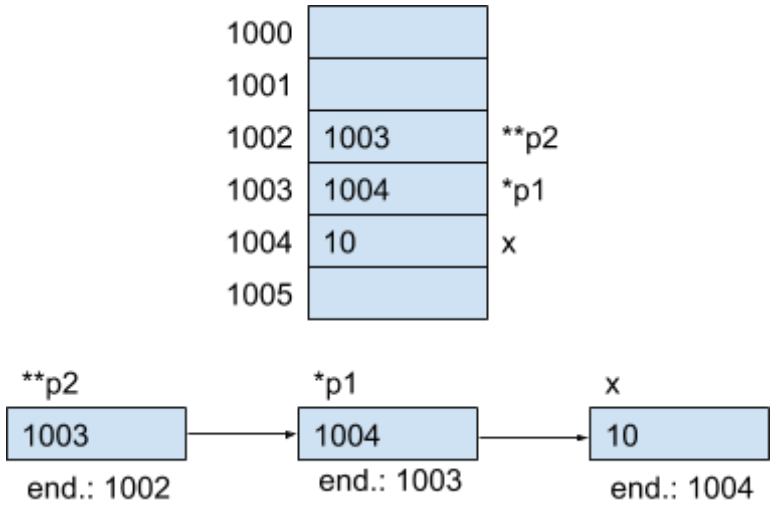
Notas de aula 01/04

Ponteiros para ponteiros:

Se *p1 é um ponteiro, então **p2 é um ponteiro para ponteiro. Considere o código abaixo:

```c
#include <stdio.h>

main() {
      int *p1, **p2, x;
      x = 10;
      p1 = &x;
      p2 = &p1;
      printf("End. de x: %ld\n", (long int)&x);
      printf("p1 guarda: %ld\n", (long int)p1);
      printf("Conteudo apontado por p1: %d\n", *p1);
      printf("\nEnd. de p1: %ld\n", (long int)&p1);
      printf("p2 guarda: %ld\n", (long int)p2);
      printf("Endereço guardado por p1: %ld\n", (long int)*p2);
      printf("Conteudo apontado por p1: %d\n", **p2);
}
```

Exemplo:



- Ao acessar p2, tenho 1003 (endereço apontado por p2);
- Ao acessar *p2, tenho 1004 (o endereço apontado por p1);
- Ao acessar **p2, tenho 10 (o valor guardado no endereço apontado por p1);
  Cada * faz andar um pouquinho mais no encadeamento.

Cuidados:

Se p2 aponta para p1, p1 aponta para x e x teve o valor alterado, a mudança em x reflete em p2:

```c
#include <stdio.h>

main() {
	int *p1, **p2, x = 10;
	p1 = &x;
	p2 = &p1;
	printf("%d\n", **p2);
	x = 20;
	printf("%d\n", **p2);
}
```

Se p2 aponta para p1, p1 aponta para x e x teve o valor alterado a partir do ponteiro p2, a alteração também reflete em p1 e x:

```c
#include <stdio.h>

main() {
	int *p1, **p2, x = 10;
	p1 = &x;
	p2 = &p1;
	printf("%d\n", *p1);
	**p2 = 20;
	printf("%d\n", *p1);
	printf("%d\n", x);
}
```

Se p2 é ponteiro para p1 e p1 for liberado, o que acontece com p2? Ele ainda aponta para o endereço que era de p1, mas o conteúdo de p1 é perdido:

```c
#include <stdio.h>

main() {
	int *p1, **p2;
	p1 = malloc(sizeof(int));
	*p1 = 10;
	p2 = &p1;
	printf("%ld\n", (long int)p2);
	printf("%d\n", **p2);
	free(p1);
	printf("%ld\n", (long int)p2);
	printf("%d\n", **p2);
}
```

Tratando-se de estruturas, o cuidado deve ser ainda maior. Imagine que temos uma estrutura chamada Ponto.

- “ponto” é do tipo Ponto.
- “*p1” é um ponteiro para um tipo Ponto;
- “**p2” é um ponteiro para um ponteiro para um tipo Ponto;
- “***p3” é um ponteiro para um ponteiro para um ponteiro para um tipo Ponto.

```c
#include <stdio.h>

struct ponto{
	int x;
	int y;
};
typedef struct ponto Ponto;

main(){
	Ponto ponto, *p1, **p2, ***p3;

	ponto.x = 10;
	ponto.y = 20;
	p1 = &ponto;
	p2 = &p1;
	p3 = &p2;

	printf("A partir de *p:\n");
	printf("x: %d\n", (*p1).x);
	printf("y: %d\n", (*p1).y);
	// ou:
	printf("x: %d\n", p1->x);
	printf("y: %d\n", p1->y);
	/*
	Dentro dos () é acessado o conteúdo de p1 através de uma op. de deferência.
     O conteúdo de p1 é um tipo Ponto, é possível então acessar seus campos.
     */

	printf("A partir de **p2:\n");
	printf("x: %d\n", (*(*p2)).x);
	printf("x: %d\n", (*(*p2)).y);
	// ou
	printf("x: %d\n", (*p2)->x);
	printf("x: %d\n", (*p2)->y);
	/*
	Dentro dos () mais internos, é acessado o conteúdo de p2 através de uma op. de deferência.
```

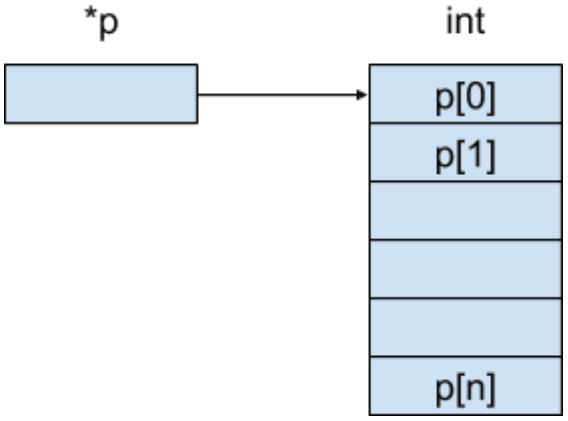
```
      O conteúdo de p2 é o endereço de p1, é preciso fazer mais uma op.
de deferência.
      O conteúdo de p1 é um tipo Ponto, é possível então acessar seus
campos.
      */

      printf("A partir de ***p3:\n");
      printf("x: %d\n", (*(*(*p3))).x);
      printf("x: %d\n", (*(*(*p3))).y);
      // ou
      printf("x: %d\n", (*(*p3))->x);
      printf("x: %d\n", (*(*p3))->y);
      /*
      Dentro dos () mais internos, é acessado o conteúdo de p3 através
de uma operação de deferência.
      O conteúdo de p3 é o endereço de p2, é preciso fazer mais uma op.
de deferência.
      O conteúdo de p2 é o endereço de p1, é preciso fazer mais uma op.
de deferência.
      O conteúdo de p1 é um tipo Ponto, é possível então acessar seus
campos.
      */
}
```

Alocação dinâmica de matrizes:

Considerando vetores, vimos que:

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

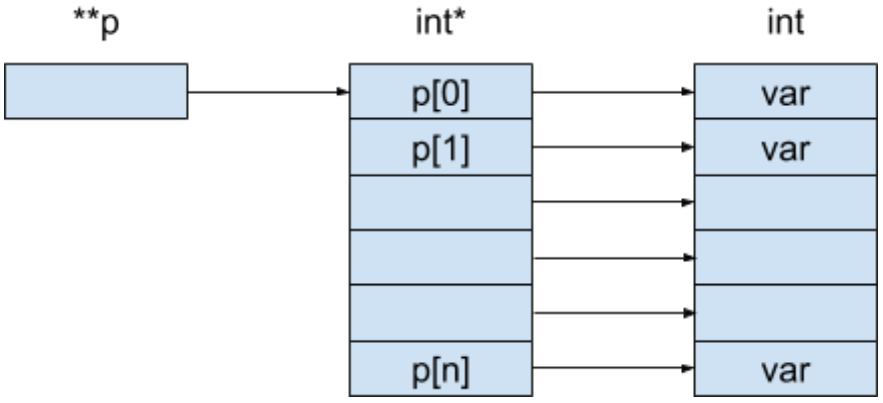
cria um vetor de int. Ou seja, *p aponta para o primeiro elemento de um vetor de int:



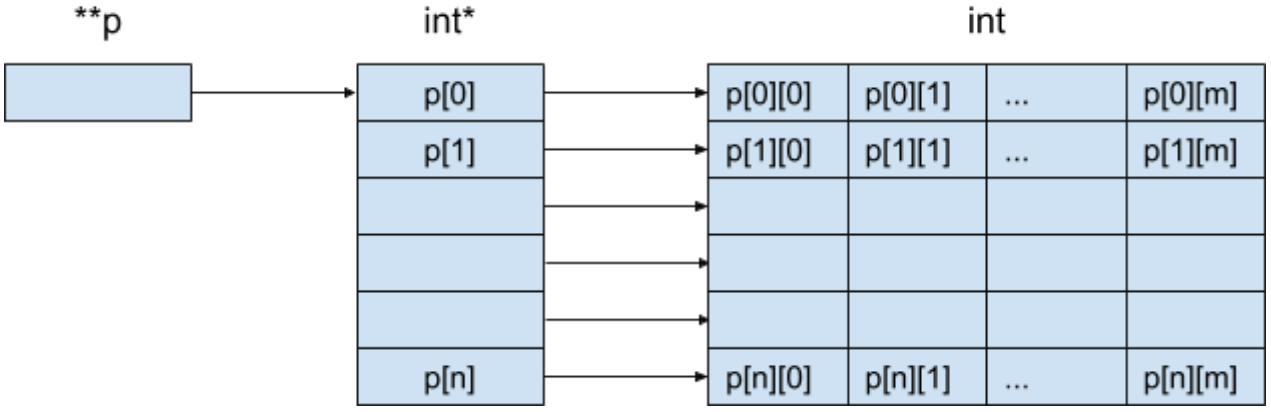
O que dizer se tivermos:

```
int **p;
```

Se *p for um ponteiro para um vetor de int, **p é um ponteiro para um vetor de int*, ou seja, um vetor de ponteiros para int, onde cada um desses ponteiros aponta para alguma variável do tipo int:



Sabemos que uma matriz é um vetor de vetores. Se fizermos cada ponteiro do vetor int* acima apontar para um vetor de inteiros em vez de apontar para uma única variável, temos uma matriz alocada dinamicamente.



Pensando em matrizes, o vetor tipo int*, guarda um ponteiro para cada linha da matriz.

Alocando uma matriz dinamicamente em C

```c
// matriz de inteiros alocada dinamicamente
#include <stdio.h>

main() {
      int **m, l, c, n_linhas, n_colunas;

      // matriz 4 x 5
      n_linhas = 4;
      n_colunas = 5;

      // 1 - alocamos um vetor com um ponteiro para cada linha:
```

```
        m = malloc(n_linhas * sizeof(int*));
        // 2 - fazemos cada ponteiro apontar para uma linha:
        for(l = 0; l < n_linhas; l++)
            m[l] = malloc(n_colunas * sizeof(int));

        // exibindo os endereços que foram alocados:
        for(l = 0; l < n_linhas; l++) {
            for(c = 0; c < n_colunas; c++)
                printf("%ld\t", (long int)&m[l][c]);
            printf("\n");
        }
}
```

Obs.: se eu quisesse garantir que a matriz estaria toda preenchida com 0s, poderia usar calloc.

Analise os endereços alocados. Perceba que os itens de uma linha sempre são alocados em espaços contíguos de memória. Já as linhas, nem sempre estarão ocupando blocos contíguos de memória. Cada linha pode estar alocada em um bloco bem distante um do outro, diferente de uma matriz alocada estaticamente.

Liberando uma matriz alocada dinamicamente:

Relembrando o que fizemos para alocar a matriz:

1. Alocamos um vetor de ponteiros, contendo um ponteiro para cada linha da matriz;
2. Fazemos cada um desses ponteiros apontar para um vetor (linha da matriz).

O processo de liberar memória deve então ser o inverso:

1. Liberar cada linha;
2. Liberar os ponteiros que apontam para essas linhas.

Podemos fazer uma analogia com uma caixa de livros. Imagine que você quer guardar seus livros em uma caixa:

1. Você pega uma caixa;
2. Você coloca os livros dentro da caixa;

Imagine agora que você instalou uma prateleira e quer jogar a caixa fora:

1. Você remove todos os livros de dentro da caixa;
2. Agora que a caixa está vazia, você se livra dela.

Implementado isto em C, temos:

```
for(l = 0; l < n_linhas; l++) {
    free(m[l]); // tirar os livros da caixa
}
free(m); // livrar-se da caixa
```

Matrizes dinâmicas e funções:

```c
// matriz de inteiros alocada dinamicamente
#include <stdio.h>
#include <stdlib.h>

int ** aloca_matriz(n_linhas, n_colunas) {
    int **m, l, c;

    // 1 - alocamos um vetor com um ponteiro para cada linha:
    m = malloc(n_linhas * sizeof(int*));
    // 2 - fazemos cada um ponteiro apontar para uma linha:
    for(l = 0; l < n_linhas; l++)
        m[l] = malloc(n_colunas * sizeof(int));

    return m;
}

void imprime(int **m, int n_linhas, int n_colunas) {
    int l, c;
    // exibindo os endereços que foram alocados:
    for(l = 0; l < n_linhas; l++) {
        for(c = 0; c < n_colunas; c++)
            printf("%ld\t", (long int)&m[l][c]);
        printf("\n");
    }
}

void libera_matriz(int **m, int n_linhas) {
    int l, c;
    for(l = 0; l < n_linhas; l++) {
        free(m[l]);
    }
    free(m);
}

main() {
    int l, c, n_linhas, n_colunas, **m;

    n_linhas = 4;
    n_colunas = 5;
    m = aloca_matriz(n_linhas, n_colunas);
    imprime(m, n_linhas, n_colunas);
    libera_matriz(m, n_linhas);
}
```

Vantagens de usar matriz alocada dinamicamente:

- Se o código for extenso e a matriz for usada somente no começo, assim que se termina de usá-la, a memória ocupada por ela pode ser liberada.
- Se cada linha é alocada dinamicamente, é possível até mesmo ter uma "matriz" onde cada linha possui um tamanho diferente, otimizando espaço.

Aplicação:

Exercício TAD livro - ver código no github.

Links interessantes:

Vídeo aula sobre alocação dinâmica de matriz do prof. André Backes: https://youtu.be/W4vbwEJn11U