

Aula 11:

Ponteiros e Alocação Dinâmica de Memória

Disciplina: Fundamentos de Programação

Prof. Luiz Olmes

olmes@unifei.edu.br



Nas aulas anteriores...

▶ **O QUE JÁ ESTUDAMOS?**

- ▶ Algoritmos.
- ▶ Linguagem C.
- ▶ Variáveis, operadores e tipos.
- ▶ Estruturas de controle condicionais.
- ▶ Estruturas de controle iterativas.
- ▶ Vetores.
- ▶ Matrizes.
- ▶ Strings.
- ▶ Estruturas.

▶ **OBJETIVOS:**

- ▶ Ponteiros:
 - ▶ Definição.
 - ▶ Manipulação.
 - ▶ Operações aritméticas.
- ▶ Alocação Dinâmica:
 - ▶ Funções `malloc`, `calloc` e `free`.
 - ▶ Alocação de estruturas.
 - ▶ Alocação de arrays.

Introdução

- ▶ Ponteiros estão entre os recursos mais avançados, poderosos e perigosos das linguagens C e C++.
- ▶ Dentre as linguagens de programação modernas, ponteiros são diretamente suportados, sem restrições, apenas em C e C++.
 - ▶ Presentes com restrições em outras linguagens: Pascal, Fortran, Basic...
 - ▶ Disfarçados sob o nome de “referência” em Java, C#, entre outras.
- ▶ Basicamente, ponteiros são utilizados em 3 situações:
 - ▶ Para **referenciar** uma variável na memória.
 - ▶ Para **passagem** de parâmetros **por referência** à funções (*próximas aulas*).
 - ▶ Para realizar **alocação dinâmica** de memória (*próxima aula*).

Definição

- ▶ Um **ponteiro** é um tipo especial de variável que **permite armazenar endereços de memória** em vez de dados numéricos (como os tipos `int`, `float`, `double` e `char`).
- ▶ Variável × Ponteiro:
 - ▶ **Variável**: é um espaço reservado de memória usado para guardar um **valor** que pode ser modificado pelo programa.
 - ▶ **Ponteiro**: é um espaço reservado de memória usado para guardar um **endereço** de memória.
- ▶ A linguagem C permite declarar ponteiros de qualquer tipo primitivo (`int`, `float`, `double`, `char`, e `void`), e também de tipos definidos pelo programador (`struct`).

Declaração

- ▶ Do mesmo modo que as variáveis, **ponteiros** obrigatoriamente **possuem** um **tipo** e um **nome**, e devem ser declarados antes que possam ser utilizados.
- ▶ Ponteiros são declarados através do operador asterisco (*) precedendo o seu nome:

```
tipo *nome_do_ponteiro;
```

- ▶ Apesar de usar o mesmo símbolo, o operador * (**multiplicação**) não é o mesmo operador que o * (**ponteiros**). O significado do operador asterisco (*) depende de como ele é utilizado dentro do programa.

Declaração

- ▶ Um ponteiro é dito **apontar** para uma variável de seu tipo:
 - ▶ Um ponteiro de **int** aponta para uma variável do tipo **int**, ou seja, só pode armazenar o endereço de uma variável do tipo **int**.
 - ▶ Um ponteiro de **float** aponta para uma variável do tipo **float**, ou seja, só pode armazenar o endereço de uma variável do tipo **float**.

▶ Exemplo:

```
int main()
{
    int *p, i;
    char *s;
    ...
}
```

Declaração

- ▶ Um ponteiro é dito **apontar** para uma variável de seu tipo:
 - ▶ Um ponteiro de **int** aponta para uma variável do tipo **int**, ou seja, só pode armazenar o endereço de uma variável do tipo **int**.
 - ▶ Um ponteiro de **float** aponta para uma variável do tipo **float**, ou seja, só pode armazenar o endereço de uma variável do tipo **float**.

- ▶ Exemplo:

```
int main()
{
    int *p, i;
    char *s;
    ...
}
```

p é um ponteiro do tipo int

i é uma variável do tipo int

s é um ponteiro do tipo char

Manipulação de ponteiros

- ▶ As operações mais comuns de manipulação de ponteiros envolvem:
 - ▶ Inicialização de ponteiros.
 - ▶ O uso dos operadores `&` e `*`.
 - ▶ Operações de atribuição e comparação entre ponteiros.
 - ▶ Operações aritméticas sobre ponteiros.
 - ▶ Alocação dinâmica de memória sob a forma de *arrays*.

Inicialização de ponteiros

- ▶ A simples declaração de um ponteiro faz com que ele aponte para um lugar indefinido.
- ▶ Qualquer tentativa de utilizar ponteiros não inicializados causa um comportamento indefinido no programa.
 - ▶ Geralmente, *segmentation fault*.
- ▶ A ocorrência de erros se deve ao fato do ponteiro conter um endereço inválido ou apontar para regiões de memória não pertencentes ao programa.

Inicialização de ponteiros

- ▶ Ponteiros devem SEMPRE ser inicializados antes de serem usados.
- ▶ Existem duas formas de inicializar um ponteiro: apontando para lugar nenhum ou apontando para uma variável.

Inicialização de ponteiros

- ▶ Ponteiros devem **SEMPRE** ser inicializados antes de serem usados.
- ▶ Existem duas formas de inicializar um ponteiro: **apontando para lugar nenhum** ou **apontando para uma variável**.
- ▶ **Apontando para lugar nenhum**: basta atribuir a constante **NULL** ao ponteiro em sua declaração:

```
int main()
{
    int *p = NULL;
    ...
}
```

Inicialização de ponteiros

- ▶ Ponteiros devem SEMPRE ser inicializados antes de serem usados.
- ▶ Existem duas formas de inicializar um ponteiro: **apontando para lugar nenhum** ou **apontando para uma variável**.
- ▶ **Apontando para lugar nenhum**: basta atribuir a constante **NULL** ao ponteiro em sua declaração:

```
int main()
{
    int *p = NULL;
    ...
}
```

- ▶ **Apontando para uma variável**: através da utilização do operador **&** (visto a seguir).

Operadores & e *

- ▶ Existem dois operadores especiais para ponteiros: **&** e *****.
- ▶ O **operador &** é um operador unário que devolve o **endereço** de memória do seu operando.
 - ▶ Utilizado na inicialização de ponteiros:
 1. `int a = 2; // declara um inteiro 'a'`
 2. `int *x = &a; // faz o ponteiro apontar para o inteiro 'a'`
- ▶ O **operador *** é o complemento de **&**. Trata-se de um operador unário que devolve o **conteúdo**, ou seja, o valor armazenado no endereço de uma variável.

Operadores & e *

▶ Exemplo:

```
1. #include <stdio.h>
2. int main()
3. {
4.     int a = 2;
5.     int *x = NULL;
6.
7.     x = &a;
8.
9.     printf("Endereco de A:\t %p \n", &a);
10.    printf("X aponta para:\t %p \n", x);
11.    printf("Endereco de X:\t %p \n", &x);
12.    printf("Conteudo de X:\t %d \n", *x);
13.
14.    return 0;
15. }
```

Operadores & e *

▶ Exemplo:


```
1. #include <stdio.h>
2. int main()
3. {
4.     int a = 2;
5.     int *x = NULL;
6.
7.     x = &a;
8.
9.     printf("Endereco de A:\t %p \n", &a);
10.    printf("X aponta para:\t %p \n", x);
11.    printf("Endereco de X:\t %p \n", &x);
12.    printf("Conteudo de X:\t %d \n", *x);
13.
14.    return 0;
15. }
```



%p para exibir endereço de memória.

Operadores & e *

▶ Exemplo:

```
1. #include <stdio.h>
2. int main()
3. {
4.  int a = 2;
5.   int *x = NULL;
6.
7.   x = &a;
8.
9.   printf("Endereco de A:\t %p \n", &a);
10.  printf("X aponta para:\t %p \n", x);
11.  printf("Endereco de X:\t %p \n", &x);
12.  printf("Conteudo de X:\t %d \n", *x);
13.
14.   return 0;
15. }
```

Memória		
Endereço	Variável	Valor
...		
28FF18	int a	2
...		
...		

Operadores & e *

▶ Exemplo:

```
1. #include <stdio.h>
2. int main()
3. {
4.     int a = 2;
5.     int *x = NULL;
6.
7.     x = &a;
8.
9.     printf("Endereco de A:\t %p \n", &a);
10.    printf("X aponta para:\t %p \n", x);
11.    printf("Endereco de X:\t %p \n", &x);
12.    printf("Conteudo de X:\t %d \n", *x);
13.
14.    return 0;
15. }
```

Memória		
Endereço	Variável	Valor
...		
28FF18	int a	2
28FF1C	int *x	NULL
...		

Operadores & e *

▶ Exemplo:

```
1. #include <stdio.h>
2. int main()
3. {
4.     int a = 2;
5.     int *x = NULL;
6.
7.     x = &a;
8.
9.     printf("Endereco de A:\t %p \n", &a);
10.    printf("X aponta para:\t %p \n", x);
11.    printf("Endereco de X:\t %p \n", &x);
12.    printf("Conteudo de X:\t %d \n", *x);
13.
14.    return 0;
15. }
```

Memória		
Endereço	Variável	Valor
...		
28FF18	int a	2
28FF1C	int *x	28FF18
...		



Operadores & e *

Exemplo:

```
1. #include <stdio.h>
2. int main()
3. {
4.     int a = 2;
5.     int *x = NULL;
6.
7.     x = &a;
8.
9.     printf("Endereco de A:\t %p \n", &a);
10.    printf("X aponta para:\t %p \n", x);
11.    printf("Endereco de X:\t %p \n", &x);
12.    printf("Conteudo de X:\t %d \n", *x);
13.
14.    return 0;
15. }
```

Endereço de a

Memória		
Endereço	Variável	Valor
...		
28FF18	int a	2
28FF1C	int *x	28FF18
...		

Operadores & e *

Exemplo:

```
1. #include <stdio.h>
2. int main()
3. {
4.     int a = 2;
5.     int *x = NULL;
6.
7.     x = &a;
8.
9.     printf("Endereco de A:\t %p \n", &a);
10.    printf("X aponta para:\t %p \n", x);
11.    printf("Endereco de X:\t %p \n", &x);
12.    printf("Conteudo de X:\t %d \n", *x);
13.
14.    return 0;
15. }
```

Memória		
Endereço	Variável	Valor
...		
28FF18	int a	2
28FF1C	int *x	28FF18
...		

x aponta para o endereço de a

Operadores & e *

Exemplo:

```
1. #include <stdio.h>
2. int main()
3. {
4.     int a = 2;
5.     int *x = NULL;
6.
7.     x = &a;
8.
9.     printf("Endereco de A:\t %p \n", &a);
10.    printf("X aponta para:\t %p \n", x);
11.    printf("Endereco de X:\t %p \n", &x);
12.    printf("Conteudo de X:\t %d \n", *x);
13.
14.    return 0;
15. }
```

Memória		
Endereço	Variável	Valor
...		
28FF18	int a	2
28FF1C	int *x	28FF18
...		

Endereço de x

Operadores & e *

Exemplo:

```
1. #include <stdio.h>
2. int main()
3. {
4.     int a = 2;
5.     int *x = NULL;
6.
7.     x = &a;
8.
9.     printf("Endereco de A:\t %p \n", &a);
10.    printf("X aponta para:\t %p \n", x);
11.    printf("Endereco de X:\t %p \n", &x);
12.    printf("Conteudo de X:\t %d \n", *x);
13.
14.    return 0;
15. }
```

Memória		
Endereço	Variável	Valor
...		
28FF18	int a	2
28FF1C	int *x	28FF18
...		

O valor da posição
em que x aponta

Atribuição de ponteiros

- ▶ Ao se trabalhar com ponteiros, existem dois tipos de atribuição: **atribuição de endereço** e **atribuição de conteúdo**.
- ▶ Na **atribuição de endereço**, um ponteiro só pode receber o endereço de memória de uma variável do mesmo tipo que o seu, ou receber outro ponteiro.
- ▶ Na **atribuição de conteúdo**, o ponteiro já aponta para uma variável em memória e o operador de atribuição é empregado para atribuir ou modificar o valor da variável em questão.

Atribuição de ponteiros

```
1. #include <stdio.h>
2. int main()
3. {
4.     int a = 2;
5.     int *p = NULL;
6.     int *q = NULL;
7.
8.     p = &a;
9.     q = p;
10.
11.     printf("Conteudo de Q:\t %d \n", *q);
12.
13.     return 0;
14. }
```

Atribuição de endereço

Atribuição de ponteiros

```
1. #include <stdio.h>
2. int main()
3. {
4.     int a = 2;
5.     int *p = NULL;
6.     int *q = NULL;
7.
8.     p = &a;
9.     q = p;
10.
11.     printf("Conteúdo de Q:\t %d \n", *q);
12.
13.     return 0;
14. }
```

p aponta para uma variável de seu tipo

q aponta para um ponteiro de seu tipo

O conteúdo de q é o mesmo que o de p = 2

Atribuição de ponteiros

```
1. #include <stdio.h>
2. int main()
3. {
4.     int a = 2;
5.     int *p = &a;
6.
7.     printf("Valor de A: %d \n", a);
8.
9.     *p = 3;
10.
11.    printf("Valor de A: %d \n", a);
12.
13.    return 0;
14. }
```

Atribuição de conteúdo

Atribuição de ponteiros

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

```
2. int main()
```

```
3. {
```

```
4.     int a = 2;
```

```
5.     int *p = &a;
```

p aponta para uma variável de seu tipo

```
6.
```

```
7.     printf("Valor de A: %d \n", a);
```

```
8.
```

```
9.     *p = 3;
```

O valor da variável para onde p aponta é alterado

```
10.
```

```
11.     printf("Valor de A: %d \n", a);
```

```
12.
```

```
13.     return 0;
```

```
14. }
```

Comparação entre ponteiros

- ▶ A linguagem C permite comparar o endereço de dois ponteiros utilizando uma expressão relacional.
- ▶ Os operadores `==` e `!=` são usados para saber se dois ponteiros apontam para o mesmo lugar ou se o conteúdo é o mesmo.
- ▶ Já os operadores relacionais (`<`, `<=`, `>=`, `>`) podem ser usados para saber se um ponteiro aponta para uma região mais adiante na memória que outro.
- ▶ No caso de operações aritméticas, quando usadas com o operador `*` na frente do ponteiro, a comparação é realizada com o seu conteúdo.

Comparação entre ponteiros

```
1. #include <stdio.h>
2. int main()
3. {
4.     int a = 2, b = 2;
5.     int *p = &a, *q = &b;
6.
7.     if (p == q) printf("Mesma posicao\n");
8.     else printf("Posicoes diferentes\n");
9.
10.    if (*p == *q) printf("Mesmo conteudo\n");
11.    else printf("Conteudos diferentes\n");
12.
13.    return 0;
14. }
```

Comparando posições
de memória

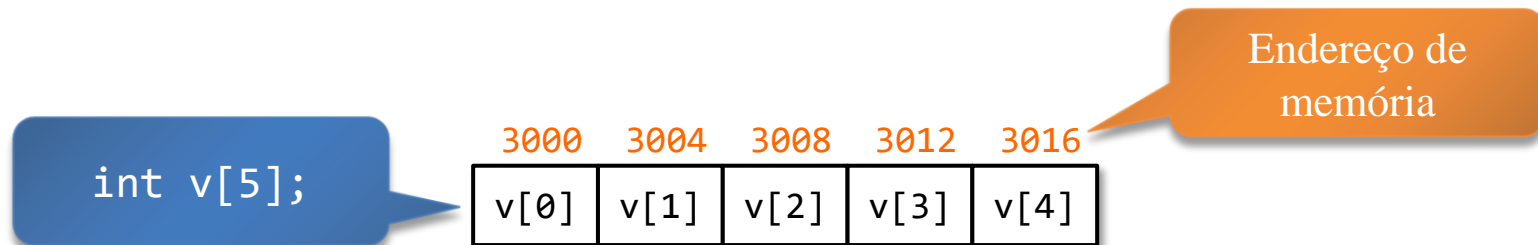
Comparando
conteúdo

Operações aritméticas sobre ponteiros

- ▶ O conjunto de operações aritméticas que podem ser realizadas sobre os endereços armazenados nos ponteiros limitado à duas operações:
 - ▶ Adição e subtração.
- ▶ **Inteiros** podem ser **adicionados** (+ ou +=) ou **subtraídos** (- ou -=) de ponteiros.
- ▶ Um ponteiro pode ser **subtraído** de outro ponteiro:
 - ▶ Somente quando ambos os ponteiros apontam para o mesmo *array*.

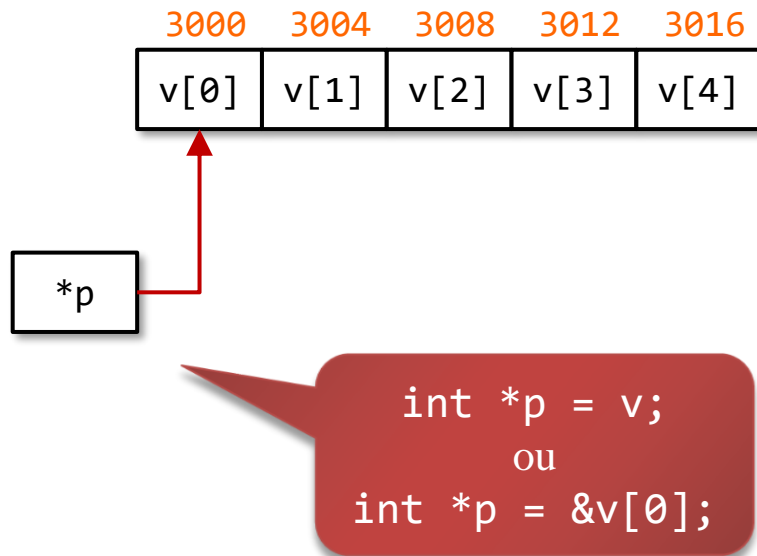
Operações aritméticas sobre ponteiros

- Incremento / decremento:



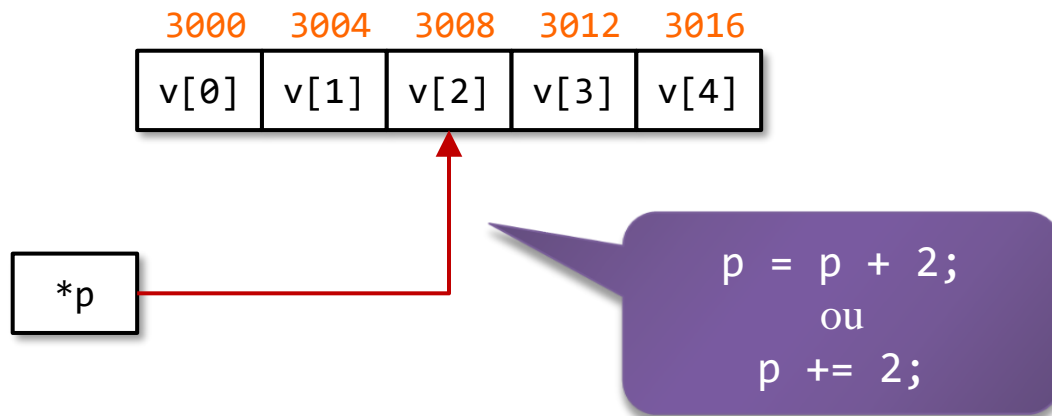
Operações aritméticas sobre ponteiros

- Incremento / decremento:



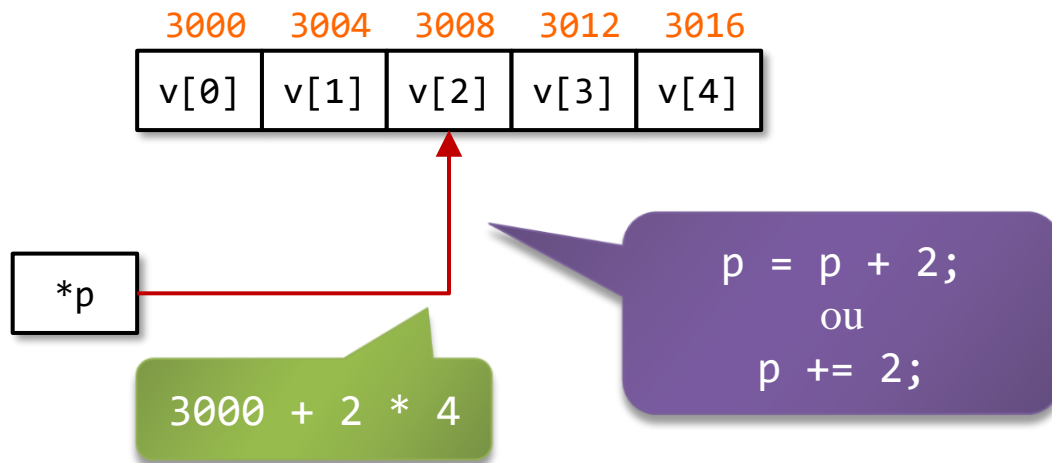
Operações aritméticas sobre ponteiros

- Incremento / decremento:



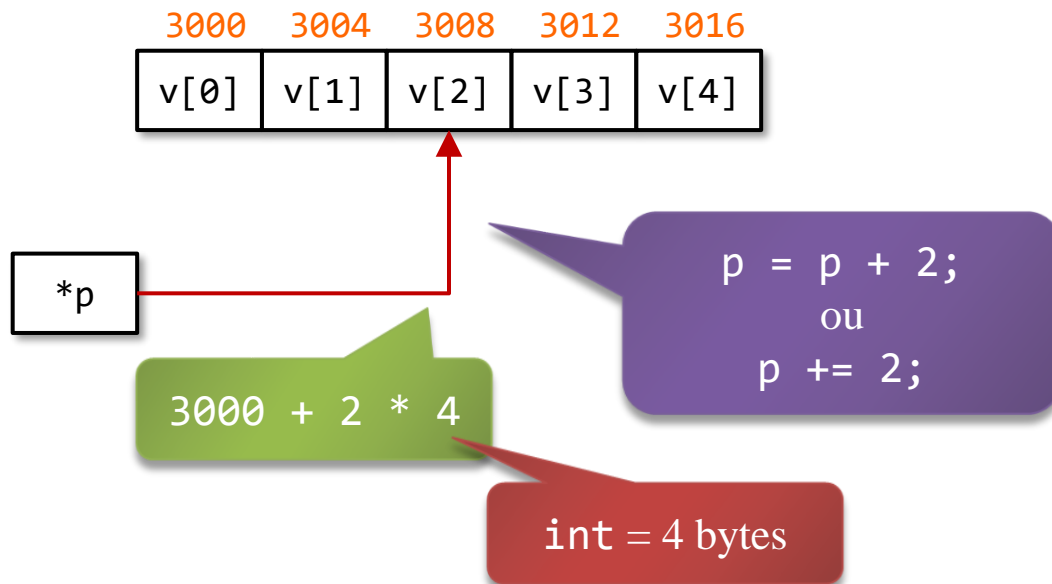
Operações aritméticas sobre ponteiros

- Incremento / decremento:



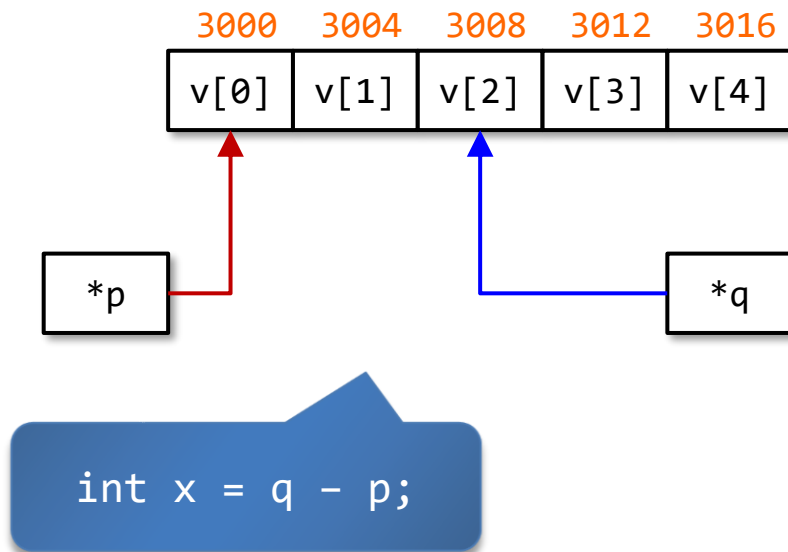
Operações aritméticas sobre ponteiros

- Incremento / decremento:



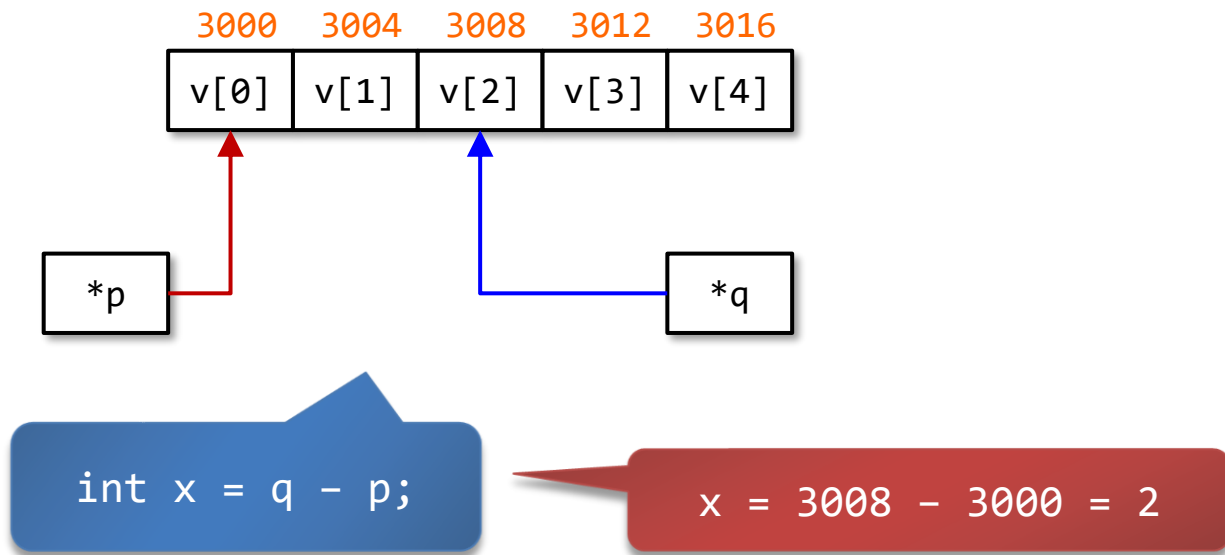
Operações aritméticas sobre ponteiros

- ▶ Subtração de ponteiros:



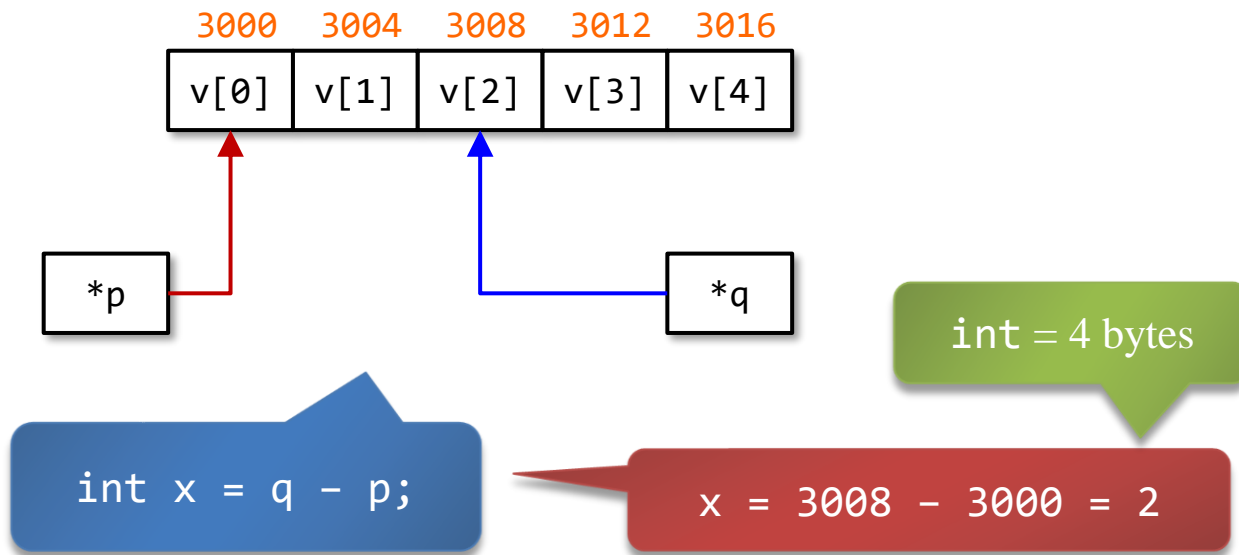
Operações aritméticas sobre ponteiros

- Subtração de ponteiros:



Operações aritméticas sobre ponteiros

- Subtração de ponteiros:



Operações aritméticas sobre ponteiros

Atenção!!!

A aritmética de ponteiros é indefinida
a menos que executada sobre um vetor.

Operações aritméticas sobre ponteiros

Atenção!!!

A aritmética de ponteiros é indefinida
a menos que executada sobre um vetor.

Por quê???

Operações aritméticas sobre ponteiros

Atenção!!!

A aritmética de ponteiros é indefinida a menos que executada sobre um vetor.

Por quê???

Não é possível garantir que duas variáveis estejam armazenadas sequencialmente na memória, exceto para o caso de vetores.

Alocação dinâmica de memória

- ▶ A linguagem C permite alocar **dinamicamente** blocos de memória utilizando ponteiros.
- ▶ A alocação dinâmica permite criar *arrays* de **qualquer tipo** em **tempo de execução**, ou seja, quando o programa está sendo executado, e não apenas quando se está escrevendo o programa.
- ▶ Utilizada quando não se sabe com exatidão o quanto de memória é necessário para armazenar os dados com os quais se deseja trabalhar.

Funções para alocação dinâmica

- ▶ A biblioteca padrão da linguagem C possui funções especialmente desenvolvidas para realizar a alocação dinâmica de memória.
- ▶ Estas funções estão definidas no arquivo de cabeçalho `stdlib.h`. As mais utilizadas são:
- ▶ `malloc`: aloca memória.
- ▶ `calloc`: aloca memória e inicializa o espaço alocado com zeros.
- ▶ `free`: devolve regiões de memória previamente alocadas ao sistema operacional (desaloca).
- ▶ Além do operador `sizeof`.

O operador `sizeof`

- ▶ Alocar memória para um elemento do tipo `int` é diferente de alocar memória para um elemento do tipo `double`.
- ▶ Isso ocorre porque **tipos diferentes têm tamanhos distintos** na memória.
- ▶ No momento da alocação, deve-se considerar o tamanho do dado alocado.
- ▶ O operador `sizeof` retorna o tamanho em bytes necessário para alocar um tipo de dado.

O operador sizeof

► Sintaxe: sizeof(**tipo**);

```
#include <stdio.h>
int main()
{
    printf("int = %d bytes \n", sizeof( int ) );
    printf("float = %d bytes \n", sizeof( float ) );
    printf("double = %d bytes \n", sizeof( double ) );
    printf("char = %d bytes \n", sizeof( char ) );
    printf("void = %d bytes \n", sizeof( void ) );
    printf("short int = %d bytes \n", sizeof( short int ) );
    printf("long int = %d bytes \n", sizeof( long int ) );
    printf("long long int = %d bytes \n", sizeof( long long int ) );
    printf("long double = %d bytes \n", sizeof( long double ) );
    return 0;
}
```

A função `malloc`

- ▶ Utilizada para alocar memória durante a execução do programa.
- ▶ Responsável por solicitar uma quantidade de memória ao sistema operacional e devolver um ponteiro para o início do espaço alocado.
- ▶ Sintaxe:

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

A função `malloc`

- ▶ A função `malloc` recebe um parâmetro de entrada:
 - ▶ `num`: o tamanho do espaço de memória a ser alocado.
- ▶ A função `malloc` pode retornar:
 - ▶ Um ponteiro para o início do bloco de memória alocado.
 - ▶ Um ponteiro nulo (NULL), no caso do sistema operacional se negar a fornecer memória ao programa.
- ▶ ***É IMPORTANTE SEMPRE TESTAR SE FOI POSSÍVEL REALIZAR A ALOCAÇÃO DE MEMÓRIA.***
- ▶ O bloco de memória alocado é manipulado como um *array*, e deve ser convertido, via *typecast*, para o tipo do ponteiro.

A função malloc

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. int main()
4. {
5.     int *p = NULL, i;
6.
7.     p = (int *)malloc( 5 * sizeof(int) );
8.
9.     if (!p)
10.    {
11.        printf("Memoria insuficiente...\n");
12.        exit(1);
13.    }
14.
15.    for (i = 0; i < 5; i++) scanf("%d", &p[i]);
16.
17.    return 0;
18. }
```


A função `malloc`

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. int main()
4. {
5.     int *p = NULL, i;
6.
7.     p = (int *)malloc( 5 * sizeof(int) );
8.
9.     if (!p)
10.    {
11.        printf("Memoria insuficiente...\n");
12.        exit(1);
13.    }
14.
15.    for (i = 0; i < 5; i++) scanf("%d", &p[i]);
16.
17.    return 0;
18. }
```

Alocando um array com
5 posições de `int`.

A função `malloc`

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. int main()
```

*Typecast para
ponteiro de int.*

```
4. {
5.     int *p = NULL, i;
```

Alocando um array com
5 posições de int.

```
6.     p = (int *)malloc( 5 * sizeof(int) );
```

```
7.
8.
```

```
9.     if (!p)
```

```
10.    {
```

```
11.        printf("Memoria insuficiente...\n");
```

```
12.        exit(1);
```

```
13.    }
```

```
14.
```

```
15.     for (i = 0; i < 5; i++) scanf("%d", &p[i]);
```

```
16.
```

```
17.     return 0;
```

```
18. }
```

A função `malloc`

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. int main()
```

*Typecast para
ponteiro de int.*

```
4. {
5.     int *p = NULL, i;
```

Alocando um array com
5 posições de int.

```
6.     p = (int *)malloc( 5 * sizeof(int) );
```

```
7.
8.
```

```
9.     if (!p)
```

Verificando se o sistema operacional
forneceu memória. Se não, encerra.

```
10.    {
```

```
11.        printf("Memoria insuficiente...\n");
```

```
12.        exit(1);
```

```
13.    }
```

```
14.
```

```
15.     for (i = 0; i < 5; i++) scanf("%d", &p[i]);
```

```
16.
```

```
17.     return 0;
```

```
18. }
```

A função `malloc`

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. int main()
```

*Typecast para
ponteiro de int.*

```
4. {
5.     int *p = NULL, i;
```

Alocando um array com
5 posições de int.

```
6.     p = (int *)malloc( 5 * sizeof(int) );
```

```
7.
8.
9.     if (!p)
```

Verificando se o sistema operacional
forneceu memória. Se não, encerra.

```
10. {
```

```
11.     printf("Memoria insuficiente...\n");
```

```
12.     exit(1);
```

```
13. }
```

```
14.
```

```
15.     for (i = 0; i < 5; i++) scanf("%d", &p[i]);
```

```
16.
```

```
17.     return 0;
```

```
18. }
```

p passa a ser tratado
como um *array*: `p[i]`.

A função `calloc`

- ▶ A função `calloc` é similar à função `malloc`, permitindo requisitar memória ao sistema operacional.
- ▶ A diferença entre elas está na sintaxe. `calloc` é definida como:

```
void* calloc(num, tam);
```

A função `calloc`

- ▶ A função `calloc` é similar à função `malloc`, permitindo requisitar memória ao sistema operacional.
- ▶ A diferença entre elas está na sintaxe. `calloc` é definida como:

```
void* calloc(num, tam);
```

Número de elementos
do array.

O tamanho em bytes
de cada elemento.

A função `calloc`

- ▶ A função `calloc` é similar à função `malloc`, permitindo requisitar memória ao sistema operacional.
- ▶ A diferença entre elas está na sintaxe. `calloc` é definida como:

```
void* calloc(num, tam);
```

Número de elementos
do array.

O tamanho em bytes
de cada elemento.

- ▶ `calloc` também inicializa os bits do espaço alocado com zeros.

A função `calloc`

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. int main()
4. {
5.     int *p = (int *)malloc( 5 * sizeof(int) );
6.     int *q = (int *)calloc( 5, sizeof(int) );
7.     int i;
8.
9.     if (!p || !q)
10.    {
11.        exit(1);
12.    }
13.
14.    for (i = 0; i < 5; i++)
15.        printf("%d: %d \t %d \n", i, p[i], q[i]);
16.
17.    return 0;
18. }
```


A função `free`

- ▶ Sempre que as funções `malloc` ou `calloc` são utilizadas para se alocar memória, a memória alocada **deve ser explicitamente liberada pelo programador**.
- ▶ Se o programador não realizar a “desalocação”, a memória permanece associada ao programa e não é devolvida ao sistema operacional.
- ▶ Programas que não liberam memória levam a erros conhecidos por **vazamentos de memória** (*memory leak*), o que pode, após algum tempo consumir toda a memória do computador.
- ▶ A função `free` permite devolver memória ao sistema operacional.

A função `free`

- ▶ Sintaxe: `free(ponteiro);`

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. int main()
4. {
5.     int *p = (int *)malloc( 5 * sizeof(int) );
6.
7.     // utilização de p...
8.
9.
10.    free(p); // liberação de memória
11.           // antes de encerrar o programa
12.    return 0;
13. }
```

Alocação dinâmica de estruturas

- ▶ De modo análogo aos tipos primitivos da linguagem C, também é possível realizar a **alocação dinâmica de estruturas**.
- ▶ As funções empregadas são exatamente as mesmas para a alocação de uma struct.
- ▶ Pode-se realizar a alocação de uma **única struct** ou de um **array de struct**.

Alocação de estruturas

- ▶ Para alocar uma **única struct**:
 - ▶ Um ponteiro para struct receberá a função **malloc()**.
 - ▶ O **operador seta** é empregado para acessar os membros da struct.
 - ▶ A função **free()** libera a memória.

Alocação de estruturas

- ▶ Para alocar uma **única struct**:

- ▶ Um ponteiro para struct receberá a função **malloc()**.
- ▶ O **operador seta** é empregado para acessar os membros da struct.
- ▶ A função **free()** libera a memória.

```
1. typedef struct cadastro
2. {
3.     char nome[50];
4.     int idade;
5. }cadastro;
6. int main()
7. {
8.     cadastro *c = (cadastro*)malloc(sizeof(cadastro));
9.     strcpy(c->nome, "Maria");
10.    c->idade = 25;
11.    free(c);
12.    return 0;
13. }
```

Alocação de arrays de estruturas

- ▶ Para alocar um **array de struct**:
 - ▶ Um ponteiro para struct receberá a função **malloc()**.
 - ▶ O **operador ponto** é empregado para acessar os membros da struct.
 - ▶ O **operador colchetes** é empregado para acessar a posição do array.
 - ▶ A função **free()** libera a memória.

Alocação de arrays de estruturas

▶ Para alocar um array de struct:

- ▶ Um ponteiro para struct receberá a função `malloc()`.
- ▶ O operador ponto é empregado para acessar os membros da struct.
- ▶ O operador colchetes é empregado para acessar a posição do array.
- ▶ A função `free()` libera a memória.

```
1. typedef struct cadastro...
2. int main()
3. {
4.     cadastro *c = (cadastro*)malloc(10 * sizeof(cadastro));
5.     strcpy(c[0].nome, "Maria");
6.     c[0].idade = 25;
7.     strcpy(c[1].nome, "Pedro");
8.     c[1].idade = 27;
9.     free(c);
10.    return 0;
11. }
```

Alocação dinâmica de matrizes

- ▶ Para alocação de *arrays* com mais de uma dimensão, é necessário utilizar o conceito de **ponteiro de ponteiro**.
- ▶ Para cada dimensão extra, declara-se a variável possuindo um asterisco a mais.

Alocação dinâmica de matrizes

- ▶ Para alocação de *arrays* com mais de uma dimensão, é necessário utilizar o conceito de **ponteiro de ponteiro**.
- ▶ Para cada dimensão extra, declara-se a variável possuindo um asterisco a mais.
- ▶ Para um *array* bidimensional (matriz), tem-se:

```
tipo **matriz;
```

Alocação dinâmica de matrizes

- ▶ Utilizando um ponteiro para inteiros com dois níveis (`int **p`), aloca-se no primeiro nível do ponteiro um *array de ponteiros* representando as *linhas* da matriz.
- ▶ Essa tarefa é realizada alocando um *array* usando o tamanho de um ponteiro de `int`, ou seja, `sizeof(int *)`:

Alocação dinâmica de matrizes

- ▶ Utilizando um ponteiro para inteiros com dois níveis (`int **p`), aloca-se no primeiro nível do ponteiro um *array de ponteiros* representando as *linhas* da matriz.
- ▶ Essa tarefa é realizada alocando um *array* usando o tamanho de um ponteiro de `int`, ou seja, `sizeof(int *)`:

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

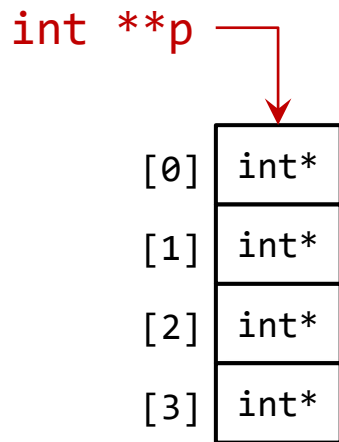
Alocação dinâmica de matrizes

- ▶ A seguir, para cada posição do *array* de ponteiros aloca-se um *array de inteiros*, representando as *colunas* da matriz, as quais efetivamente conterão os dados.
- ▶ Essa tarefa é realizada dentro de um laço *for*, alocando um *array* usando o tamanho de *int*, isto é, `sizeof(int)`:

```
for (i = 0; i < linhas; i++)  
{  
    p[i] = (int *)malloc(colunas * sizeof(int));  
}
```

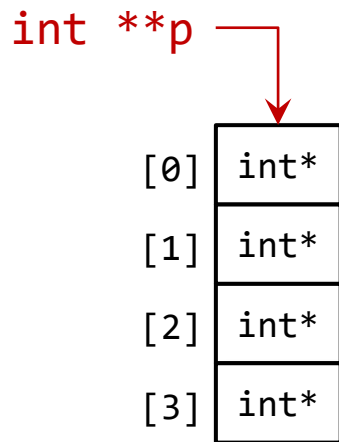
Alocação dinâmica de matrizes

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



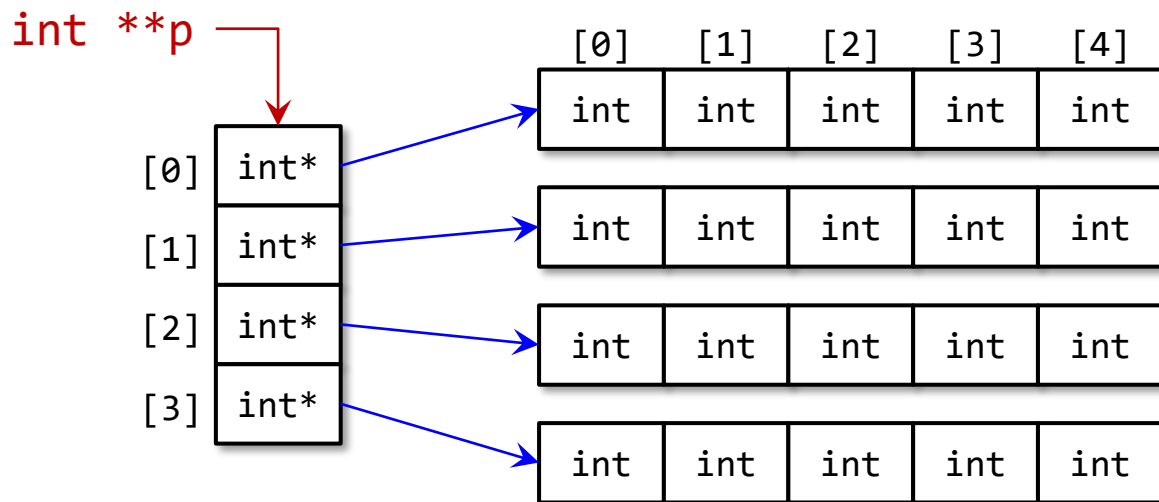
Alocação dinâmica de matrizes

```
int **p = (int **)malloc(4 * sizeof(int *));  
for (i = 0; i < 4; i++)  
{  
    p[i] = (int *)malloc(5 * sizeof(int));  
}
```



Alocação dinâmica de matrizes

```
int **p = (int **)malloc(4 * sizeof(int *));  
for (i = 0; i < 4; i++)  
{  
    p[i] = (int *)malloc(5 * sizeof(int));  
}
```



Alocação dinâmica de matrizes

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #define L 4 // linhas
4. #define C 5 // colunas
5. int main()
6. {
7.     int i;
8.     int **p = (int **)malloc( L * sizeof(int *) );
9.     for (i = 0; i < L; i++)
10.         p[i] = (int *)malloc( C * sizeof(int) );
11.
12.     // utilização de p como uma matriz,
13.     // referenciando como p[x][y]
14.
15.     for (i = 0; i < L; i++) // desaloca colunas
16.         free(p[i]);
17.
18.     free(p); // desaloca linhas
19.
20.     return 0;
21. }
```


Alocação dinâmica de matrizes

- ▶ A solução anterior é **computacionalmente ruim** por algumas razões:
- ▶ O mecanismo de alocação **consome tempo** para realizar **cada alocação** (linhas e colunas).
- ▶ O mecanismo de alocação **acrescenta bytes** extras entre as linhas para manter a contagem. Estes bytes extras **consomem espaço**...
- ▶ O mecanismo de “desalocação” também consome **tempo extra** para **liberar memória**.
- ▶ Cada alocação é realizada em uma posição **descontínua** de memória.
- ▶ Para percorrer toda a matriz, o programa faz uma série de **saltos** por **entre** essas **posições**.

Alocação dinâmica de matrizes

- ▶ Para evitar esses problemas, **aloca-se um único bloco de memória** e rearranja-se os ponteiros, de modo a formar uma matriz.

Alocação dinâmica de matrizes

- ▶ Para evitar esses problemas, **aloca-se um único bloco de memória** e rearranja-se os ponteiros, de modo a formar uma matriz.

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

p[0] = (int *)malloc(linhas * colunas * sizeof(int));

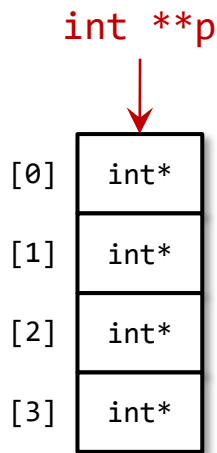
for (i = 1; i < linhas; i++)
{
    p[i] = p[i - 1] + colunas;
}

// uso de p como uma matriz: p[x][y]

free(p[0]);
free(p);
```

Alocação dinâmica de matrizes

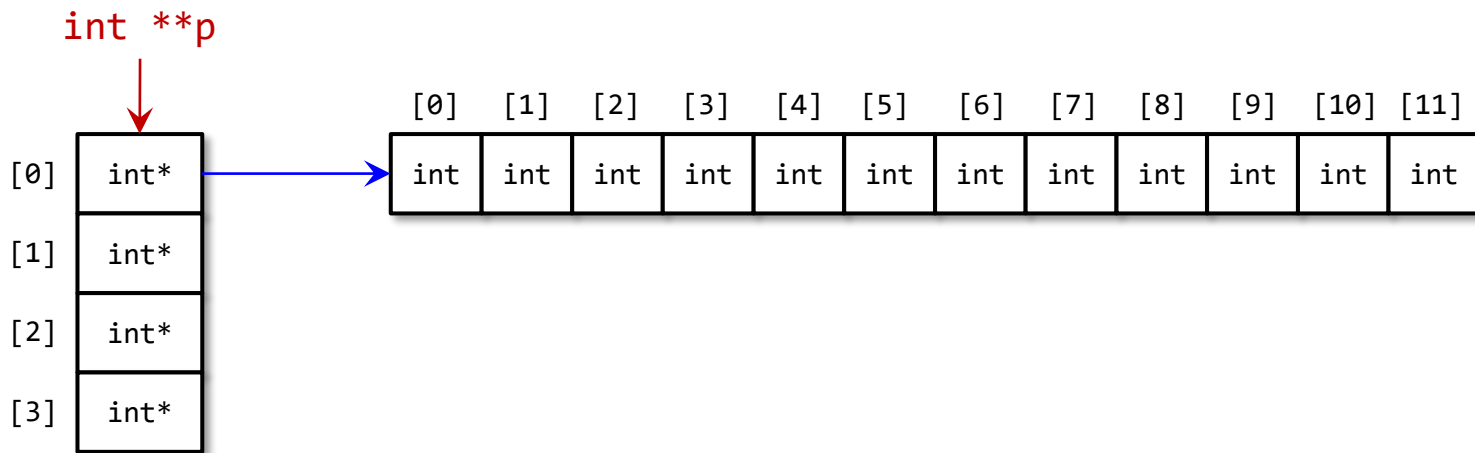
```
int **p = (int **)malloc(4 * sizeof(int *));
```



Alocação dinâmica de matrizes

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

```
p[0] = (int *)malloc(4 * 3 * sizeof(int));
```

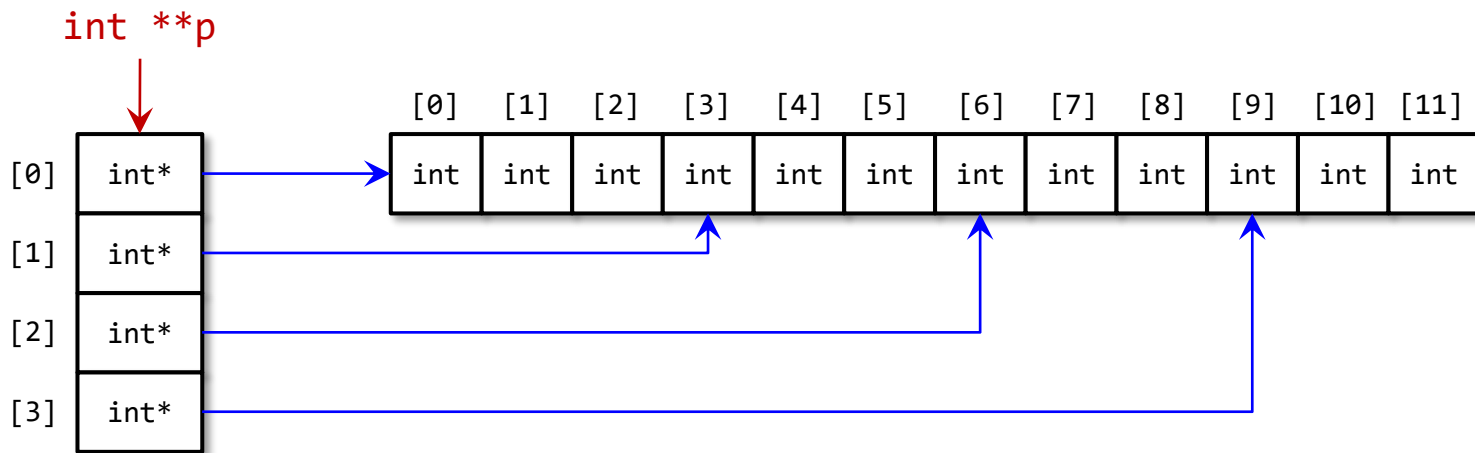


Alocação dinâmica de matrizes

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

```
p[0] = (int *)malloc(4 * 3 * sizeof(int));
```

```
for (i = 1; i < 4; i++)  
    p[i] = p[i - 1] + 3;
```



Dúvidas?



Aula 11:

Ponteiros e Alocação Dinâmica de Memória

Disciplina: Fundamentos de Programação

Prof. Luiz Olmes

olmes@unifei.edu.br

