



ALOCAÇÃO DINÂMICA EM C

Prof. Manoel Pereira Junior
IFMG – Campus Formiga

- Ponteiros para ponteiros... Sim, é possível!

tipo_do_ponteiro **nome_do_ponteiro;

Exemplo: ponteiro para ponteiro

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  int main(){
04      int x = 10;
05      int *p = &x;
06      int **p2 = &p;
07      //Endereço em p2
08      printf("Endereco em p2: %p\n",p2);
09      //Conteúdo do endereço
10      printf("Conteúdo em *p2: %p\n",*p2);
11      //Conteúdo do endereço do endereço
12      printf("Conteúdo em **p2: %d\n",**p2);
13      system("pause");
14      return 0;
15  }
```

Memória		
#	var	conteúdo
119		
120	int **p2	#122
121		
122	int *p	#124
123		
124	int x	10
125		

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4 //variável inteira
5 int x;
6 //ponteiro para um inteiro (1 nível)
7 int *p1;
8 //ponteiro para ponteiro de inteiro (2 níveis)
9 int **p2;
10 //ponteiro para ponteiro para ponteiro de inteiro(3 níveis)
11 int ***p3;
12 return 0;
13 }
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     char letra='a';
5     char *ptrChar = &letra;
6     char **ptrPtrChar = &ptrChar;
7     char ***ptrPtr = &ptrPtrChar;
8     printf("Conteudo em *ptrChar: %c\n",*ptrChar);
9     printf("Conteudo em **ptrPtrChar: %c\n",**ptrPtrChar);
10    printf("Conteudo em ***ptrPtr: %c\n",***ptrPtr);
11    return 0;
12 }
```

```
Conteudo em *ptrChar: a
Conteudo em **ptrPtrChar: a
Conteudo em ***ptrPtr: a
```

ALOCAÇÃO DINÂMICA...

- Toda variável deve ser declarada antes de ser usada!
- Nem sempre conhecemos a priori o tamanho das estruturas

EXEMPLO

- Preciso processar os salários dos funcionários de uma empresa.

- Solução 1:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4
5     float salarios[1000];
6
7 }
```

Essa solução é
boa?

- Ao declarar um array, necessariamente precisamos definir seu tamanho (quantidade de memória fixa)!
- Para contornar este problema, usamos arrays com ponteiros!
- Relembrando:
 - (1) um array é um agrupamento sequencial de elementos na memória
 - (2) o nome de um array é um ponteiro para o primeiro elemento

ALOCAÇÃO DINÂMICA É...

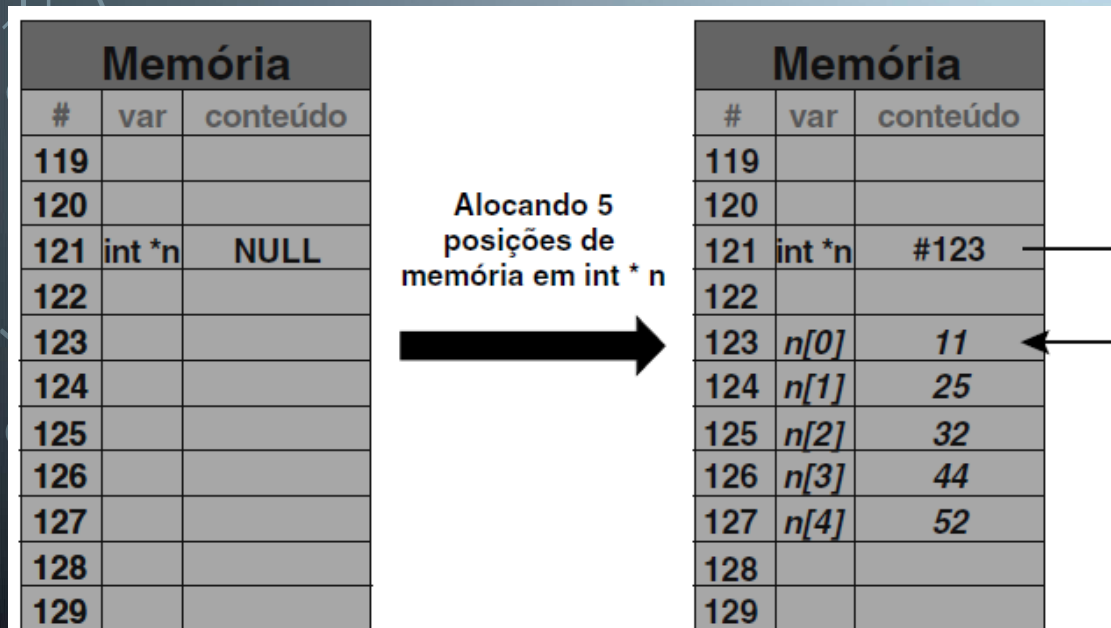
- Requisitar memória em tempo de execução!
- A alocação devolve um ponteiro (endereço) para o início do espaço alocado.

(1) Ponteiro `n` apontando para **NULL**

(2) Requisitamos 5 posições de memória

(3) Recebemos as posições de #123 a #127

(4) `n` passa a se comportar como um array de 5 posições `int(n)`



FUNÇÕES PARA ALOCAÇÃO...

- malloc
- calloc
- realloc
- free
- sizeof

sizeof

- Usada para saber o tamanho (em bytes) de variáveis ou tipos.

```
sizeof nome_da_variável
```

```
sizeof (nome_do_tipo)
```

sizeof

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 struct ponto{
4     int x,y;
5 };
6 int main(){
7     printf("Tamanho char: %d\n",sizeof(char));
8     printf("Tamanho int: %d\n",sizeof(int));
9     printf("Tamanho float: %d\n",sizeof(float));
10    printf("Tamanho double: %d\n",sizeof(double));
11    printf("Tamanho struct ponto: %d\n",sizeof(struct ponto));
12    int x;
13    double y;
14    printf("Tamanho da variavel x: %d\n",sizeof x);
15    printf("Tamanho da variavel y: %d\n",sizeof y);
16    return 0;
17 }
```

```
Tamanho char: 1
Tamanho int: 4
Tamanho float: 4
Tamanho double: 8
Tamanho struct ponto: 8
Tamanho da variavel x: 4
Tamanho da variavel y: 8
```

malloc

- Usada para alocar memória durante a execução do programa.

Pq void?

• `void *malloc (unsigned int num);`

num: o tamanho do espaço de memória a ser alocado

→ Retorna **NULL** em caso de erro

malloc

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int *p;
5     p = (int *) malloc(5*sizeof(int));
6     int i;
7     for (i=0; i<5; i++){
8         printf("Digite o valor da posicao %d: ",i);
9         scanf("%d",&p[i]);
10    }
11    return 0;
12 }
```

Essa operação de "cast"
não é necessária para
ponteiros!!!
(embora possa ser feita)

```
Digite o valor da posicao 0: 1
Digite o valor da posicao 1: 2
Digite o valor da posicao 2: 3
Digite o valor da posicao 3: 4
Digite o valor da posicao 4: 5
```

malloc

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(){
5     int *p;
6     p = (int *) malloc(5*sizeof(int));
7     if(p == NULL){
8         printf("Erro: Memoria Insuficiente!\n");
9         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    return 0;
17 }
```

malloc - cuidado!

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(){
5      char *p;
6      //aloca espaço para 1.000 chars
7      p = malloc(1000);
8      int *y;
9      //aloca espaço para 250 inteiros
10     y = malloc(1000);
11     return 0;
12 }
```


calloc

- Usada para alocar memória durante a execução do programa.

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

num: o número de elementos no array a ser alocado.

size: o tamanho de cada elemento do array.

→ Retorna **NULL** em caso de erro

calloc

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     //alocação com malloc
5     int *p;
6     p = malloc(50*sizeof(int));
7     if(p == NULL){
8         printf("Erro: Memoria Insuficiente!\n");
9     }
10    //alocação com calloc
11    int *p1;
12    p1 = calloc(50,sizeof(int));
13    if(p1 == NULL){
14        printf("Erro: Memoria Insuficiente!\n");
15    }
16    return 0;
17 }
```

realloc

- Usada para alocar memória ou realocar blocos de memória previamente alocados pelas funções malloc(), calloc() ou realloc().

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

***ptr:** ponteiro para um bloco de memória previamente alocado.

num: o tamanho em bytes do espaço de memória a ser alocado.

→ Retorna **NULL** em caso de erro

realloc

- Na prática, a função `realloc()` é utilizada para modificar o tamanho de estruturas em tempo de execução (diminuir ou aumentar o tamanho de uma estrutura).
- Quando faremos isso?

realloc

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int i;
5     int *p = malloc(5*sizeof(int));
6     for (i = 0; i < 5; i++){
7         p[i] = i+1;
8     }
9     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     return 0;
25 }
```

```
1
2
3
4
5

1
2
3

1
2
3
4
5
0
135137
0
0
0
```

free

- Diferentemente das variáveis declaradas durante o desenvolvimento do programa, as variáveis alocadas dinamicamente não são liberadas automaticamente por ele.
- Usada para desalocar memória previamente alocada.



free

```
void free (void *p);
```

p: ponteiro para o início do bloco que será
desalocado

free

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int *p,i;
5     p = malloc(10*sizeof(int));
6     if(p == NULL){
7         printf("Erro: Memoria Insuficiente!\n");
8         exit(1);
9     }
10    for (i = 0; i < 10; i++){
11        p[i] = i+1;
12    }
13    for (i = 0; i < 10; i++){
14        printf("%d\n",p[i]);
15    }
16    //libera a memória alocada
17    free(p);
18    return 0;
19 }
```

```
1
2
3
4
5
6
7
8
9
10
```


free

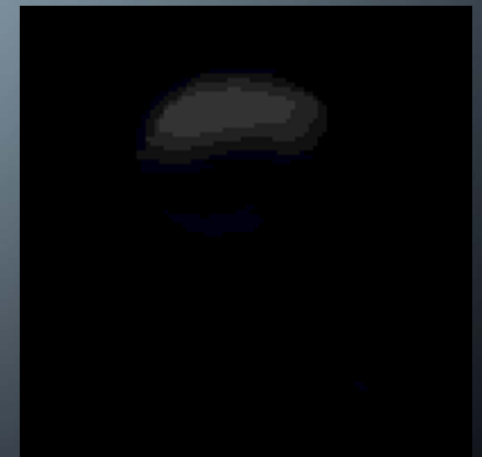
```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int *p,i;
5     p = malloc(10*sizeof(int));
6     if(p == NULL){
7         printf("Erro: Memoria Insuficiente!\n");
8         exit(1);
9     }
10    for (i = 0; i < 10; i++){
11        p[i] = i+1;
12    }
13    //libera a memória alocada
14    free(p);
15    //tenta imprimir o array
16    //cuja memória foi liberada
17    for (i = 0; i < 10; i++){
18        printf("%d\n",p[i]);
19    }
20    return 0;
21 }
```

1
2
3
4
5
6
7
8
9
10

Por que
consegui
imprimir o
vetor p?

free

- Apenas libere a memória quando tiver certeza de que ela não será mais usada!
- não deixe ponteiros “solto” (*dangling pointers*) – ataque de hackers!
- Sempre que usar free, atribua NULL para o ponteiro



free

- Sempre que usar free, atribua NULL para o ponteiro

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(){
4      int *p;
5      p = malloc(10*sizeof(int));
6      free(p);
7      p = NULL;
8  }
```

Alocando arrays multidimensionais...

- Usamos ponteiros para ponteiros
- O nível de apontamento depende da quantidade de dimensões que o array terá

Alocando uma matriz (2 dimensões)

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int **p; //2 "*" = 2 níveis = 2 dimensões
5     int i, j, N = 2;
6     // abaixo defino a quantidade de linhas
7     // da matriz
8     p = malloc(N*sizeof(int *));
9
10    for (i = 0; i < N; i++){
11        // agora para cada linha, defino a
12        // quantidade de colunas
13        p[i] = malloc(N*sizeof(int));
14        for (j = 0; j < N; j++)
15            scanf("%d", &p[i][j]);
16    }
17    return 0;
18 }
```

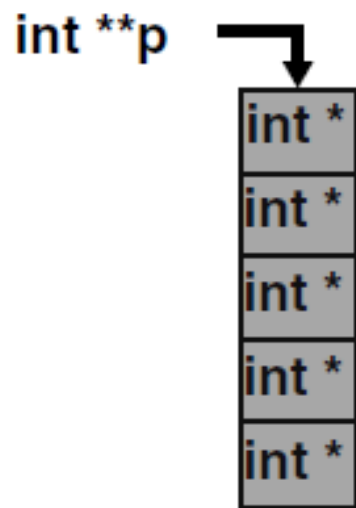
Memória		
#	var	conteúdo
119	int **p;	#120
120	p[0]	#123
121	p[1]	#126
122		
123	p[0][0]	69
124	p[0][1]	74
125		
126	p[1][0]	14
127	p[1][1]	31
128		

Desalocando uma matriz (2 dimensões)

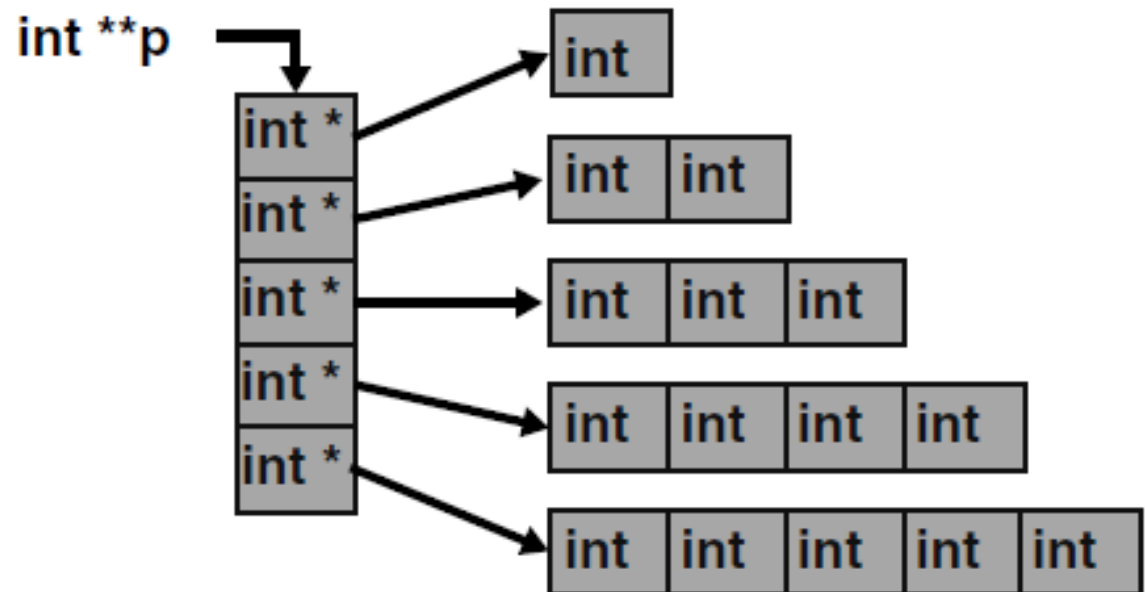
```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(){
4      int **p; //2 "*" = 2 níveis = 2 dimensões
5      int i, j, N = 2;
6      p = malloc(N*sizeof(int *));
7      for (i = 0; i < N; i++){
8          p[i] = malloc(N*sizeof(int));
9          for (j = 0; j < N; j++)
10             scanf("%d",&p[i][j]);
11     }
12     for (i = 0; i < N; i++){
13         free(p[i]);
14     }
15     free(p);
16     return 0;
17 }
```

Outro exemplo...

1º malloc
Cria as linhas da matriz

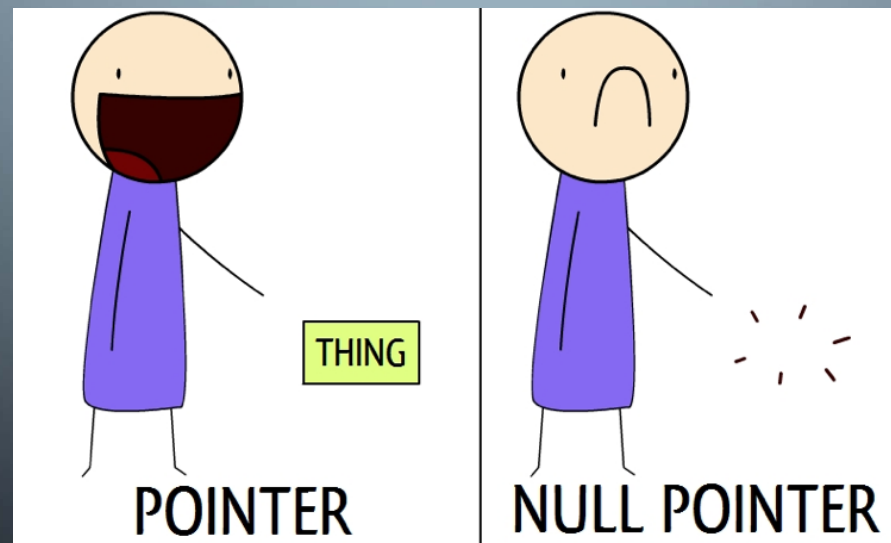


2º malloc
Cria as colunas da matriz

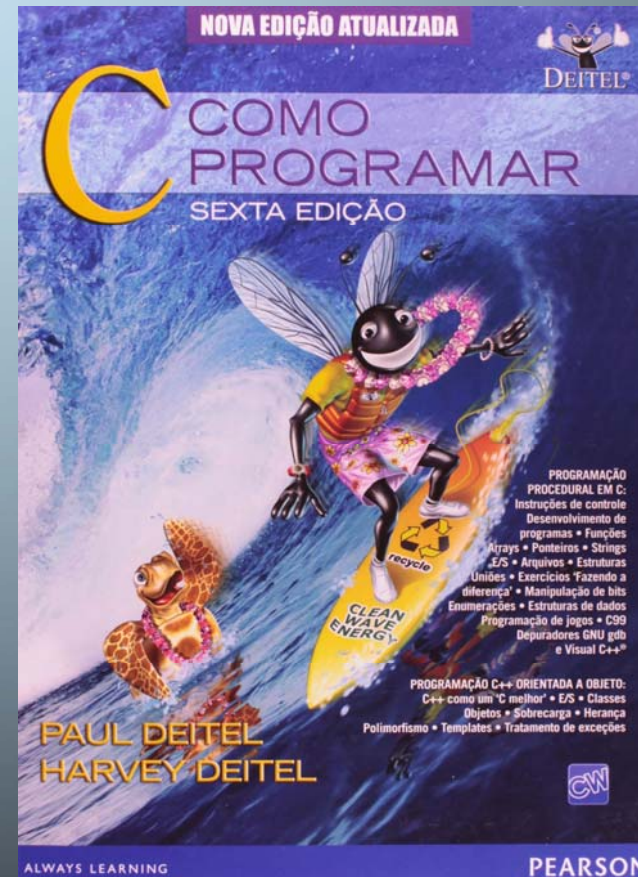
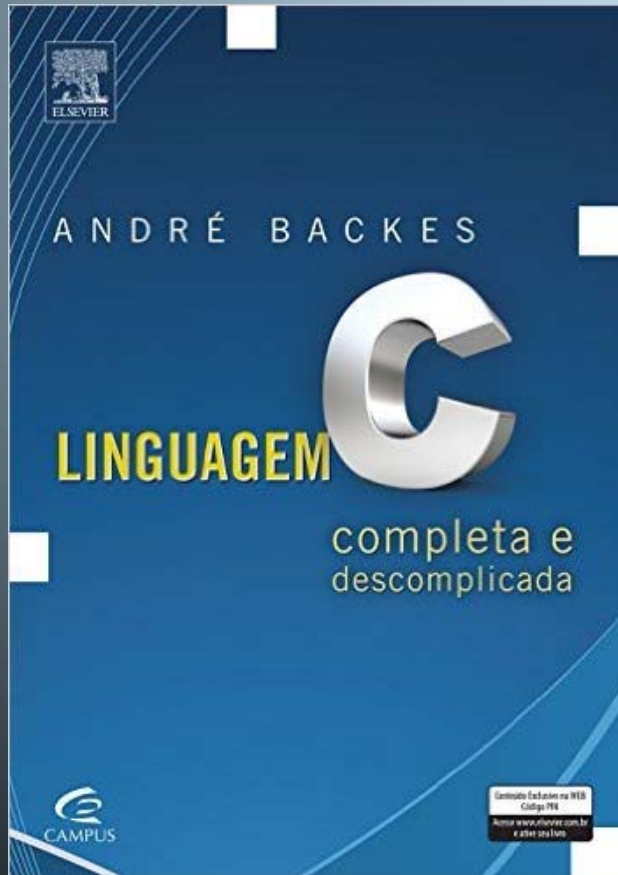


EXERCÍCIOS

- Linguagem C completa e descomplicada.
- Capítulo 11



- Referências:



Prof. Manoel Pereira Junior
IFMG – Campus Formiga