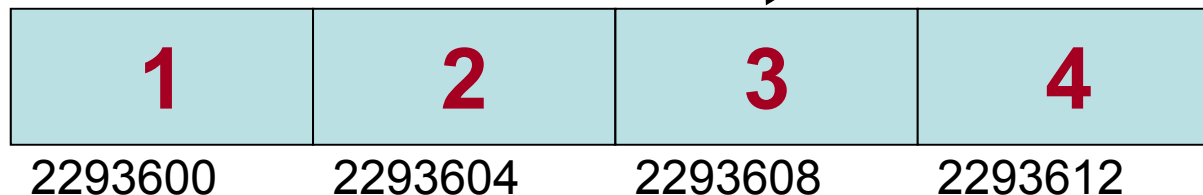
- São impressos o conteúdo do ponteiro **p** (endereço de **A[2]**) e também o conteúdo da variável apontada por **p** (conteúdo de **A[2]**).

Saída padrão:

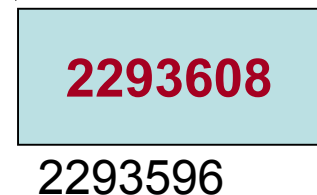


```
C:\ F:\PED07s2\devcpp\vetpont.exe
&A[0]: 2293600
&A[1]: 2293604
&A[2]: 2293608
&A[3]: 2293612
A: 2293600
p: 2293600, *p: 1
p: 2293608, *p: 3
```

**int** A[4]



**int** \*p



# Alocação Dinâmica de Memória

## •Alocação dinâmica de vetores:

–Vimos que o nome de um vetor é o endereço do seu primeiro elemento. Logo temos que o nome do vetor é um ponteiro constante, que armazena um endereço fixo que não pode ser alterado.

```
int A[4] = {1, 2, 3, 4};
```

–Portanto,  $A[1]$  equivale à  $*(A+1)$  em notação de ponteiros, pois quando somamos **1** a um ponteiro obtemos o endereço do próximo elemento do vetor.

–No caso geral temos que  $A[i]$  é o mesmo que  $*(A+i)$ .

# Alocação Dinâmica de Memória

## •Alocação dinâmica de vetores:

–A alocação dinâmica de vetores é análoga à alocação de variáveis anônimas. A diferença reside na quantidade de bytes a serem alocados pelo `malloc`.

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

int main(){
    int *p,i,n;
    n = 3;
    p = (int *)malloc(n*4);
    for(i=0; i<n; i++)
        p[i] = i+1;

    free(p);
    return 0;
}
```

# Alocação Dinâmica de Memória

## •Alocação dinâmica de vetores:

–A alocação dinâmica de vetores é análoga à alocação de variáveis anônimas. A diferença reside na quantidade de bytes a serem alocados pelo `malloc`.

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

int main(){
    int *p,i,n;
    n = 3;
    p = (int *)malloc(n*4);
    for(i=0; i<n; i++)
        p[i] = i+1;

    free(p);
    return 0;
}
```

- São alocados **12** bytes o equivalente a **3** variáveis inteiras de **4** bytes cada.
- Os elementos do vetor podem ser acessados através do ponteiro usando notação convencional de vetores `p[i]`.
- No exemplo ao lado, o vetor é inicializado com valores de **1** a **3**.

# Alocação Dinâmica de Memória

## •Alocação dinâmica de vetores:

–Para alocar vetores de outros tipos (ex: `float`, `char`) basta mudar o tipo do ponteiro, o **cast** de conversão e o número de bytes a serem alocados.

–Para obter o tamanho em bytes de qualquer tipo da linguagem basta chamar o operador **`sizeof()`**.

–A quantidade de bytes a ser alocada é obtida multiplicando pelo número `n` de elementos desejados.

```
tipo *p;  
p = (tipo *)malloc(n*sizeof(tipo));
```

–**Ex:**

```
float *p;  
p = (float *)malloc(n*sizeof(float));
```

```
struct Aluno *p;  
p = (struct Aluno *)malloc(n*sizeof(struct Aluno));
```



# Alocação Dinâmica de Matrizes

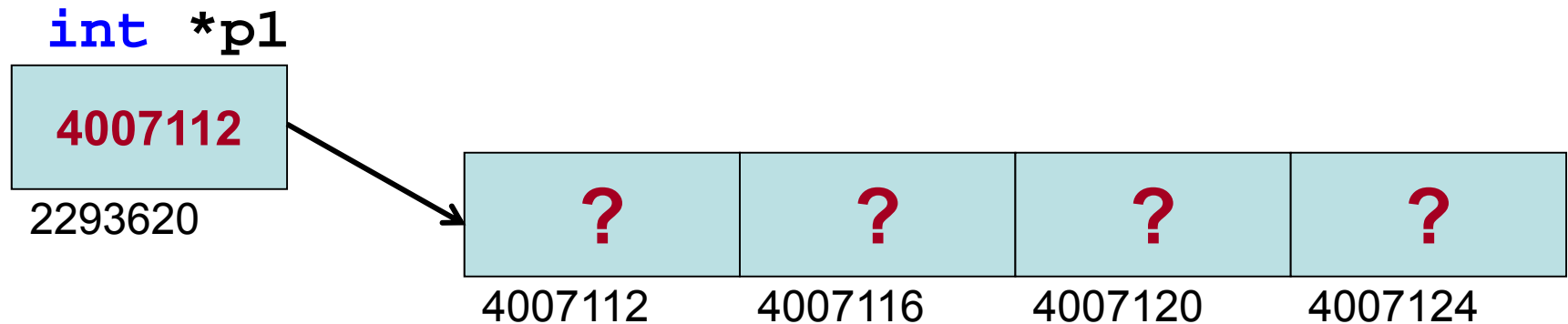
## •Introdução:

- Vimos que para alocar um vetor de forma dinâmica usamos a função **malloc** que retorna o endereço do primeiro elemento do vetor alocado.
- Precisamos de um ponteiro para guardar o endereço retornado.
- Os elementos do vetor podem ser acessados através do ponteiro usando notação convencional de vetores **p[i]** ou através da notação de ponteiros **\*(p+i)**.

# Alocação Dinâmica de Matrizes

- **Exemplo:** Alocando um vetor de inteiros

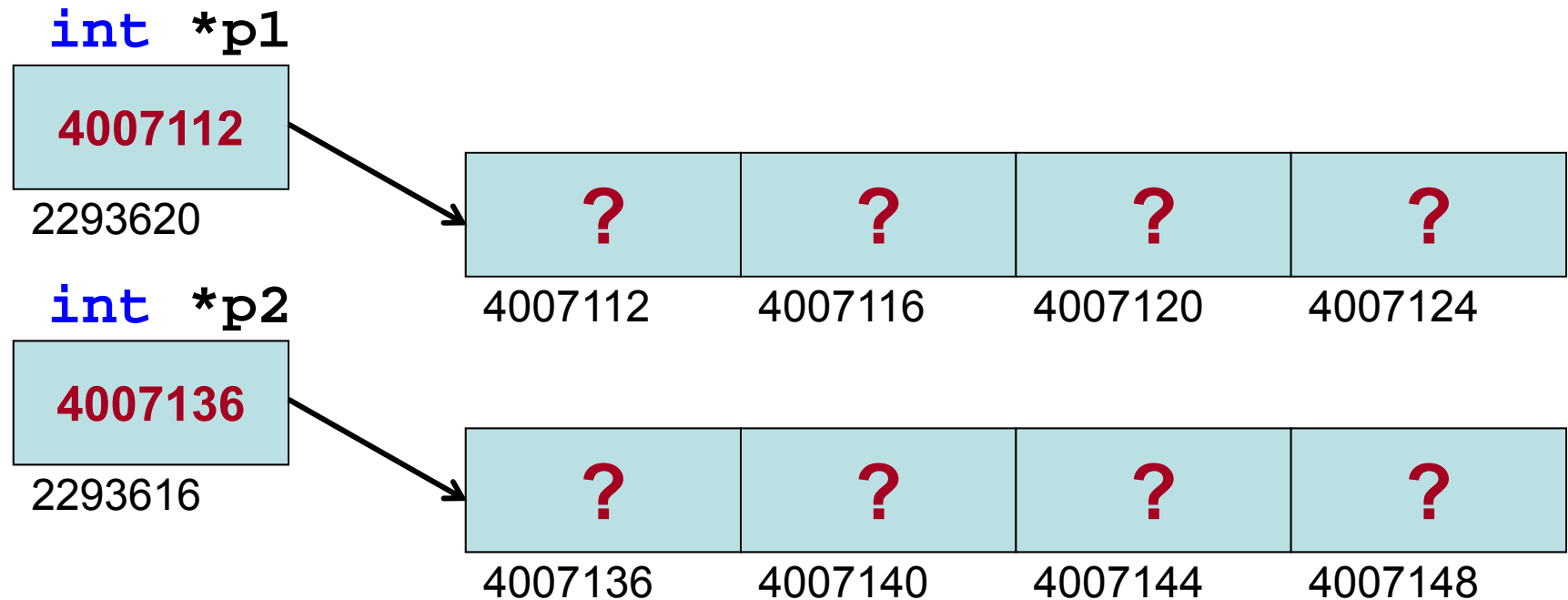
```
int *p1;  
p1 = (int *)malloc(4*sizeof(int));
```



# Alocação Dinâmica de Matrizes

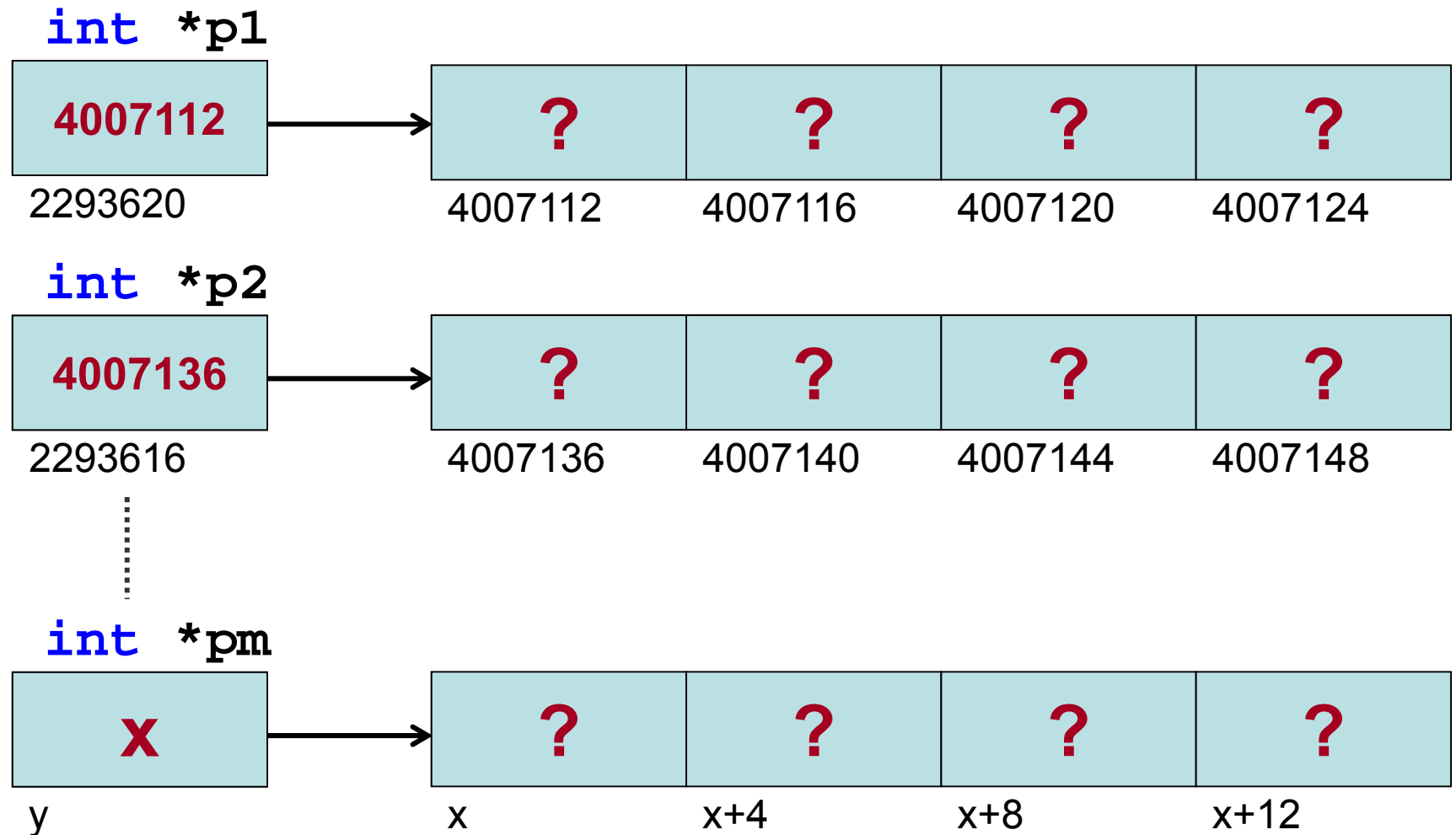
- **Exemplo:** Alocando dois vetores de inteiros

```
int *p1,*p2;  
p1 = (int *)malloc(4*sizeof(int));  
p2 = (int *)malloc(4*sizeof(int));
```



# Alocação Dinâmica de Matrizes

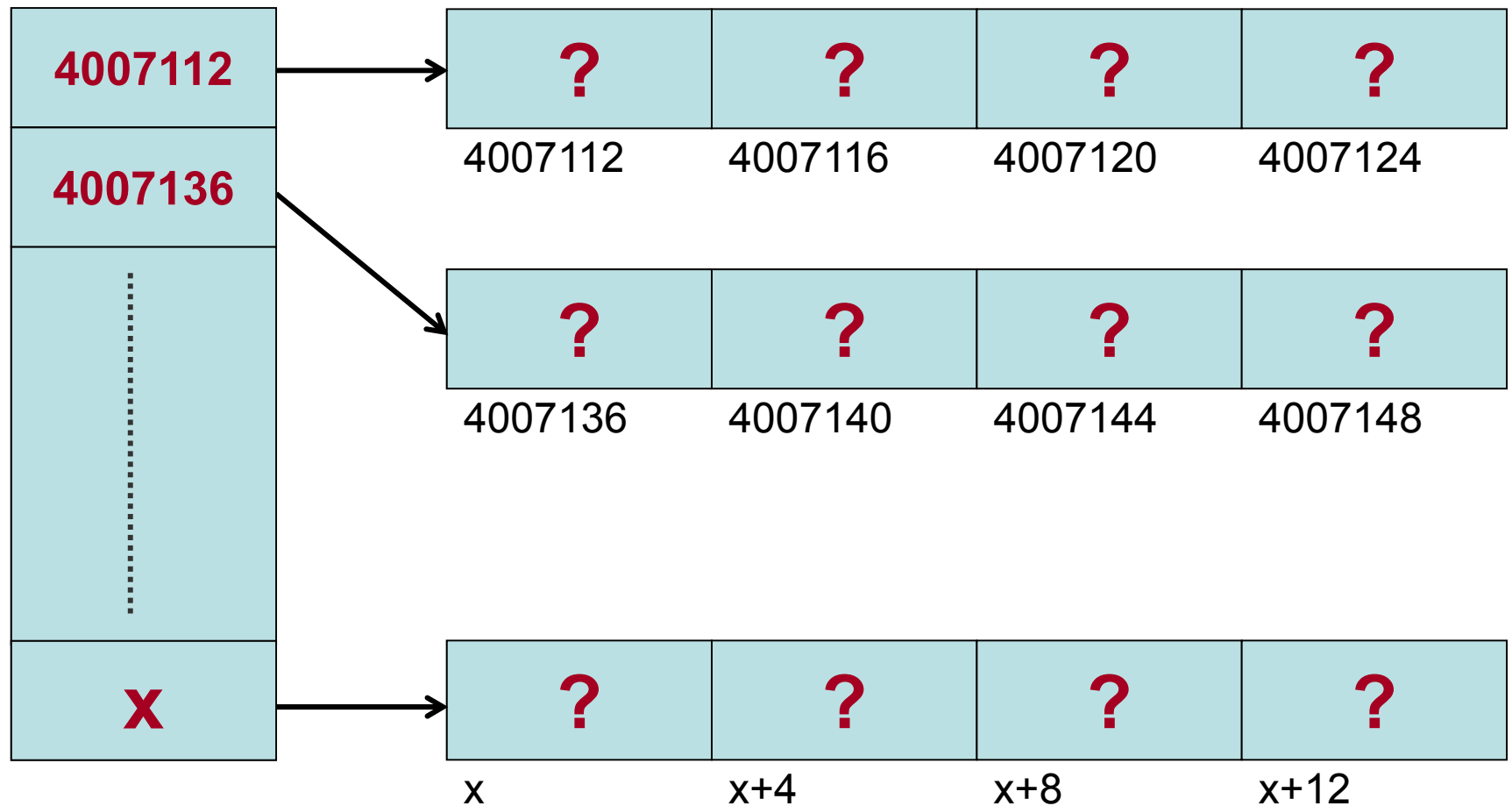
- **Exemplo:** Alocando “m” vetores de inteiros



# Alocação Dinâmica de Matrizes

- “m” vetores = vetor de vetores

```
int *p[LIM]
```



# Alocação Dinâmica de Matrizes

- **Alocação de memória para matrizes:**

- Matriz é um caso particular de vetor, onde os elementos são vetores (**vetor de vetores**).

- Devemos portanto alocar um vetor de apontadores e depois um vetor de elementos para cada linha.

- Para alocar um vetor de apontadores dinamicamente é necessário um apontador para apontadores (**ponteiro duplo**).

- **Desalocar a memória da matriz:**

- Para desalocar devemos chamar **free** para cada linha e também para o vetor de apontadores.

# Alocação Dinâmica de Matrizes

- Alocação de memória para matrizes:

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

int main(){
    int **M;
    int i,ncols=5,nrows=6;

    M = (int **)malloc(nrows*sizeof(int *));
    for(i=0; i<nrows; i++)
        M[i] = (int *)malloc(ncols*sizeof(int));

    /* Agora podemos acessar M[i][j]:
       Matriz M na linha i, coluna j.  */
    ...
    return 0;
}
```

# Alocação Dinâmica de Matrizes

- Desalocar a memória da matriz:

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

int main(){
    int **M;
    int i,ncols=5,nrows=6;

    ...

    /* Desaloca memória. */
    for(i=0; i<nrows; i++)
        free(M[i]);
    free(M);

    return 0;
}
```



# Alocação Dinâmica

## •Exercícios:

1)Escreva um programa que leia um número inteiro positivo  $n$  seguido de  $n$  números inteiros e imprima esses  $n$  números em ordem invertida. Por exemplo, ao receber

**5 22 33 44 55 88**

o seu programa deve imprimir:

**88 55 44 33 22**

O programa não deve impor limitações sobre o valor de  $n$ .