

# Algoritmos e Estruturas de Dados I

## Aula 03: Alocação Dinâmica

Prof. Márcio Porto Basgalupp

*créditos: Prof. Jurandy G. Almeida Jr.*

Universidade Federal de São Paulo  
Departamento de Ciência e Tecnologia

# O que é uma variável em um programa?

- Uma **variável** é um espaço de memória reservado para armazenar dados, que é composta por:
  - **Nome:**
    - Identificador para acessar o conteúdo
  - **Tipo:**
    - Determina a capacidade de armazenamento
    - Ex: int, char, float, ...
  - **Endereço:**
    - Posição na memória

## ■ Exemplo:

- Nome: `dia`
- Tipo: **`int`**
- Endereço: `0022FF74` (hexadecimal) ou  
`2293620` (decimal) ou  
`&dia` (representação simbólica)
- Conteúdo: **`27`**

`int dia`

`0022FF74` **`27`**

## ■ Definição:

- Apontador é uma **variável** que armazena um **endereço de memória**, por exemplo, o endereço de uma outra variável

## ■ Declaração:

```
tipo *nome_do_apontador;
```

## ■ Exemplos:

```
int *p;           // declara apontador para um int  
char *tmp;        // declara apontador para um char  
float *ptr;       // declara apontador para um float
```

## ■ Por que usar apontadores?

- Até agora, o acesso ao conteúdo das **variáveis** se dava através do **nome** delas
- Apontadores fornecem um novo **modo de acesso** que explora o **endereço** das variáveis
- Para isso, usa-se o **operador indireto** (\*), que permite ler e alterar o conteúdo das **variáveis apontadas** por um apontador

# Exemplo

```
int main(void)
{
    int dia;
    int *p;

    dia = 27;
    p = &dia;
    *p = 10;
    return 0;
}
```

```
int main(void)
{
    int dia;
    int *p;

    dia = 27;
    p = &dia;
    *p = 10;
    return 0;
}
```

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



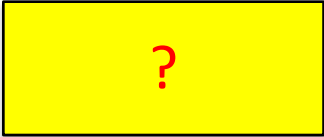
# Exemplo

```
int main(void)
{
    int dia;
    int *p;

    dia = 27;
    p = &dia;
    *p = 10;
    return 0;
}
```

int dia

0022FF74



int \*p

0022FF70



- As variáveis são **declaradas**
- O apontador, como toda variável, possui um endereço

# Exemplo

```
int main(void)
{
    int dia;
    int *p;

    dia = 27;
    p = &dia;
    *p = 10;
    return 0;
}
```

int dia

0022FF74

27

int \*p

0022FF70

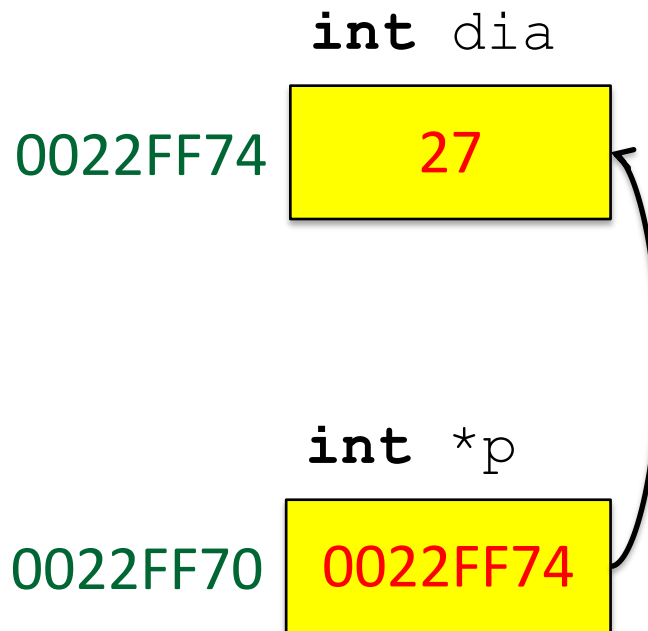
?

- O **conteúdo** da variável dia é alterado

# Exemplo

```
int main(void)
{
    int dia;
    int *p;

    dia = 27;
    p = &dia;
    *p = 10;
    return 0;
}
```

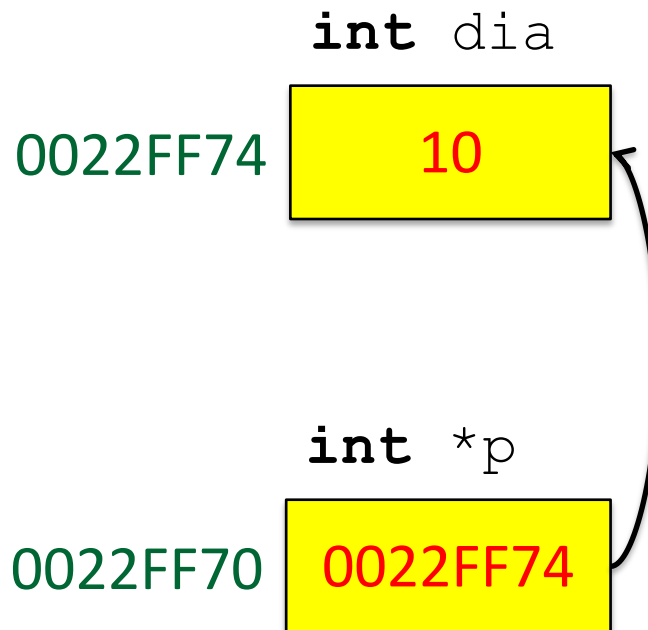


- O **endereço** de `dia` é atribuído para o apontador `p`
- Dizemos que `p` **aponta** para a variável `dia`

# Exemplo

```
int main(void)
{
    int dia;
    int *p;

    dia = 27;
    p = &dia;
    *p = 10;
    return 0;
}
```

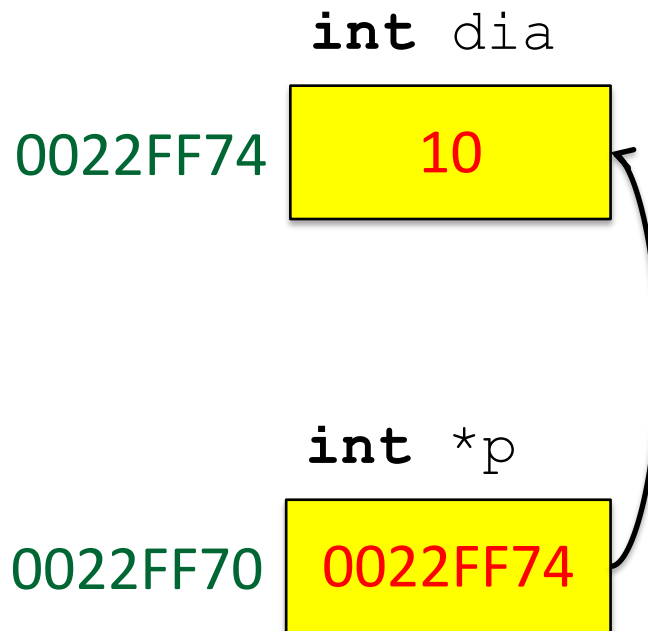


- O código `*p` é o **conteúdo da variável apontada** por `p`, ou seja, o conteúdo de `dia`, que recebe o valor **10**

# Exemplo

```
int main(void)
{
    int dia;
    int *p;

    dia = 27;
    p = &dia;
    *p = 10;
    return 0;
}
```



- A declaração `int *p;` indica que a variável `p` é um apontador para um inteiro e que `*p` é do tipo `int`

## ■ Alocação Estática:

- O espaço para as variáveis é **reservado automaticamente no início** da execução, o qual é liberado posteriormente pelo compilador

## ■ Alocação Dinâmica:

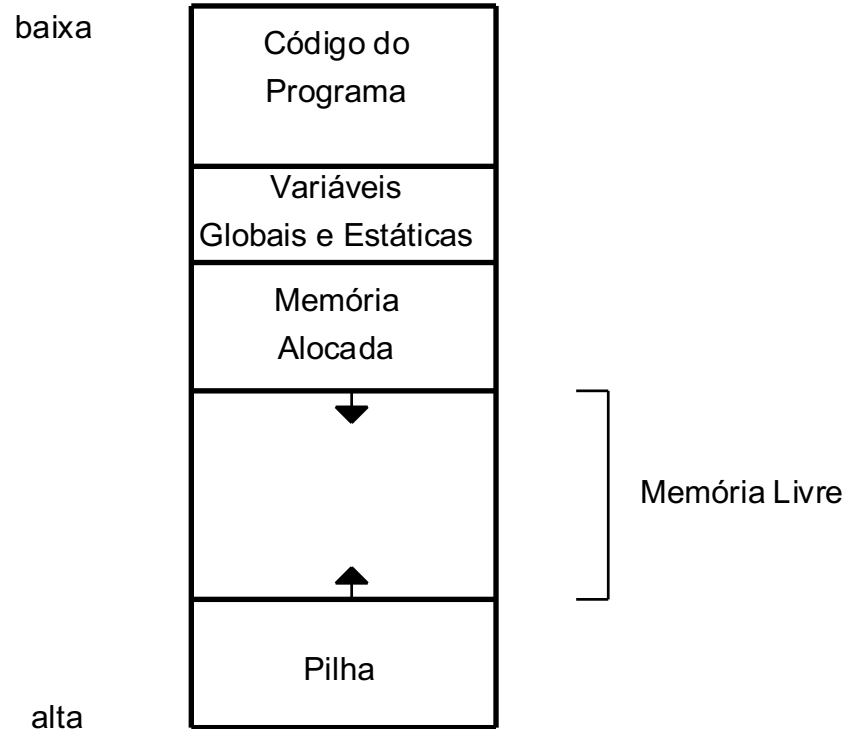
- O espaço para as variáveis é **alocado dinamicamente durante** a execução do programa pelo programador, que é o responsável por liberar esse espaço

- Até agora, era necessário **pré-fixar** o número de variáveis a serem utilizadas em um programa
- No caso de **vetores** e **matrizes**, o tamanho era fixado como sendo um **limitante superior** previsto
  - Problema: Desperdício de memória!
- Variáveis locais são armazenadas em uma parte da memória chamada **pilha**

- Existe uma parte da memória para a **alocação dinâmica**, que contém toda memória disponível não reservada para outras finalidades
- Nessa área, é possível **alocar** (reservar) espaço de memória de **tamanho arbitrário** em **tempo de execução**, tornando os programas mais flexíveis



# Esquema de Memória



Esquema da memória do sistema

- O espaço alocado dinamicamente faz parte de uma área de memória chamada **heap**
  - Basicamente, o programa aloca e desaloca porções de memória do *heap* durante a execução
- O acesso à memória alocada no *heap* é realizada por meio de **apontadores**

- Toda memória alocada **deve ser liberada** após o término de seu uso
- A liberação deve ser feita por quem fez a alocação:
  - **Alocação Estática:** compilador
  - **Alocação Dinâmica:** programador

- Funções da `stdlib.h`:
  - `malloc` ou `calloc` para **alocar** memória no *heap*
  - `realloc` para **alterar o tamanho** de um bloco de memória alocado, **preservando o conteúdo** já existente
  - `free` para **desalocar** (liberar) memória previamente alocada com `malloc`, `calloc` ou `realloc`

## ■ Função `malloc`:

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

- Aloca um bloco de memória no *heap* com tamanho em bytes dado pelo argumento `size`
- Retorna o endereço do primeiro byte de memória recém alocado ou o valor `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 apontador por meio de uma conversão (*cast*) para um tipo particular

## ■ Função `calloc`:

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

- Aloca um bloco de memória no *heap* com tamanho em bytes dado por `num*size`
- Retorna o endereço do primeiro byte de memória recém alocado ou o valor `NULL` em caso de falta de memória
- Ela também inicializa todo o conteúdo do bloco com zero

## ■ Função `realloc`:

```
void *realloc(void *ptr,  
              unsigned int size);
```

- Altera o tamanho do bloco de memória apontado por `ptr` para ter o tamanho em bytes dado pelo argumento `size`
- Se o apontador `ptr` armazenar o valor `NULL`, a função irá se comportar exatamente igual à função `malloc`
- Retorna o endereço do primeiro byte de memória recém alocado ou o valor `NULL` em caso de falta de memória

- Função `free`:

```
void free(void *ptr) ;
```

- Libera o espaço ocupado por um bloco de memória apontado por `ptr`, que foi previamente alocado no *heap*



# Exemplo

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

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

# Exemplo

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

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

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

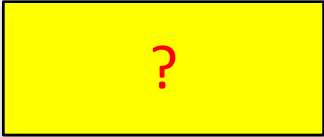
# Exemplo

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

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

int \*p

0022FF74

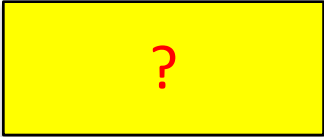


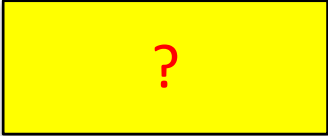
- É alocado espaço para as variáveis locais da função principal (**main**)

# Exemplo

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

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

int \*p  
0022FF74 

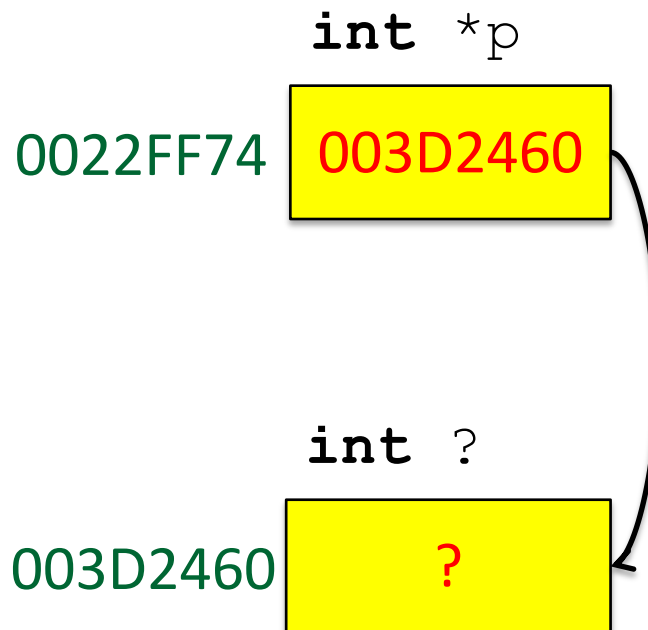
int ?  
003D2460 

- Comando `malloc` aloca 4 bytes de memória de forma dinâmica, i.e., tamanho de uma variável do tipo `int`

# Exemplo

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

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

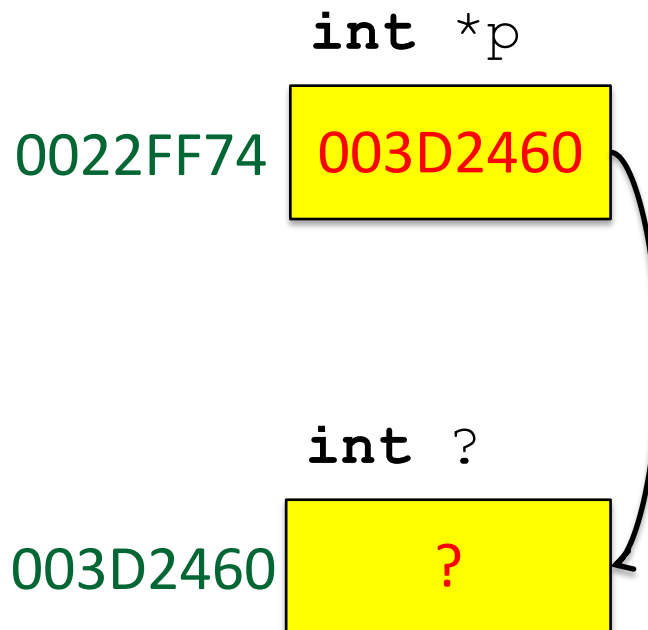


- A variável recém criada não possui nome (**anônima**)
- Seu endereço é retornado pela função `malloc` ...

# Exemplo

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

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

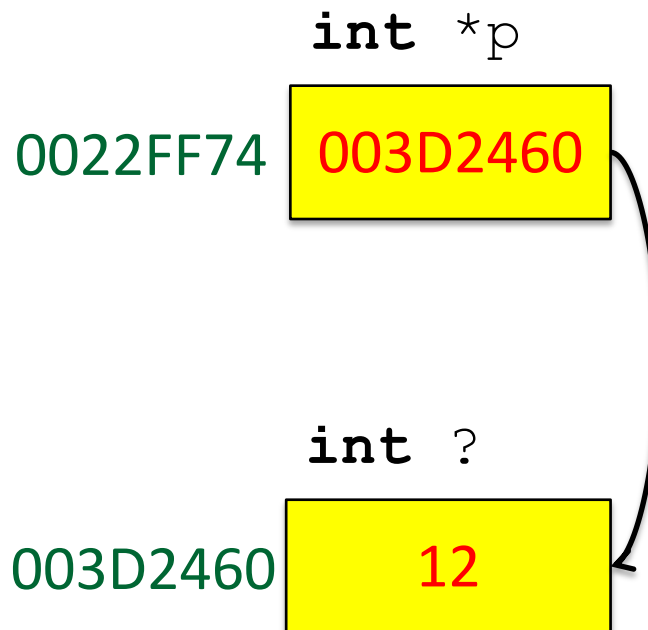


- ... e atribuído ao apontador `p`
- Dizemos que `p` aponta para a variável anônima

# Exemplo

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

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

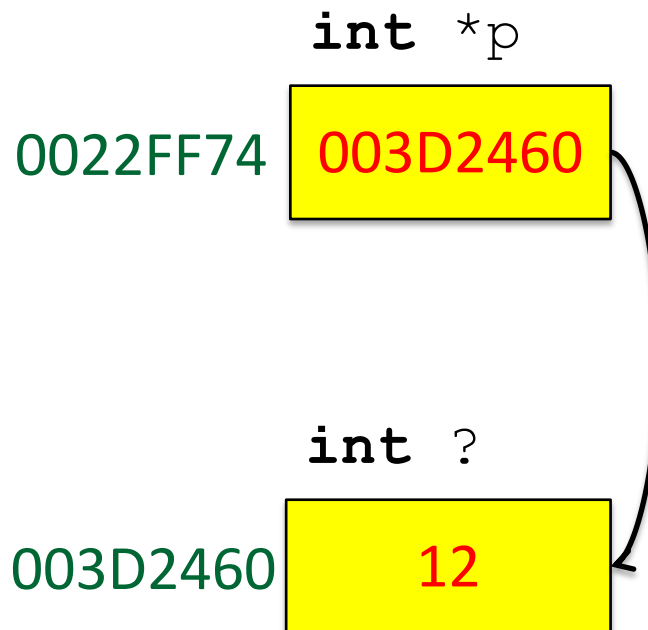


- A partir do apontador `p` é possível **alterar** o conteúdo (`*p`) da variável anônima apontada

# Exemplo

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

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



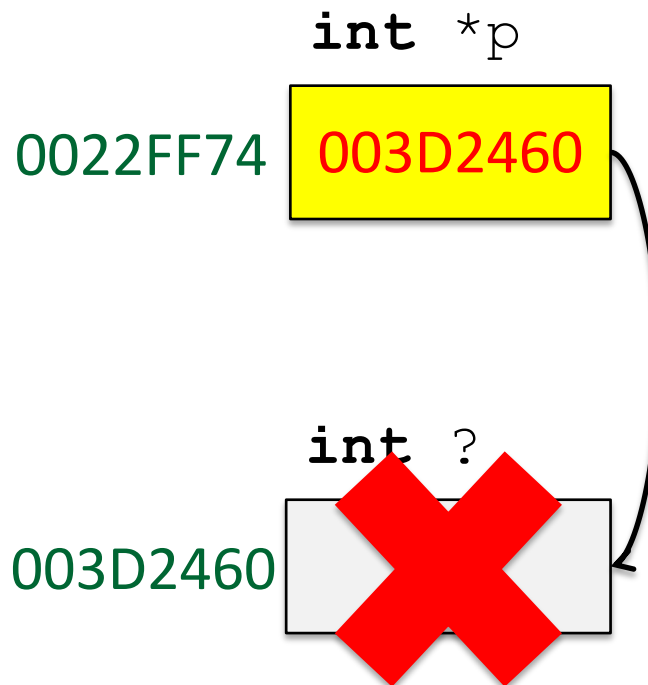
- O conteúdo **12** da variável anônima é impresso na saída padrão



# Exemplo

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

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



- A memória da variável anônima é desalocada, ficando disponível para futuras alocações

## ■ Alocação dinâmica de uma variável anônima:

- Para alocar variáveis de outros tipos (ex.: **float**, **char**) basta mudar o tipo do apontador, o *cast* de conversão e o número de bytes a serem alocados
- Na linguagem C, para obter o tamanho em bytes de qualquer tipo basta chamar o operador **sizeof** ( )

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

## ■ Alocação dinâmica de uma variável anônima:

- Para alocar variáveis de outros tipos (ex.: **float**, **char**) basta mudar o tipo do apontador, o *cast* de conversão e o número de bytes a serem alocados
- Na linguagem C, para obter o tamanho em bytes de qualquer tipo basta chamar o operador **sizeof** ( )

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

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

- Pergunta que não quer calar ...

```
int *p;
```

não é a declaração de um vetor de **int**?

- Pergunta que surge ...

```
int *p;
```

não é a declaração de um vetor de **int**?

- Na linguagem C, todo apontador pode se comportar como um vetor

# Exemplo

```
#include <stdio.h>

int main(void)
{
    int A[3] = {1, 2, 3};
    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: %u\n", A);
    p = &A[0]; // p = A;
    printf("p: %u,*p: %d\n", p, *p);
    p = p + 2;
    printf("p: %u,*p: %d\n", p, *p);
    return 0;
}
```

```
#include <stdio.h>

int main(void)
{
    int A[3] = {1, 2, 3};
    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: %u\n", A);
    p = &A[0]; // p = A;
    printf("p: %u,*p: %d\n", p, *p);
    p = p + 2;
    printf("p: %u,*p: %d\n", p, *p);
    return 0;
}
```

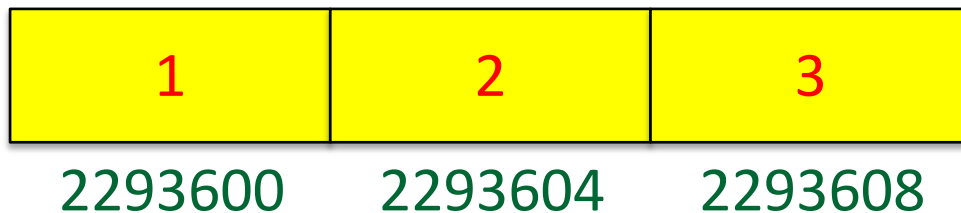
- O programa inicia a execução na função principal (**main**)

```
#include <stdio.h>

int main(void)
{
    int A[3] = {1, 2, 3};
    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: %u\n", A);
    p = &A[0]; // p = A;
    printf("p: %u,*p: %d\n", p, *p);
    p = p + 2;
    printf("p: %u,*p: %d\n", p, *p);
    return 0;
}
```

**int** A[3]



- É declarado um vetor com **3** elementos inteiros
- Para facilitar o entendimento, os endereços estão na base decimal



```
#include <stdio.h>

int main(void)
{
    int A[3] = {1, 2, 3};
    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: %u\n", A);
    p = &A[0]; // p = A;
    printf("p: %u,*p: %d\n", p, *p);
    p = p + 2;
    printf("p: %u,*p: %d\n", p, *p);
    return 0;
}
```

- É declarado um apontador para um inteiro

**int** A[3]



2293600

2293604

2293608

**int** \*p



2293596

# Exemplo

```
#include <stdio.h>

int main(void)
{
    int A[3] = {1, 2, 3};
    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: %u\n", A);
    p = &A[0]; // p = A;
    printf("p: %u,*p: %d\n", p, *p);
    p = p + 2;
    printf("p: %u,*p: %d\n", p, *p);
    return 0;
}
```

**int** A[3]



2293600

2293604

2293608

- São impressos os endereços de todos os elementos do vetor

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

**int** \*p



2293596

# Exemplo

```
#include <stdio.h>

int main(void)
{
    int A[3] = {1, 2, 3};
    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: %u\n", A);
    p = &A[0]; // p = A;
    printf("p: %u,*p: %d\n", p, *p);
    p = p + 2;
    printf("p: %u,*p: %d\n", p, *p);
    return 0;
}
```

**int** A[3]



2293600

2293604

2293608

- Os endereços são inteiros de **4** bytes em posições consecutivas

&A[0]: 2293600  
&A[1]: 2293604  
&A[2]: 2293608

**int** \*p



2293596

# Exemplo

```
#include <stdio.h>

int main(void)
{
    int A[3] = {1, 2, 3};
    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: %u\n", A);
    p = &A[0]; // p = A;
    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, que é o endereço do 1o elemento

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

**int** A[3]



2293600

2293604

2293608

**int** \*p



2293596

# Exemplo

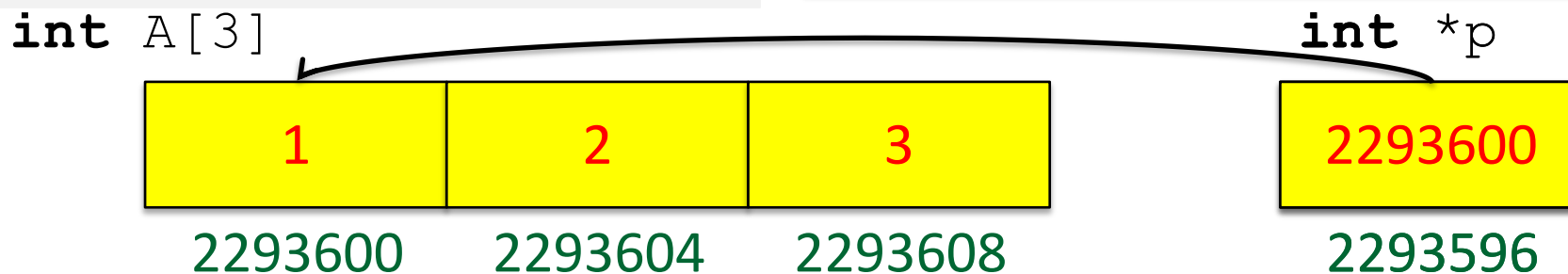
```
#include <stdio.h>

int main(void)
{
    int A[3] = {1, 2, 3};
    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: %u\n", A);
    p = &A[0]; // p = A;
    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 do vetor é atribuído ao apontador

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



# Exemplo

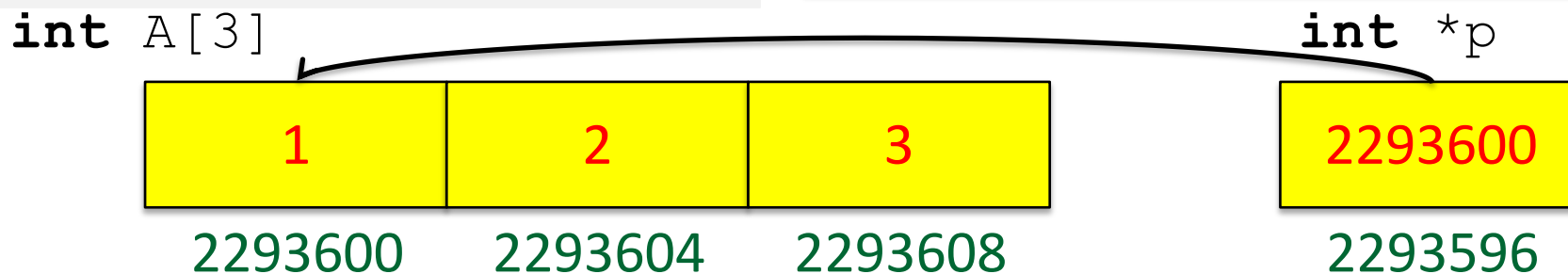
```
#include <stdio.h>

int main(void)
{
    int A[3] = {1, 2, 3};
    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: %u\n", A);
    p = &A[0]; // p = A;
    printf("p: %u,*p: %d\n", p, *p);
    p = p + 2;
    printf("p: %u,*p: %d\n", p, *p);
    return 0;
}
```

- São impressos o endereço do apontador e o conteúdo apontado

```
&A[0]: 2293600
&A[1]: 2293604
&A[2]: 2293608
A: 2293600
p: 2293600, *p: 1
```



# Exemplo

```
#include <stdio.h>

int main(void)
{
    int A[3] = {1, 2, 3};
    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: %u\n", A);
    p = &A[0]; // p = A;
    printf("p: %u,*p: %d\n", p, *p);
    p = p + 2;
    printf("p: %u,*p: %d\n", p, *p);
    return 0;
}
```

- Ao somar **1** em um apontador, ele aponta o próximo elemento

```
&A[0]: 2293600
&A[1]: 2293604
&A[2]: 2293608
A: 2293600
p: 2293600, *p: 1
```

**int** A[3]



# Exemplo

```
#include <stdio.h>

int main(void)
{
    int A[3] = {1, 2, 3};
    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: %u\n", A);
    p = &A[0]; // p = A;
    printf("p: %u,*p: %d\n", p, *p);
    p = p + 2;
    printf("p: %u,*p: %d\n", p, *p);
    return 0;
}
```

- Somando-se **2**, obtemos o segundo elemento do tipo apontado após **A[0]**

```
&A[0]: 2293600
&A[1]: 2293604
&A[2]: 2293608
A: 2293600
p: 2293600, *p: 1
```

**int** A[3]





# Exemplo

```
#include <stdio.h>

int main(void)
{
    int A[3] = {1, 2, 3};
    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: %u\n", A);
    p = &A[0]; // p = A;
    printf("p: %u,*p: %d\n", p, *p);
    p = p + 2;
    printf("p: %u,*p: %d\n", p, *p);
    return 0;
}
```

- São impressos o endereço do apontador e o conteúdo apontado

```
&A[0]: 2293600
&A[1]: 2293604
&A[2]: 2293608
A: 2293600
p: 2293600, *p: 1
p: 2293608, *p: 3
```

**int** A[3]

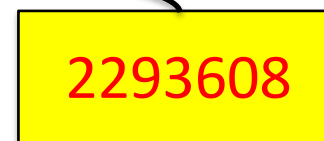


2293600

2293604

2293608

**int** \*p



2293596

## ■ Alocação dinâmica de vetores:

- O nome de um vetor é o endereço do seu primeiro elemento
- Logo, o **nome** de um vetor é um **apontador constante**, que armazena um endereço fixo que não pode ser alterado

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

- Portanto,  $A[1]$  equivale à  $\ast(A+1)$  em notação de apontadores, pois quando somamos 1 a um apontador, obtemos o endereço do próximo elemento do vetor
- No caso geral, temos que  $A[i]$  é o mesmo que  $\ast(A+i)$

## ■ 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`, `calloc` ou `realloc`

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

int main(void)
{
    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

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

int main(void)
{
    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 apontador usando a notação convencional de vetores

## ■ Alocação dinâmica de vetores:

- Para alocar vetores de outros tipos (ex.: **float**, **char**) basta mudar o tipo do apontador, o *cast* de conversão e o número de bytes a serem alocados
- A quantidade de bytes a ser alocada é obtida multiplicando pelo número **n** de elementos desejados

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

## ■ Alocação dinâmica de vetores:

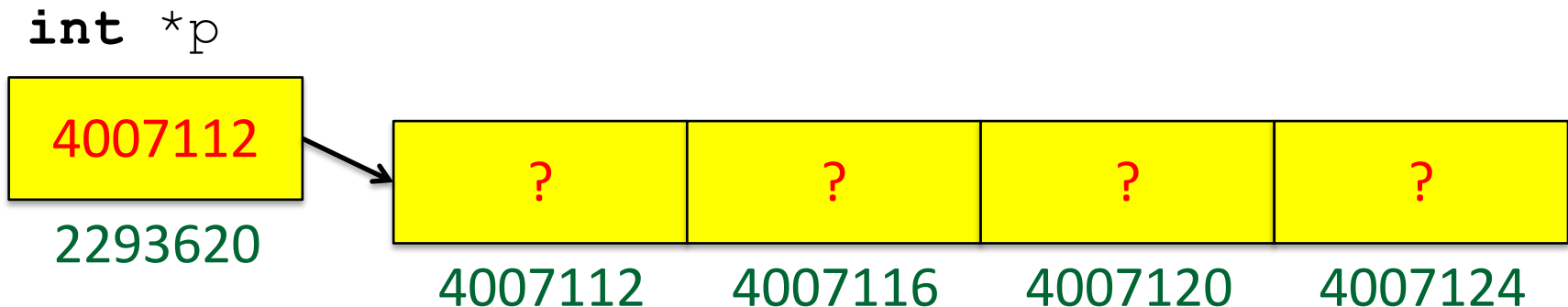
- Para alocar vetores de outros tipos (ex.: **float**, **char**) basta mudar o tipo do apontador, o *cast* de conversão e o número de bytes a serem alocados
- A quantidade de bytes a ser alocada é obtida multiplicando pelo número **n** de elementos desejados

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

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

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

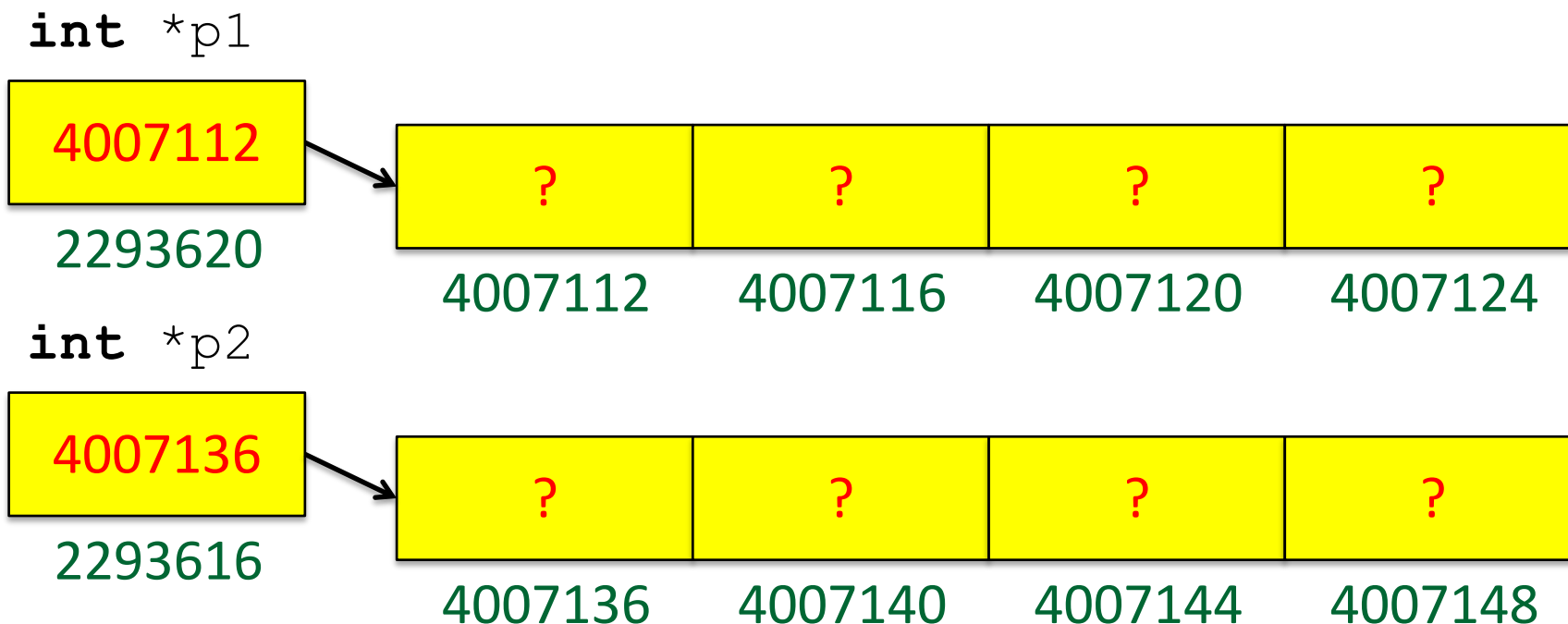
```
int *p;  
p = (int *) malloc(4 * sizeof(int));
```



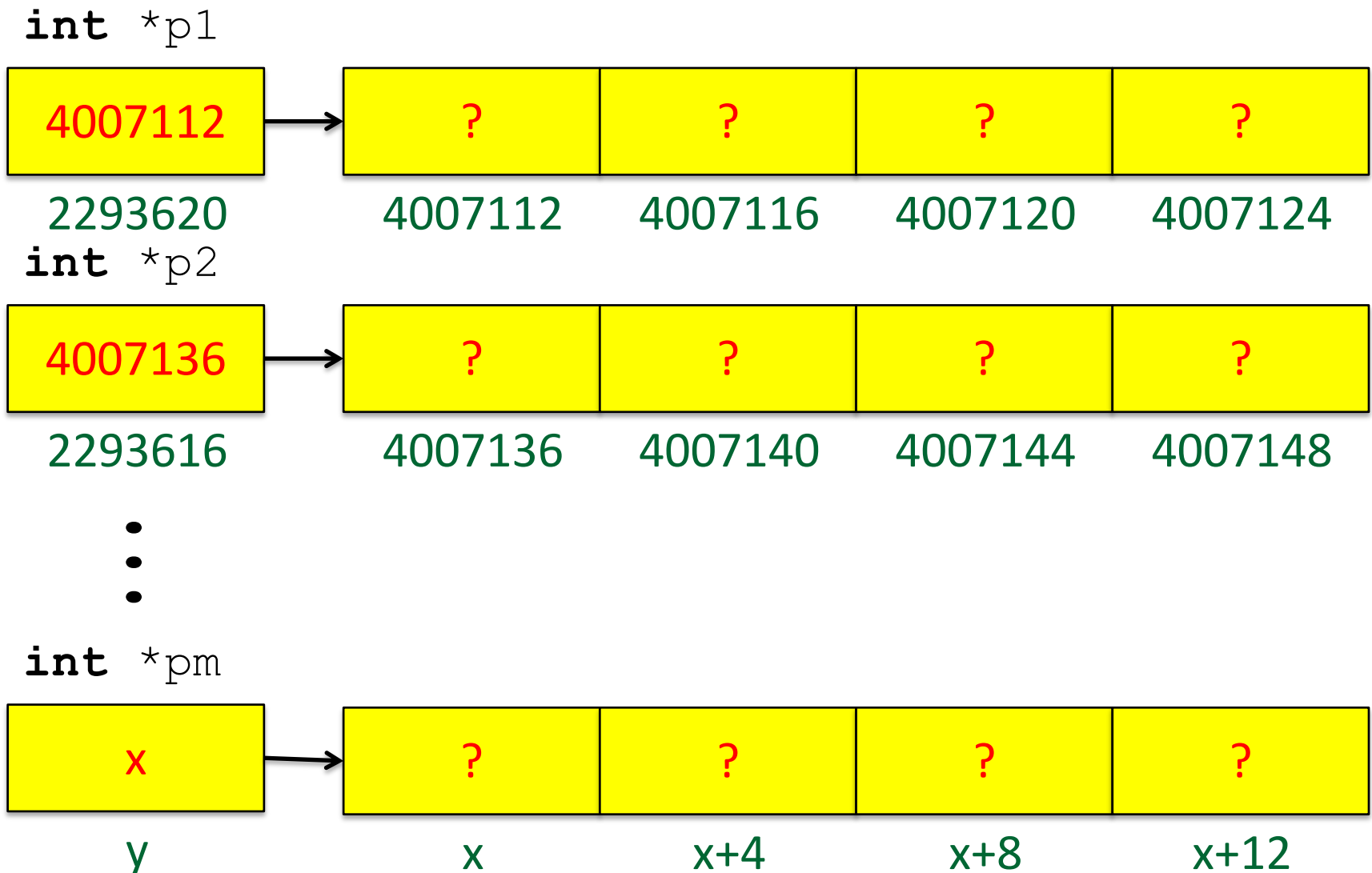


- Exemplo: Alocando **dois** vetores de inteiros

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

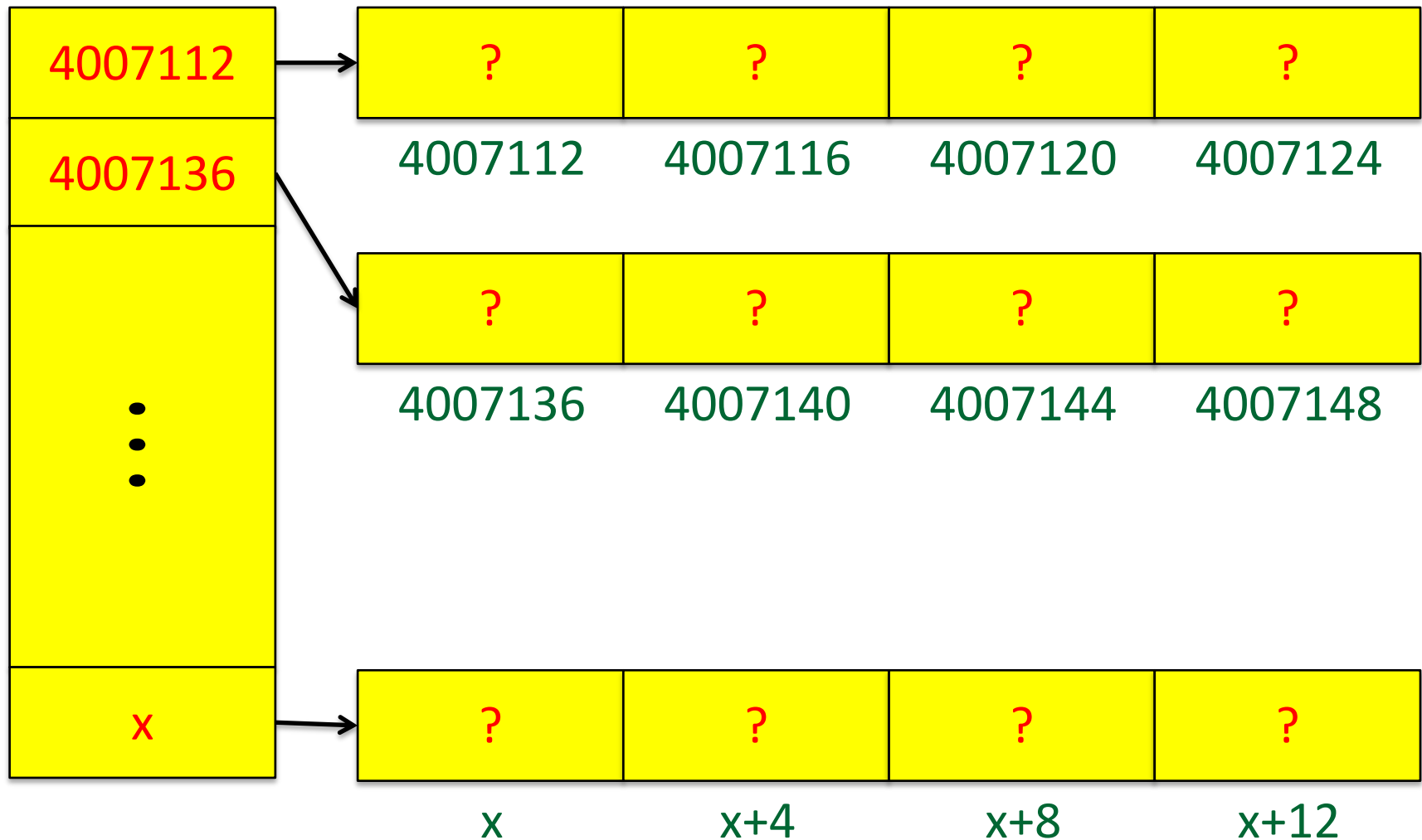


# Alocação Dinâmica



# Alocação Dinâmica

```
int *p[MAX]
```



## ■ Alocação dinâmica de matrizes:

- Uma matriz é um caso particular de um vetor, em que os elementos são vetores, ou seja, um **vetor de vetores**
- Portanto, devemos 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

## ■ Desalocar a memória da matriz:

- Para desalocar, devemos chamar `free` para cada linha e também para o vetor de apontadores

# Exemplo

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

int main(void)
{
    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
    ...

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

    return 0;
}
```

- Na linguagem Pascal, parâmetros para função podem ser passados **por valor** ou **por referência**:
  - **Por valor**: o parâmetro formal (recebido no procedimento) **é uma cópia** do parâmetro real (passado na chamada)
  - **Por referência**: o parâmetro formal (recebido no procedimento) **é uma referência** para o parâmetro real (passado na chamada)
    - Usa-se o termo **var** precedendo o parâmetro formal
- Na linguagem C só existe passagem por valor, logo deve-se implementar a passagem por referência utilizando-se apontadores

# Exemplo

```
#include <stdio.h>

void LeInteiro(int x)
{
    printf("Entre com x: ");
    scanf("%d", &x);
}

int main(void)
{
    int x;
    LeInteiro(x);
    printf("x: %d\n", x);
    return 0;
}
```

# Exemplo

```
#include <stdio.h>

void LeInteiro(int x)
{
    printf("Entre com x: ");
    scanf("%d", &x);
}

int main(void)
{
    int x;
    LeInteiro(x);
    printf("x: %d\n", x);
    return 0;
}
```

**Qual valor será impresso na saída padrão?**



# Exemplo

```
#include <stdio.h>

void LeInteiro(int x)
{
    printf("Entre com x: ");
    scanf("%d", &x);
}

int main(void)
{
    int x;
    LeInteiro(x);
    printf("x: %d\n", x);
    return 0;
}
```

```
#include <stdio.h>

int LeInteiro()
{
    int x;
    printf("Entre com x: ");
    scanf("%d", &x);
    return x;
}

int main(void)
{
    int x;
    x = LeInteiro();
    printf("x: %d\n", x);
    return 0;
}
```

# Exemplo

```
#include <stdio.h>

void LeInteiro(int x)
{
    printf("Entre com x: ");
    scanf("%d", &x);
}

int main(void)
{
    int x;
    LeInteiro(x);
    printf("x: %d\n", x);
    return 0;
}
```

```
#include <stdio.h>

int LeInteiro()
{
    int x;
    printf("Entre com x: ");
    scanf("%d", &x);
    return x;
}

int main(void)
{
    int x;
    x = LeInteiro();
    printf("x: %d\n", x);
    return 0;
}
```

**E se for necessário alterar duas ou mais variáveis?**

# Exemplo

```
#include <stdio.h>

void LeInteiro(int x)
{
    printf("Entre com x: ");
    scanf("%d", &x);
}

int main(void)
{
    int x;
    LeInteiro(x);
    printf("x: %d\n", x);
    return 0;
}
```

```
#include <stdio.h>

void LeInteiro(int *p)
{
    int x;
    printf("Entre com x: ");
    scanf("%d", &x);
    *p = x;
}

int main(void)
{
    int x;
    LeInteiro(&x);
    printf("x: %d\n", x);
    return 0;
}
```

- E para alocar memória dentro de um procedimento?
  - Na linguagem Pascal, basta passar a variável (apontador) como referência
  - Na linguagem C também, mas como não há passagem por referência as coisas são um pouco mais complicadas

- E para alocar memória dentro de um procedimento?

```
#include <stdlib.h>

void aloca(int *x, int n)
{
    x = (int *) malloc(n * sizeof(int));
    x[0] = 20;
}

int main(void)
{
    int *a;
    aloca(a, 10);
    a[1] = 40;
}
```

- E para alocar memória dentro de um procedimento?

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

```
void aloca(int *x, int n)
```

```
{
```

```
    x = (int *) malloc(n * sizeof(int));
```

```
    x[0] = 20;
```

```
}
```

```
int main(void)
```

```
{
```

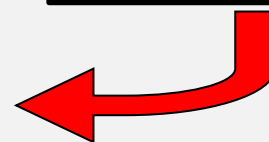
```
    int *a;
```

```
    aloca(a, 10);
```

```
    a[1] = 40;
```

```
}
```

Alocando memória  
para a cópia de \*x



- E para alocar memória dentro de um procedimento?

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

```
void aloca(int *x, int n)
{
    x = (int *) malloc(n * sizeof(int));
    x[0] = 20;
}
```

Alocando memória  
para a **cópia** de \*x

```
int main(void)
{
    int *a;
    aloca(a, 10);
    a[1] = 40;
}
```

**Error!**  
**Access Violation!**

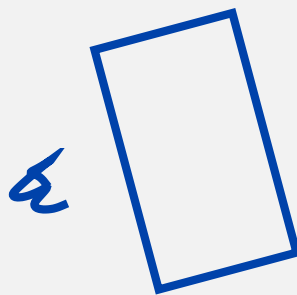


- E para alocar memória dentro de um procedimento?

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

```
void aloca(int **x, int n)
{
    (*x) = (int *) malloc(n * sizeof(int));
    (*x)[0] = 20;
}
```

```
int main(void)
{
    int *a;
    aloca(&a, 10);
    a[1] = 40;
}
```





- É comum usar apontadores com tipos estruturados

```
#include <stdlib.h>

typedef struct {
    int idade;
    double salario;
} TRegistro;

int main(void)
{
    TRegistro *reg;

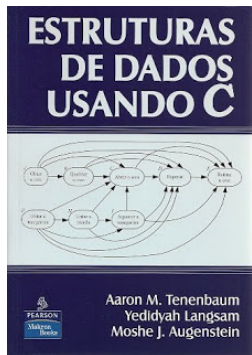
    ...

    reg = (TRegistro *) malloc(sizeof(TRegistro));
    reg->idade = 30;      // (*a).idade = 30
    reg->salario = 80;    // (*a).salario = 80

    return 0;
}
```

- **Esquecer de alocar memória e tentar acessar o conteúdo da variável**
- Copiar o valor do apontador em vez do valor da variável apontada
- Esquecer de desalocar memória
  - A memória deve ser desalocada ao fim do programa, procedimento ou função em que a variável é declarada, mas pode ser um grande problema em loops
- Tentar acessar o conteúdo da variável depois de desalocá-la

- Faça um programa que leia um valor ***n***, crie dinamicamente um vetor de ***n*** elementos, passe esse vetor pra uma função que vai ler os elementos desse vetor e depois para outra que vai imprimir todos os elementos desse vetor na saída padrão



TENENBAUM, A. M.; LANGSAM, Y.;  
AUGENSTEIN, M. J. **Estruturas de Dados  
usando C**. Pearson Makron Books, 2008.

## Capítulo 1