

Alocação Dinâmica

Prof. Bruno Travençolo
Slides cedidos pelo Prof. André Backes

Definição

- ▶ Sempre que escrevemos um programa, é preciso reservar espaço para as informações que serão processadas.
- ▶ Para isso utilizamos as variáveis
 - ▶ Uma variável é uma posição de memória que armazena uma informação que pode ser modificada pelo programa.
 - ▶ Ela deve ser definida antes de ser usada.



Definição

- ▶ Infelizmente, nem sempre é possível saber o quanto de memória um programa irá precisar.
- ▶ Não é possível saber, em tempo de execução, o quanto de memória é necessário para executar um programa



Motivação

- ▶ Faça um programa para cadastrar o preço de 100 produtos

```
double produtos[100];
```

```
int i;
```

```
for (i = 0; i < 100; i++){  
    printf("Informe o valor do produto %d R$:", i+1);  
    scanf("%lf", &produtos[i]);  
}
```

```
printf("\nProdutos cadastrados\n");  
for (i = 0; i < 100; i++){  
    printf("Produto %d - R$: %f\n", i+1, produtos[i]);  
}
```




Motivação

- ▶ Faça um programa para cadastrar o preço de **N produtos**, em que N é um valor informado pelo usuário

```
int N,i;
```

```
double produtos[N];
```



Errado*! Em tempo de compilação não sabemos o valor de N

```
for (i = 0; i < N; i++){  
    printf("Informe o valor do produto %d R$:",i+1);  
    scanf("%lf", &produtos[i]);  
}  
  
printf("\nProdutos cadastrados\n");  
for (i = 0; i < N; i++){  
    printf("Produto %d - R$: %f\n",i+1, produtos[i]);  
}
```

- ▶ *obs: na verdade a sintaxe é válida a partir do C99. Mas não vamos usá-la neste curso.

Definição

- ▶ Processo de alocar memória para um programa em tempo de execução
- ▶ Quantidade de memória é alocada sob demanda, ou seja, quando o programa precisa
 - ▶ Menos desperdício de memória
- ▶ Espaço é reservado até liberação explícita
 - ▶ Depois de liberado, estará disponibilizado para outros usos e não pode mais ser acessado
 - ▶ Espaço alocado e não liberado explicitamente é automaticamente liberado ao final da execução



Alocação Dinâmica

- ▶ A linguagem C ANSI usa apenas 4 funções para o sistema de alocação dinâmica, disponíveis na `stdlib.h`:

<code>malloc()</code>	Alocar memória
<code>calloc()</code>	
<code>realloc()</code>	Realocar memória
<code>free()</code>	Liberar memória



Alocação Dinâmica - malloc

▶ malloc

- ▶ A função malloc() serve para alocar memória e tem o seguinte protótipo:

void *malloc (unsigned int num);

▶ Funcionalidade

- ▶ Dado o número de bytes que queremos alocar (**num**), ela aloca na memória e retorna um ponteiro **void*** para o primeiro byte alocado.
- ▶ O ponteiro **void*** pode ser atribuído a qualquer tipo de ponteiro via *type cast*. Se não houver memória suficiente para alocar a memória requisitada a função malloc() retorna um ponteiro nulo (NULL).



Alocação Dinâmica - malloc

- ▶ Alocar 1000 bytes de memória livre.

```
char *p;  
p = (char *) malloc(1000);
```

- ▶ Alocar espaço para 50 inteiros:

```
int *p;  
p = (int *) malloc(50*sizeof(int));
```



Operador **sizeof**

- ▶ Traduzindo: *sizeof*: size (tamanho) of (de)
 - ▶ Retorna o tamanho em bytes ocupado por objetos ou tipos
 - ▶ Exemplo de uso
 - ▶ `printf("\nTamanho em bytes de um char: %u", sizeof(char));`
 - ▶ Retorna 1, pois o tipo char tem 1 byte
 - ▶ `printf("\nTamanho em bytes de um char: %u", sizeof char);`
 - ▶ Também funciona sem o parênteses

Retorna um tipo `size_t`, normalmente `unsigned int`, por isso o `%u` ao invés de `%d`

`unsigned int` - é um número inteiro sem sinal negativo



Operador sizeof

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

```
int main()
{
```

```
    // descobrindo o tamanho ocupado por diferentes tipos de dados
    printf("\nTamanho em bytes de um char: %u", sizeof(char));
    printf("\nTamanho em bytes de um inteiro: %u", sizeof(int));
    printf("\nTamanho em bytes de um float: %u", sizeof(float));
    printf("\nTamanho em bytes de um double: %u", sizeof(double));
```

```
    // descobrindo o tamanho ocupado por uma variável
```

```
    int Numero_de_Alunos;
    printf("\nTamanho em bytes de Numero_de_Alunos (int): %u", sizeof Numero_de_Alunos );
```

```
    // também é possível obter o tamanho de vetores
```

```
    char nome[40];
    printf("\nTamanho em bytes de nome[40]: %u", sizeof(nome));
```

```
    double notas[60];
    printf("\nTamanho em bytes de notas[60]: %u", sizeof notas );
```

```
    return 0;
```

```
}
```

```
D:\Dropbox\Aulas\2014-01\ipc\projetos\memoria\sizeofdemo\bin\Debug\sizeofd
Tamanho em bytes de um char: 1
Tamanho em bytes de um inteiro: 4
Tamanho em bytes de um float: 4
Tamanho em bytes de um double: 8
Tamanho em bytes de Numero_de_Alunos (int): 4
Tamanho em bytes de nome[40]: 40
Tamanho em bytes de notas[60]: 480
Process returned 0 (0x0) execution time : 1.519 s
Press any key to continue.
```

Alocando memória

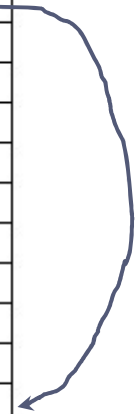
```
char *nome;  
nome = (char *) malloc(5*sizeof(char));
```

67			
68			
69		nome	char *
70	lx		
71			
72			
73			
74			
75			
76			
77			
78			
79			
80			
81			
82			
83			
84			

**Alocando 5
posições de
memória em
char * n**



67			
68			
69		nome	char *
70	79		
71			
72			
73			
74			
75			
76			
77			
78			
79		nome[0]	char
80		nome[1]	char
81		nome[2]	char
82		nome[3]	char
83		nome[4]	char
84			



Alocação Dinâmica - malloc

- ▶ Se não houver memória suficiente para alocar a memória requisitada, a função malloc() retorna um ponteiro nulo

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  int main(){
04      int *p;
05      p = (int *) malloc(5*sizeof(int));
06      if(p == NULL){
07          printf("Erro: Memoria Insuficiente!\n");
08          system("pause");
09          exit(1);
10      }
11      int i;
12      for (i=0; i<5; i++){
13          printf("Digite o valor da posicao %d: ",i);
14          scanf("%d",&p[i]);
15      }
16      system("pause");
17      return 0;
18  }
```

Alocação Dinâmica - calloc

- ▶ **calloc**

- ▶ A função calloc() também serve para alocar memória, mas possui um protótipo um pouco diferente:

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

- ▶ **Funcionalidade**

- ▶ Basicamente, a função calloc() faz o mesmo que a função malloc(). A diferença é que agora passamos a quantidade de posições a serem alocadas e o tamanho do tipo de dado alocado como parâmetros distintos da função.
 - ▶ A função inicializa em zero o vetor alocado



Alocação Dinâmica - calloc

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  int main(){
04      //alocação com malloc
05      int *p;
06      p = (int *) malloc(50*sizeof(int));
07      if(p == NULL){
08          printf("Erro: Memoria Insuficiente!\n");
09      }
10      //alocação com calloc
11      int *p1;
12      p1 = (int *) calloc(50,sizeof(int));
13      if(p1 == NULL){
14          printf("Erro: Memoria Insuficiente!\n");
15      }
16      system("pause");
17      return 0;
18  }
```



Alocação Dinâmica - realloc

- ▶ **realloc**

- ▶ A função `realloc()` serve para realocar memória e tem o seguinte protótipo:

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

- ▶ **Funcionalidade**

- ▶ A função modifica o tamanho da memória previamente alocada e apontada por ***ptr** para aquele especificado por **num**.
 - ▶ O valor de **num** pode ser maior ou menor que o original.



Alocação Dinâmica - realloc

► realloc

- Um ponteiro para o bloco é devolvido porque realloc() pode precisar mover o bloco para aumentar seu tamanho.
- Se isso ocorrer, o conteúdo do bloco antigo é copiado para o novo bloco, e nenhuma informação é perdida.

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  int main(){
04      int i;
05      int *p = malloc(5*sizeof(int));
06      for (i = 0; i < 5; i++){
07          p[i] = i+1;
08      }
09      for (i = 0; i < 5; i++){
10          printf("%d\n",p[i]);
11      }
12      printf("\n");
13      //Diminui o tamanho do array
14      p = realloc(p,3*sizeof(int));
15      for (i = 0; i < 3; i++){
16          printf("%d\n",p[i]);
17      }
18      printf("\n");
19      //Aumenta o tamanho do array
20      p = realloc(p,10*sizeof(int));
21      for (i = 0; i < 10; i++){
22          printf("%d\n",p[i]);
23      }
24      system("pause");
25      return 0;
26  }
```

Alocação Dinâmica - realloc

- ▶ Observações sobre realloc()
 - ▶ Se ***ptr** for nulo, aloca **num** bytes e devolve um ponteiro (igual malloc);
 - ▶ se **num** é zero, a memória apontada por ***ptr** é liberada (igual free).
 - ▶ Se não houver memória suficiente para a alocação, um ponteiro nulo é devolvido e o bloco original é deixado inalterado.



Alocação Dinâmica - free

▶ free

- ▶ Diferente das variáveis definidas durante a escrita do programa, as variáveis alocadas dinamicamente não são liberadas automaticamente pelo programa após a finalização do seu escopo.
 - ▶ Obs: Quando o programa finaliza toda sua memória ocupada é liberada
- ▶ Quando alocamos memória dinamicamente é necessário que nós a liberemos quando ela não for mais necessária.
- ▶ Para isto existe a função `free()` cujo protótipo é:
`void free (void *p);`



Alocação Dinâmica - free

- ▶ Assim, para liberar a memória, basta passar como parâmetro para a função `free()` o ponteiro que aponta para o início da memória a ser desalocada.
- ▶ Como o programa sabe quantos bytes devem ser liberados?
 - ▶ Quando se aloca a memória, o programa guarda o número de bytes alocados numa "tabela de alocação" interna.



Alocação Dinâmica - free

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  int main(){
04      int *p,i;
05      p = (int *) malloc(50*sizeof(int));
06      if(p == NULL){
07          printf("Erro: Memoria Insuficiente!\n");
08          system("pause");
09          exit(1);
10      }
11      for (i = 0; i < 50; i++){
12          p[i] = i+1;
13      }
14      for (i = 0; i < 50; i++){
15          printf("%d\n",p[i]);
16      }
17      //libera a memória alocada
18      free(p);
19      system("pause");
20      return 0;
21  }
```



Exemplo 1

- ▶ Podemos alocar memória para um único dado de um determinado tipo (tipo simples)

```
int *p;  
p = (int *) malloc(sizeof(int));  
  
*p = 10;  
  
printf("Valor: %d", *p);  
  
free(p);
```



Memory leak (vazamento de memória)

- ▶ Quando deixamos de usar o free em uma alocação dinâmica, ocorre o que chamamos de memory leak.
- ▶ Ao deixar de usar o free, a memória alocada dinamicamente fica ocupada, mesmo se ela não está sendo mais usada



Memory leak (vazamento de memória)

```
int main()
{
    int i = 0;
    double *p;
    for (i = 0; i < 500; i++){
        // alocando 100MB a cada passo do loop
        p = (double *)malloc(25*1024*1024*sizeof(double));

        if (p != NULL){
            printf("Passo: %d\n", i);
            printf("Memoria alocada\n");
            system("pause");
        } else {
            printf("Passo: %d\n", i);
            printf("Erro! Memoria insuficiente\n");
            system("pause");
        }

    }
    system("pause");
    return 0;
}
```



Alocação de Estruturas

- ▶ Estruturas são alocadas de modo similar aos tipos primitivos

Exemplo:

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

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

```
struct st {
```

```
    int a;
```

```
    float b;
```

```
};
```

```
typedef struct st st; // Define um "apelido" para o tipo estruturado
```

```
...
```

```
int main() {
```

```
    st *p; // Cria uma variável ponteiro p que aponta para um dado do tipo st
```

```
    p = (st *) malloc(sizeof(st)); // Aloca um dado do tipo st
```

```
    ...
```

```
}
```



Alocação de Estruturas

- ▶ Existem 2 formas de manipular os campos de uma variável estruturada alocada dinamicamente:
 - ▶ Sintaxe 1: *(*ponteiro).campo*
 - ▶ Sintaxe 2: *ponteiro->campo*
- ▶ **Exemplo:** considerando o programa anterior, podemos acessar os campos endereçados por *p* das seguintes formas:

```
(*p).a = 15; (*p).b = 3.12;           // sintaxe 1  
printf("a = %d, b = %f\n", p->a, p->b); // sintaxe 2
```



Exemplo 2

```
int main() {
    data *d;
    data a;
    d = malloc(sizeof(data));
    d->dia = 31;
    d->mes = 12;
    d->ano = 2008;
    printf("Data d:\n");
    printf("dia: %d, ", d->dia);
    printf("mes: %d, ", d->mes);
    printf("ano: %d.\n", d->ano);
    a = *d;
    printf("Data a:\n");
    printf("dia: %d, ", a.dia);
    printf("mes: %d, ", a.mes);
    printf("ano: %d.", a.ano);
    free(d);
    return 0;
}
```

```
typedef struct {
    int dia, mes, ano;
}data;
```

Neste exemplo, a variável **d** aponta para uma região alocada dinamicamente, e a variável **a** é estática, alocada em tempo de compilação

Alocação de arrays

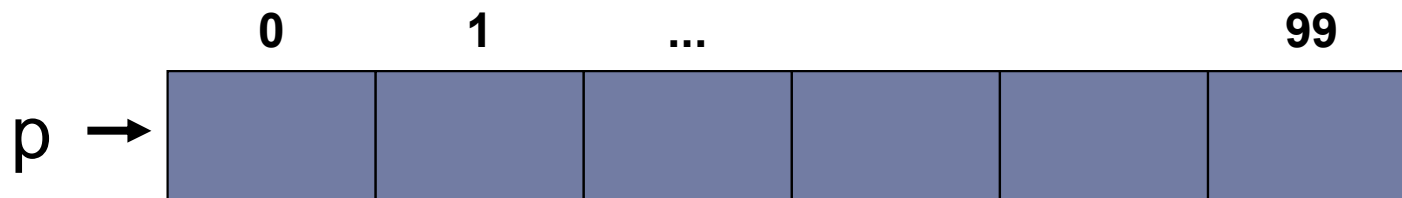
- ▶ Para armazenar um array o compilador C calcula o tamanho, em bytes, necessário e reserva posições sequenciais na memória
 - ▶ Note que isso é muito parecido com alocação dinâmica
- ▶ Existe uma ligação muito forte entre ponteiros e arrays.
 - ▶ O nome do array é apenas um ponteiro que aponta para o primeiro elemento do array.



Alocação de arrays

- ▶ Ao alocarmos memória estamos, na verdade, alocando um array.

```
int *p;  
int i, N = 100;  
p = (int *) malloc(N*sizeof(int));  
for (i = 0; i < N; i++)  
    scanf("%d",&p[i]);
```



Alocação de arrays

- ▶ Note, no entanto, que o array alocado possui apenas uma dimensão.
- ▶ Para liberá-lo da memória, basta chamar a função `free()` ao final do programa:

```
free(p); /* libera o array */
```



Exemplo 3

```
double *produtos;
int n,i;

printf("Informe o número de produtos");
scanf("%d",&n);

// é necessário usar o comando malloc para alocar a memória
produtos = (double *)malloc(n*sizeof(double));

for (i = 0; i < n; i++){
    printf("Informe o valor do produto %d R$:",i+1);
    scanf("%lf", &produtos[i]);
}

printf("\nProdutos cadastrados\n");
for (i = 0; i < n; i++){
    printf("Produto %d - R$: %f\n",i+1, produtos[i]);
}

// ao terminar de usar o vetor, devemos liberar a memória
free(produtos);
```



Exemplo 4

Slide adaptado do material do
Prof. Paulo Coelho

```
#include <stdio.h>
#include <stdlib.h>
int main () {
    int m, i, *A, *B, *C;
    printf("Entre tamanho dos vetores: ");
    scanf("%d", &m);
    A = (int *) malloc(m * sizeof(int)); // Aloca espaço para m números inteiros
    B = (int *) malloc(m * sizeof(int)); // Aloca espaço para m números inteiros
    C = (int *) malloc(m * sizeof(int)); // Aloca espaço para m números inteiros
    // Carrega os elementos do vetor A
    printf("\nVetor A: ");
    for (i = 0; i < m; i++)
        scanf("%d", &A[i]);
    // Carrega os elementos do vetor B
    printf("\nVetor B: ");
    for (i = 0; i < m; i++)
        scanf("%d", &B[i]);
    // Determina e apresenta os elementos do vetor C
    printf("\nVetor C: ");
    for (i = 0; i < m; i++) {
        C[i] = A[i] > B[i] ? A[i] : B[i];
        printf("%d ", C[i]);
    }
    return 0;
}
```



Exemplo 4

Slide adaptado do material do
Prof. Paulo Coelho

```
#include <stdio.h>
#include <stdlib.h>
int main () {
    int m, i, *A, *B, *C;
    printf("Entre tamanho dos vetores: ");
    scanf("%d", &m);
    A = (int *) malloc(m * sizeof(int)); // Aloca espaço para m números inteiros
    B = (int *) malloc(m * sizeof(int)); // Aloca espaço para m números inteiros
    C = (int *) malloc(m * sizeof(int)); // Aloca espaço para m números inteiros
    // Carrega os elementos do vetor A
    printf("\nVetor A: ");
    for (i = 0; i < m; i++)
        scanf("%d", &A[i]);
    // Carrega os elementos do vetor B
    printf("\nVetor B: ");
    for (i = 0; i < m; i++)
        scanf("%d", &B[i]);
    // Determina e apresenta os elementos do vetor C
    printf("\nVetor C: ");
    for (i = 0; i < m; i++) {
        C[i] = A[i] > B[i] ? A[i] : B[i];
        printf("%d ", C[i]);
    }
    return 0;
}
```



Exemplo 4

Slide adaptado do material do
Prof. Paulo Coelho

```
#include <stdio.h>
#include <stdlib.h>
int main () {
    int m, i, *A, *B, *C;
    printf("Entre tamanho dos vetores: ");
    scanf("%d", &m);
    A = (int *) malloc(m * sizeof(int)); // Aloca espaço para m números inteiros
    B = (int *) malloc(m * sizeof(int)); // Aloca espaço para m números inteiros
    C = (int *) malloc(m * sizeof(int)); // Aloca espaço para m números inteiros
    // Carrega os elementos do vetor A
    printf("\nVetor A: ");
    for (i = 0; i < m; i++)
        scanf("%d", &A[i]);
    // Carrega os elementos do vetor B
    printf("\nVetor B: ");
    for (i = 0; i < m; i++)
        scanf("%d", &B[i]);
    // Determina e apresenta os elementos do vetor C
    printf("\nVetor C: ");
    for (i = 0; i < m; i++) {
        C[i] = A[i] > B[i] ? A[i] : B[i];
        printf("%d ", C[i]);
    }
    return 0;
}
```



Exemplo 4

Slide adaptado do material do
Prof. Paulo Coelho

```
#include <stdio.h>
#include <stdlib.h>
int main () {
    int m, i, *A, *B, *C;
    printf("Entre tamanho dos vetores: ");
    scanf("%d", &m);
    A = (int *) malloc(m * sizeof(int)); // Aloca espaço para m números inteiros
    B = (int *) malloc(m * sizeof(int)); // Aloca espaço para m números inteiros
    C = (int *) malloc(m * sizeof(int)); // Aloca espaço para m números inteiros
    // Carrega os elementos do vetor A
    printf("\nVetor A: ");
    for (i = 0; i < m; i++)
        scanf("%d", &A[i]);
    // Carrega os elementos do vetor B
    printf("\nVetor B: ");
    for (i = 0; i < m; i++)
        scanf("%d", &B[i]);
    // Determina e apresenta os elementos do vetor C
    printf("\nVetor C: ");
    for (i = 0; i < m; i++) {
        C[i] = A[i] > B[i] ? A[i] : B[i];
        printf("%d ", C[i]);
    }
    return 0;
}
```

Define o tamanho em
bytes de **um** número
inteiro

Exemplo 4

Slide adaptado do material do
Prof. Paulo Coelho

```
#include <stdio.h>
#include <stdlib.h>
int main () {
    int m, i, *A, *B, *C;
    printf("Entre tamanho dos vetores: ");
    scanf("%d", &m);
    A = (int *) malloc(m * sizeof(int)); // Aloca espaço para m números inteiros
    B = (int *) malloc(m * sizeof(int)); // Aloca espaço para m números inteiros
    C = (int *) malloc(m * sizeof(int)); // Aloca espaço para m números inteiros
    // Carrega os elementos do vetor A
    printf("\nVetor A: ");
    for (i = 0; i < m; i++)
        scanf("%d", &A[i]);
    // Carrega os elementos do vetor B
    printf("\nVetor B: ");
    for (i = 0; i < m; i++)
        scanf("%d", &B[i]);
    // Determina e apresenta os elementos do vetor C
    printf("\nVetor C: ");
    for (i = 0; i < m; i++) {
        C[i] = A[i] > B[i] ? A[i] : B[i];
        printf("%d ", C[i]);
    }
    return 0;
}
```

**Define o quantidade de
elementos do vetor**

Exemplo 4

Slide adaptado do material do
Prof. Paulo Coelho

```
#include <stdio.h>
#include <stdlib.h>
int main () {
    int m, i, *A, *B, *C;
    printf("Entre tamanho dos vetores: ");
    scanf("%d", &m);
    A = (int *) malloc(m * sizeof(int)); // Conversão para o tipo de ponteiro desejado
    B = (int *) malloc(m * sizeof(int)); // Conversão para o tipo de ponteiro desejado
    C = (int *) malloc(m * sizeof(int)); // Conversão para o tipo de ponteiro desejado
    // Carrega os elementos do vetor A
    printf("\nVetor A: ");
    for (i = 0; i < m; i++)
        scanf("%d", &A[i]);
    // Carrega os elementos do vetor B
    printf("\nVetor B: ");
    for (i = 0; i < m; i++)
        scanf("%d", &B[i]);
    // Determina e apresenta os elementos do vetor C
    printf("\nVetor C: ");
    for (i = 0; i < m; i++) {
        C[i] = A[i] > B[i] ? A[i] : B[i];
        printf("%d ", C[i]);
    }
    return 0;
}
```

Conversão para um
ponteiro para o tipo de
dato definido no **sizeof()**



Exemplo 4

Slide adaptado do material do
Prof. Paulo Coelho

```
#include <stdio.h>
#include <stdlib.h>
int main () {
    int m, i, *A, *B, *C;
    printf("Entre tamanho dos vetores: ");
    scanf("%d", &m);
    A = (int *) malloc(m * sizeof(int)); // Aloca espaço para m números inteiros
    B = (int *) malloc(m * sizeof(int)); // Aloca espaço para m números inteiros
    C = (int *) malloc(m * sizeof(int)); // Aloca espaço para m números inteiros
    // Carrega os elementos do vetor A
    printf("\nVetor A: ");
    for (i = 0; i < m; i++)
        scanf("%d", &A[i]);
    // Carrega os elementos do vetor B
    printf("\nVetor B: ");
    for (i = 0; i < m; i++)
        scanf("%d", &B[i]);
    // Determina e apresenta os elementos do vetor C
    printf("\nVetor C: ");
    for (i = 0; i < m; i++) {
        C[i] = A[i] > B[i] ? A[i] : B[i];
        printf("%d ", C[i]);
    }
    return 0;
}
```

Ponteiros podem ser
manipulados como
variáveis indexadas

Alocação de arrays com 2 ou mais dimensões

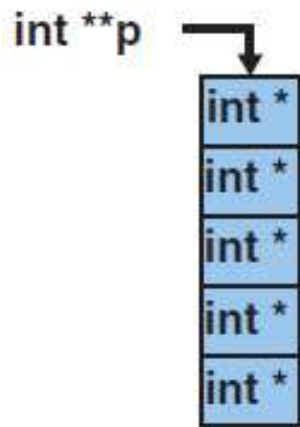
- ▶ Para alocarmos arrays com mais de uma dimensão, utilizamos o conceito de “ponteiro para ponteiro”.
 - ▶ Ex.: `char **ptrPtr;`
- ▶ Para cada nível do ponteiro, fazemos a alocação de uma dimensão do array.



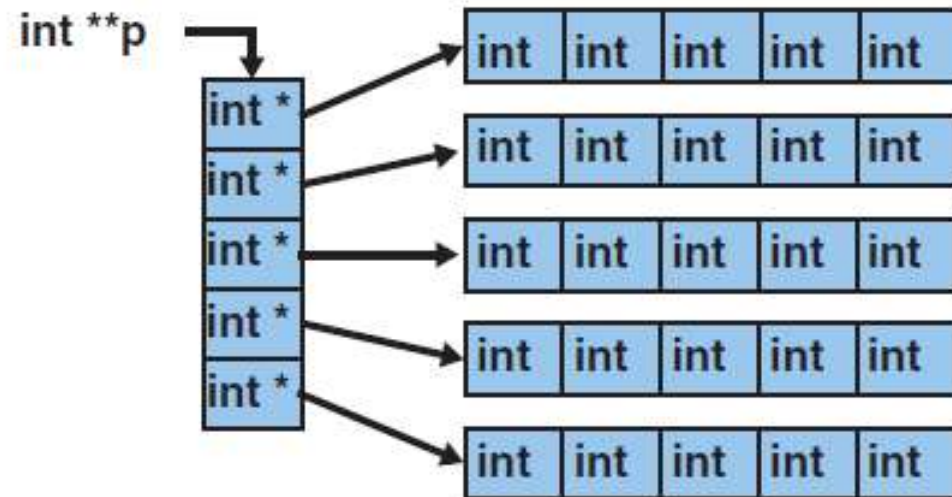
Alocação de arrays com 2 ou mais dimensões

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

1º malloc
Cria as linhas da matriz



2º malloc
Cria as colunas da matriz



Alocação de arrays com 2 ou mais dimensões

- ▶ Em um ponteiro para ponteiro, cada nível do ponteiro permite criar uma nova dimensão no array.

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  int main(){
04      int **p; //2 "*" = 2 níveis
05      = 2 dimensões
06      int i, j, N = 2;
07      p = (int**) malloc(
08          N*sizeof(int *));
09      for (i = 0; i < N; i++){
10          p[i] = (int *)
11              malloc(N*sizeof(int));
12          for (j = 0; j < N; j++){
13              scanf("%d", &p[i][j]);
14          }
15      }
16      system("pause");
17      return 0;
18  }
```

Memória		
#	var	conteúdo
119	int **p;	#120
120	p[0]	#123
121	p[1]	#126
122		
123	p[0][0]	69
124	p[0][1]	74
125		
126	p[1][0]	14
127	p[1][1]	31
128		

Alocação de arrays com 2 ou mais dimensões

- Diferente dos arrays de uma dimensão, para liberar um array com mais de uma dimensão da memória, é preciso liberar a memória alocada em cada uma de suas dimensões, na ordem inversa da que foi alocada

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  int main(){
04      int **p; //2 "*" = 2 níveis = 2 dimensões
05      int i, j, N = 2;
06      p = (int **) malloc(N*sizeof(int *));
07      for (i = 0; i < N; i++){
08          p[i] = (int *) malloc(N*sizeof(int));
09          for (j = 0; j < N; j++)
10              scanf("%d",&p[i][j]);
11      }
12      for (i = 0; i < N; i++){
13          free(p[i]);
14      }
15      free(p);
16      system("pause");
17      return 0;
18  }
```

Liberação de memória

- ▶ Variáveis que não serão utilizadas DEVEM ser liberadas
 - ▶ Recurso de memória é limitado
 - ▶ Permite reutilização de posições de memória
- ▶ Existem 2 formas de liberar uma variável dinâmica:
 - ▶ **Liberação implícita:**
 - ▶ Liberação ocorre quando **o programa é finalizado**.
 - ▶ TODO o espaço alocado para o programa é liberado.
 - ▶ **Liberação explícita:**
 - ▶ Liberação ocorre através da função **free()**
 - ▶ Boa prática: atribuir **NULL** ao ponteiro após a liberação



Exercícios

1. *Faça um programa que leia o tamanho de um vetor de inteiros e reserve dinamicamente o espaço na memória para esse vetor. Em seguida, leia os elementos do vetor, imprima-os, mostre o resultado da soma dos números ímpares presentes no vetor e, por fim, libere o espaço alocado.*
 2. *Faça um programa que leia o tamanho de uma string (vetor de caracteres) e chame uma função para alocar dinamicamente essa string. Em seguida, o usuário deverá informar o conteúdo da string e o programa mostrará a string digitada sem as vogais.*
 3. *Faça um programa que leia as dimensões de uma matriz e aloque dinamicamente a memória para esta variável. Em seguida, o programa deve ler os elementos da matriz e imprimir sua diagonal superior.*
-



Exercícios

4. *Faça um programa que mantenha uma tabela (na memória) para cadastro de bebidas, onde cada dado tem a seguinte estrutura:*

Nome	Volume (ml)	Preço
char[20]	int	float

A tabela deve ser representada por um vetor de ponteiros de bebidas, o qual deve ser devidamente inicializado com NULL (esse vetor deve comportar no máximo 20 bebidas). O programa deve apresentar um menu com as seguintes opções:

- [1] Inserir registro*
- [2] Apagar último registro*
- [3] Imprimir tabela*
- [4] Sair*

