

LINGUAGEM C: PONTEIROS

Prof. Humberto Razente
Sala 1B144

DEFINIÇÃO

○ 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 o ***endereço de memória*** de uma outra variável.
- Um ponteiro é uma variável como qualquer outra do programa – sua diferença é que ela não armazena um valor inteiro, real, caractere ou booleano.
- Ela serve para armazenar endereços de memória (são valores inteiros sem sinal).

DECLARAÇÃO

- Como qualquer variável, um ponteiro também possui um tipo
- É o *asterisco* (*) que informa ao compilador que aquela variável não vai guardar um valor mas sim um endereço para o tipo especificado.

```
//declaração de variável  
tipo_variável *nome_variável;  
  
//declaração de ponteiro  
tipo_ponteiro *nome_ponteiro;
```

```
int x;  
float y;  
struct ponto p;  
  
int *x;  
float *y;  
struct ponto *p;
```

DECLARAÇÃO

- Exemplos de declaração de variáveis e ponteiros

```
int main() {  
    //Declara um ponteiro para int  
    int *p;  
    //Declara um ponteiro para float  
    float *x;  
    //Declara um ponteiro para char  
    char *y;  
    //Declara um ponteiro para struct ponto  
    struct ponto *p;  
    //Declara uma variável do tipo int e um ponteiro para int  
    int soma, *p2,;  
  
    return 0;  
}
```

DECLARAÇÃO

- Na linguagem C, quando declaramos um ponteiro nós informamos ao compilador para que tipo de variável vamos apontá-lo
 - Um ponteiro **int*** aponta para um inteiro, isto é, **int**
 - Esse ponteiro guarda o endereço de memória onde se encontra armazenada uma variável do tipo **int**

INICIALIZAÇÃO

- Ponteiros apontam para uma posição de memória
 - **Cuidado:** Ponteiros não inicializados apontam para um lugar indefinido
- Exemplo
 - `int *p;`

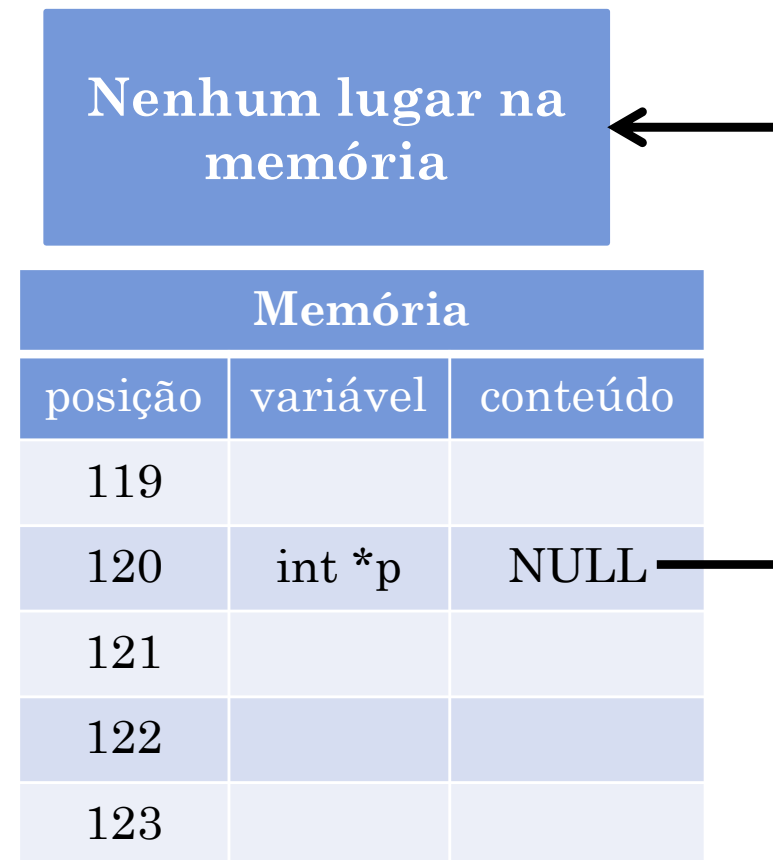
Memória		
posição	variável	conteúdo
119		
120	int *p	????
121		
122		
123		

INICIALIZAÇÃO

- Um ponteiro pode ter o valor especial NULL que é o endereço de nenhum lugar

- Exemplo


- `int *p = NULL;`



INICIALIZAÇÃO

- Os ponteiros devem ser inicializados antes de serem usados
- Assim, devemos apontar um ponteiro para um lugar conhecido
 - Podemos apontá-lo para uma variável que já exista no programa


Memória		
posição	variável	conteúdo
119		
120	int *p	122
121		
122	int c	10
123		



INICIALIZAÇÃO

- O ponteiro armazena o endereço da variável para onde ele aponta
 - Para saber o endereço de memória de uma variável do nosso programa, usamos o operador **&**
 - Ao armazenar o endereço, o ponteiro estará apontando para aquela variável

Memória		
posição	variável	conteúdo
119		
120	int *p	122
121		
122	int c	10
123		



```
int main() {  
    //Declara uma variável int contendo o valor 10  
    int c = 10;  
    //Declara um ponteiro para int  
    int *p;  
    //Atribui ao ponteiro o endereço da variável int  
    p = &c ;  
    return 0;  
}
```

UTILIZAÇÃO

- Tendo um ponteiro armazenado um endereço de memória, como saber o valor guardado dentro dessa posição?

UTILIZAÇÃO

- Para acessar o **valor** guardado dentro de uma posição na memória apontada por um ponteiro, basta usar o operador ***asterisco*** “*” na frente do nome do ponteiro

```
int main() {  
    //Declara uma variável int contendo o valor 10  
    int c = 10;  
    //Declara um ponteiro para int  
    int *p;  
    //Atribui ao ponteiro o endereço da variável int  
    p = &c;  
    printf("Conteudo apontado por p: %d \n", *p); // 10  
    //Atribui um novo valor à posição de memória apontada por p  
    *p = 12;  
    printf("Conteudo apontado por p: %d \n", *p); // 12  
    printf("Conteudo de count: %d \n", c); // 10  
  
    return 0;  
}
```

UTILIZAÇÃO

- ***p** :conteúdo da posição de memória apontado por **p**
- **&c**: o endereço na memória onde está armazenada a variável **c**

```
int main() {  
    //Declara uma variável int contendo o valor 10  
    int c = 10;  
    //Declara um ponteiro para int  
    int *p;  
    //Atribui ao ponteiro o endereço da variável int  
    p = &c;  
    printf("Conteudo apontado por p: %d \n", *p); // 10  
    //Atribui um novo valor à posição de memória apontada por p  
    *p = 12;  
    printf("Conteudo apontado por p: %d \n", *p); // 12  
    printf("Conteudo de count: %d \n", c); // 10  
  
    return 0;  
}
```

UTILIZAÇÃO

- De modo geral, um ponteiro só deveria receber o endereço de memória de uma variável do mesmo tipo do ponteiro
 - Isso ocorre porque diferentes tipos de variáveis ocupam espaços de memória de tamanhos diferentes
 - Na verdade, nós podemos atribuir a um ponteiro de inteiro (**int ***) o endereço de uma variável do tipo **float**. No entanto, o compilador assume que qualquer endereço que esse ponteiro armazene apontará para uma variável do tipo **int**
 - Isso gera problemas na interpretação dos valores

UTILIZAÇÃO

```
int main() {  
    int *p, *p1, x = 10;  
    float y = 20.0;  
    p = &x;  
    printf("Conteudo apontado por p: %d \n", *p);  
  
    p1 = p;  
    printf("Conteudo apontado por p1: %d \n", *p1);  
  
    p = &y;  
    printf("Conteudo apontado por p: %d \n", *p);  
    printf("Conteudo apontado por p: %f \n", *p);  
    printf("Conteudo apontado por p: %f \n", *((float*)p));  
  
    return 0;  
}
```

```
Conteudo apontado por p: 10  
Conteudo apontado por p1: 10  
Conteudo apontado por p: 1101004800  
Conteudo apontado por p: 0.000000  
Conteudo apontado por p: 20.000000
```

OPERAÇÕES COM PONTEIROS

○ Atribuição

- p1 aponta para o mesmo lugar que p

```
int *p, *p1;  
int c = 10;  
p = &c;  
p1 = p; //equivale a p1 = &c;
```

- A seguir, a variável apontada por p1 recebe o conteúdo da variável apontada por p

```
int *p, *p1;  
int c = 10, d = 20;  
p = &c;  
p1 = &d;  
  
*p1 = *p; //equivale a d = c;
```


OPERAÇÕES COM PONTEIROS

- Apenas duas operações aritméticas podem ser utilizadas com o endereço armazenado pelo ponteiro:
 - adição e subtração
- podemos apenas somar e subtrair valores INTEIROS

```
char *p;
```

```
p++;
```

- soma 1 no endereço armazenado no ponteiro (vezes 1 byte do char)

```
p--;
```

- subtrai 1 no endereço armazenado no ponteiro (vezes 1 byte do char)

```
p = p+15;
```

- soma 15 no endereço armazenado no ponteiro (vezes 1 byte do char)

OPERAÇÕES COM PONTEIROS

- As operações de adição e subtração no endereço dependem do tipo de dado que o ponteiro aponta
 - Considere um ponteiro para inteiro, `int *`
 - O tipo `int` ocupa um espaço de 4 bytes na memória
 - Assim, nas operações de adição e subtração são adicionados/subtraídos 4 bytes por incremento/decremento, pois esse é o tamanho de um inteiro na memória e, portanto, é também o valor mínimo necessário para sair dessa posição reservada de memória

Memória		
posição	variável	conteúdo
119		
120	int a	10
121		
122		
123		
124	int b	20
125		
126		
127		
128	char c	'k'
129	char d	's'
130		

OPERAÇÕES COM PONTEIROS

- Operações ilegais com ponteiros
 - Dividir ou multiplicar ponteiros;
 - Somar o endereço de dois ponteiros
 - Não se pode adicionar ou subtrair valores dos tipos **float** ou **double** de ponteiros
`int *x;`
`x += 5.5;` → erro de compilação

OPERAÇÕES COM PONTEIROS

- Já sobre seu conteúdo apontado, valem todas as operações
 - `(*p)++;`
 - incrementar o conteúdo da variável apontada pelo ponteiro `p`;
 - cuidado:
 - `*p++;`
 - equivale a
 - `p++;`
 - `*p = (*p) * 10;`
 - multiplica o conteúdo da variável apontada pelo ponteiro `p` por 10:

```
int main() {  
    int *p, c = 10;  
    p = &c;  
    (*p)++;  
    printf("\n c = %d", c); // imprime 11  
    *p = *p * 10;  
    printf("\n c = %d", c); // imprime 110  
}
```

OPERAÇÕES COM PONTEIROS

○ Operações relacionais

- == e != para saber se dois ponteiros são iguais ou diferentes
- >, <, >= e <= para saber qual ponteiro aponta para uma posição mais alta na memória

```
int main() {  
    int *p, *p1, x, y;  
    p = &x;  
    p1 = &y;  
    if (p == p1)  
        printf("Ponteiros iguais\n");  
    else  
        printf("Ponteiros diferentes\n");  
  
    return 0;  
}
```

PONTEIROS GENÉRICOS

- Normalmente, um ponteiro aponta para um tipo específico de dado
 - Um ponteiro genérico é um ponteiro que pode apontar para qualquer tipo de dado.
- Declaração

```
void *nome_ponteiro;
```

PONTEIROS GENÉRICOS

○ Exemplos

```
int main() {  
    void *pp;  
    int *p1, p2 = 10;  
    p1 = &p2;  
    //recebe o endereço de um inteiro  
    pp = &p2;  
    printf("Endereco em pp: %p \n", pp);  
    //recebe o endereço de um ponteiro para inteiro  
    pp = &p1;  
    printf("Endereco em pp: %p \n", pp);  
    //recebe o endereço guardado em p1 (endereço de p2)  
    pp = p1;  
    printf("Endereco em pp: %p \n", pp);  
  
    return 0;  
}
```

PONTEIROS GENÉRICOS

- Para acessar o **conteúdo** de um ponteiro genérico é preciso antes convertê-lo para o tipo de ponteiro com o qual se deseja trabalhar
 - Isso é feito via *type cast*

```
int main() {  
    void *pp;  
    int p2 = 10;  
    // ponteiro genérico recebe o endereço de um  
    // inteiro  
    pp = &p2;  
    // para acessar o conteúdo do ponteiro genérico  
    printf("Conteúdo: %d\n", *pp); //ERRO  
    // converte o ponteiro genérico pp para (int *)  
    // antes de acessar seu conteúdo.  
    printf("Conteúdo: %d\n", *(int*)pp); //CORRETO  
  
    return 0;  
}
```

PONTEIROS E ARRAYS

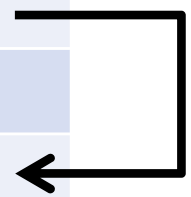
- Ponteiros e arrays possuem uma ligação muito forte
 - Arrays são agrupamentos de dados do mesmo tipo na memória
 - Quando declaramos um array, informamos ao computador para reservar uma certa quantidade de memória a fim de armazenar os elementos do array de forma sequencial
 - Como resultado dessa operação, o computador nos devolve um ponteiro que aponta para o começo dessa sequência de bytes na memória

PONTEIROS E ARRAYS

- O nome do array (sem índice) é apenas um ponteiro que aponta para o primeiro elemento do array

```
int vet[5] = {1,2,3,4,5};  
int *p;  
  
p = vet;
```

Memória		
posição	variável	conteúdo
119		
120		
121	int *p	123
122		
123	int vet[5]	1
127		2
131		3
135		4
139		5
143		



PONTEIROS E ARRAYS

- Os colchetes [] substituem o uso conjunto de operações aritméticas e de acesso ao conteúdo (operador “*”) no acesso ao conteúdo de uma posição de um array ou ponteiro.
 - O valor entre colchetes é o deslocamento a partir da posição inicial do array.
 - Nesse caso, **p[2]** equivale a ***(p+2)**.

```
int main () {  
    int vet[5] = {1, 2, 3, 4, 5};  
    int *p;  
    p = vet;  
  
    printf("Terceiro elemento: %d ou %d", p[2], *(p+2));  
  
    return 0;  
}
```

PONTEIROS E ARRAYS

- Nesse exemplo

```
int vet[5] = {1, 2, 3, 4, 5};  
int *p;  
  
p = vet;
```

- Temos que:

- ***p** é equivalente a **vet[0]**
- **vet[índice]** é equivalente a ***(p+índice)**
- **vet** é equivalente a **&vet[0]**
- **&vet[índice]** é equivalente a **(vet + índice)**

PONTEIROS E ARRAYS

Usando array

```
int main() {  
    int vet[5] = {1, 2, 3, 4, 5};  
    int *p = vet;  
    int i;  
    for (i = 0; i < 5; i++)  
        printf("%d\n", p[i]);  
  
    return 0;  
}
```

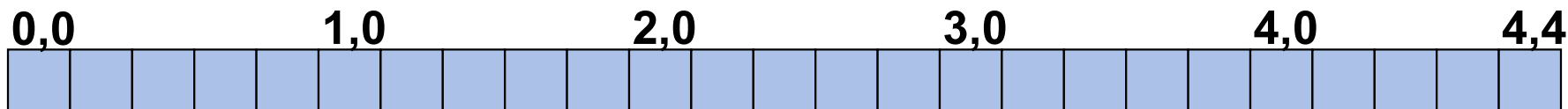
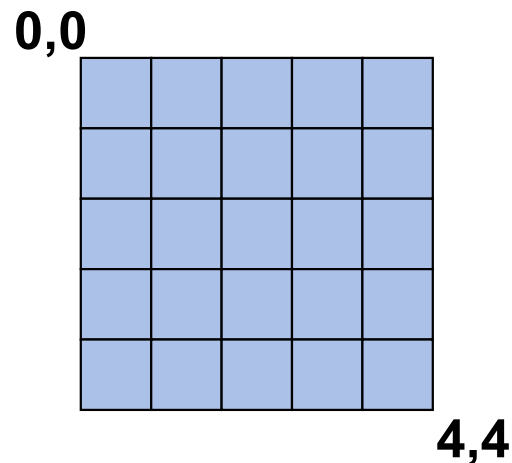
Usando ponteiro

```
int main() {  
    int vet[5] = {1, 2, 3, 4, 5};  
    int *p = vet;  
    int i;  
    for (i = 0; i < 5; i++)  
        printf("%d\n", *(p+i));  
  
    return 0;  
}
```

PONTEIROS E ARRAYS

○ Arrays Multidimensionais

- Apesar de terem mais de uma dimensão, na memória os dados são armazenados linearmente
- Ex.:
- `int mat[5][5];`



PONTEIROS E ARRAYS

- Pode-se então percorrer as várias dimensões do array como se existisse apenas uma dimensão. As dimensões mais a direita mudam mais rápido

Usando array

```
int main() {  
    int mat[2][2] = {{1,2},{3,4}};  
    int i,j;  
    for(i=0;i<2;i++)  
        for(j=0;j<2;j++)  
            printf("%d\n", mat[i][j]);  
  
    return 0;  
}
```

Usando ponteiro

```
int main() {  
    int mat[2][2] = {{1,2},{3,4}};  
    int *p = &mat[0][0];  
    int i;  
    for(i=0;i<4;i++)  
        printf("%d\n", *(p+i));  
  
    return 0;  
}
```

→ `int *p = mat;`

PONTEIRO PARA STRUCT

- Existem duas abordagens para acessar o conteúdo de um ponteiro para uma struct
- Abordagem 1
 - Devemos acessar o conteúdo do ponteiro para struct para somente depois acessar os seus campos e modificá-los.
- Abordagem 2
 - Podemos usar o *operador seta* “->”
 - **ponteiro->nome_campo**

```
struct ponto {  
    int x, y;  
};
```

```
struct ponto q;  
struct ponto *p;
```

```
p = &q;
```

```
(*p).x = 10;  
p->y = 20;
```

PONTEIRO PARA PONTEIRO

- A linguagem C permite criar ponteiros com diferentes níveis de apontamento
 - É possível criar um ponteiro que aponte para outro ponteiro, criando assim vários níveis de apontamento
 - Assim, um ponteiro poderá apontar para outro ponteiro, que, por sua vez, aponta para outro ponteiro, que aponta para um terceiro ponteiro e assim por diante.

PONTEIRO PARA PONTEIRO

- Um ponteiro para um ponteiro é como se você anotasse o endereço de um papel que tem o endereço da casa do seu amigo.
- Podemos declarar um ponteiro para um ponteiro com a seguinte notação
 - `tipo_ponteiro **nome_ponteiro;`
- Acesso ao conteúdo
 - `**nome_ponteiro` é o conteúdo final da variável apontada;
 - `*nome_ponteiro` é o conteúdo do ponteiro intermediário.

PONTEIRO PARA PONTEIRO

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

int main() {
    int x = 10;
    int *p1 = &x;
    int **p2 = &p1;

    // Endereço de p2
    printf("Endereco de p2: %d \n", p2);
    // Conteúdo do endereço (endereço de p1)
    printf("Conteúdo em *p2: %d \n", *p2);
    // Conteúdo do endereço do endereço, ou
    // seja, o valor da variável apontada por p1
    printf("Conteúdo em **p2: %d \n", **p2);

    return 0;
}
```

Memória		
posição	variável	conteúdo
119		
120		
121		
122	int **p2	124
123		
124	int *p1	126
125		
126	int x	10
127		

PONTEIRO PARA PONTEIRO

- A quantidade de asteriscos (*) na declaração do ponteiro indica o número de níveis de apontamento que ele possui

```
//variável inteira
int x;
//ponteiro para um inteiro (1 nível)
int *p1;
//ponteiro para ponteiro de inteiro (2 níveis)
int **p2;
//ponteiro para ponteiro para ponteiro de inteiro (3 níveis)
int ***p3;
```

PONTEIRO PARA PONTEIRO

- Conceito de “ponteiro para ponteiro”:

```
char letra = 'a';  
char *p1;  
char **p2;  
char ***p3;
```

```
p1 = &letra;  
p2 = &p1;  
p3 = &p2;
```

Memória		
posição	variável	conteúdo
119		
120	char ***p3	122
121		
122	char **p2	124
123		
124	char *p1	126
125		
126	char letra	'a'
127		

```
graph TD
    p3[120: char ***p3] --> p2[122: char **p2]
    p2 --> p1[124: char *p1]
    p1 --> letra[126: char letra]
    letra --> a['a']
```

MATERIAL COMPLEMENTAR

○ Vídeo Aulas

- Aula 55: Ponteiros pt.1 – Conceito
 - Aula 56: Ponteiros pt.2 – Operações
 - Aula 57: Ponteiros pt.3 – Ponteiro Genérico
 - Aula 58: Ponteiros pt.4 – Ponteiros e Arrays
 - Aula 59: Ponteiros pt.5 – Ponteiro para Ponteiro
-
- <https://programacaodescomplicada.wordpress.com/indice/linguagem-c/>



LINGUAGEM C: PONTEIROS

38

Contém slides originais gentilmente
disponibilizados pelo Prof. André R. Backes (UFU)



LINGUAGEM C: ALOCAÇÃO DINÂMICA

Prof. Humberto Razente

Sala 1B144

39

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, em tempo de execução, o quanto de memória um programa irá precisar
- Exemplo
 - 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! Não sabemos o valor de **N**

```
int N, i;  
  
scanf("%d", &N)  
  
double produtos[N];
```

Funciona, mas não é o mais indicado

DEFINIÇÃO

- A *alocação dinâmica* permite ao programador criar “variáveis” em tempo de execução, ou seja, alocar memória para novas variáveis quando o programa está sendo executado, e não apenas quando se está escrevendo o programa
 - 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
- 

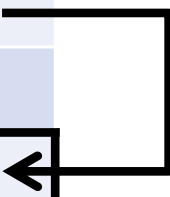
ALOCANDO MEMÓRIA

Memória		
posição	variável	conteúdo
119		
120		
121	int *p	NULL
122		
123		
124		
125		
126		
127		
128		

**Alocando 5
posições de
memória em int *p**



Memória		
posição	variável	conteúdo
119		
120		
121	int *p	123
122		
123	p[0]	11
124	p[1]	25
125	p[2]	32
126	p[3]	44
127	p[4]	52
128		



ALOCAÇÃO DINÂMICA

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

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.

ALOCAÇÃO DINÂMICA - MALLOC

- 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.

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

ALOCAÇÃO DINÂMICA - MALLOC

- Alocar 1000 bytes de memória:

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

- Alocar espaço para 50 inteiros:

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

ALOCAÇÃO DINÂMICA - MALLOC

○ Operador **sizeof()**

- Retorna o número de *bytes* de um dado tipo de dado.
Ex.: int, float, char, struct...

```
struct ponto{  
    int x,y;  
};  
  
int main() {  
  
    printf("char: %d\n", sizeof(char)); // 1  
    printf("int: %d\n", sizeof(int)); // 4  
    printf("float: %d\n", sizeof(float)); // 4  
    printf("ponto: %d\n", sizeof(struct ponto)); // 8  
  
    return 0;  
}
```


ALOCAÇÃO DINÂMICA - MALLOC

○ Operador **sizeof()**

- No exemplo anterior,

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

- **sizeof(int)** retorna 4
 - número de bytes do tipo **int** na memória
- Portanto, são alocados 200 bytes ($50 * 4$)
- 200 bytes = 50 posições do tipo **int** na memória

ALOCAÇÃO DINÂMICA - MALLOC

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

```
int main() {
    int *p;
    p = (int *) malloc(5*sizeof(int));
    if(p == NULL) {
        printf("Erro: Memoria Insuficiente!\n");
        system("pause");
        exit(1);
    }
    int i;
    for (i=0; i<5; i++) {
        printf("Digite o valor da posicao %d: ", i);
        scanf("%d", &p[i]);
    }

    return 0;
}
```

ALOCAÇÃO DINÂMICA - CALLOC

○ calloc

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

```
void *calloc (unsigned int nitems, 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.

ALOCAÇÃO DINÂMICA - CALLOC

○ Exemplo da função **calloc**

```
int main() {  
    //alocação com malloc  
    int *p;  
    p = (int *) malloc(50*sizeof(int));  
    if(p == NULL) {  
        printf("Erro: Memoria Insuficiente!\n");  
    }  
    //alocação com calloc  
    int *p1;  
    p1 = (int *) calloc(50, sizeof(int));  
    if(p1 == NULL) {  
        printf("Erro: Memoria Insuficiente!\n");  
    }  
  
    return 0;  
}
```

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

o 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.

```
int main() {
    int i;
    int *p = malloc(5*sizeof(int));
    for (i = 0; i < 5; i++) {
        p[i] = i+1;
    }
    for (i = 0; i < 5; i++) {
        printf("%d\n", p[i]);
    }
    printf("\n");
    //Diminui o tamanho do array
    p = realloc(p, 3*sizeof(int));
    for (i = 0; i < 3; i++) {
        printf("%d\n", p[i]);
    }
    printf("\n");
    //Aumenta o tamanho do array
    p = realloc(p, 10*sizeof(int));
    for (i = 0; i < 10; i++) {
        printf("%d\n", p[i]);
    }

    return 0;
}
```

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
- 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

○ Exemplo da função **free()**

```
int main() {  
    int *p, i;  
    p = (int *) malloc(50*sizeof(int));  
    if(p == NULL) {  
        printf("Erro: Memoria Insuficiente!\n");  
        system("pause");  
        exit(1);  
    }  
    for (i = 0; i < 50; i++) {  
        p[i] = i+1;  
    }  
    for (i = 0; i < 50; i++) {  
        printf("%d\n", p[i]);  
    }  
    //libera a memória alocada  
    free(p);  
  
    return 0;  
}
```

ALOCAÇÃO DINÂMICA – C++

- Palavras reservadas: **new** e **delete**

```
int main() {  
  
    // C  
    int *p1 = (int *) malloc (sizeof(int));  
    free(p1);  
  
    int *p2 = (int *) malloc (10 * sizeof(int));  
    free(p2);  
  
    // C++  
    int *p3 = new int;  
    delete p3;  
  
    int *p4 = new int[10];  
    delete []p4;  
  
    return 0;  
}
```

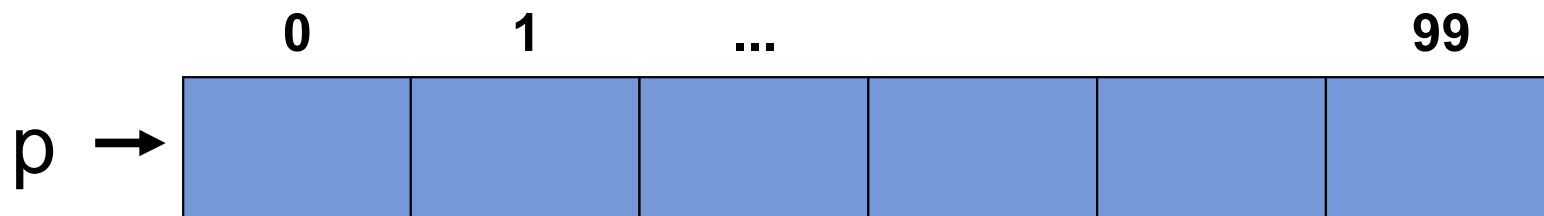
ALOCAÇÃO DE ARRAYS

- 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:

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

ALOCAÇÃO DE ARRAYS

- Para alocarmos arrays com mais de uma dimensão, utilizamos o conceito de “ponteiro para ponteiro”.
 - Ex.: `char ***p3;`
- Para cada nível do ponteiro, fazemos a alocação de uma dimensão do array

ALOCAÇÃO DE ARRAYS

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

```
int **p; //2 "*" = 2 níveis = 2 dimensões
int i, j, N = 2;
p = (int**) malloc(N*sizeof(int*));

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

Memória		
posição	variável	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		

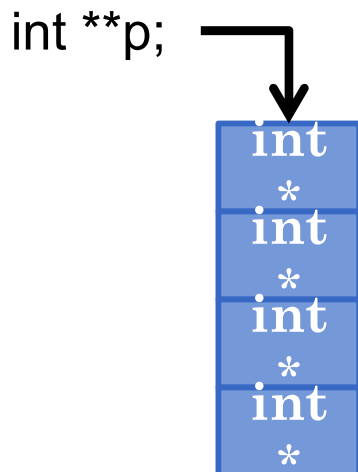
```
graph TD
    p[119: int **p] --> p0[120: p[0]]
    p --> p1[121: p[1]]
    p0 --> p00[123: p[0][0]]
    p0 --> p01[124: p[0][1]]
    p1 --> p10[126: p[1][0]]
    p1 --> p11[127: p[1][1]]
    p00 --> 123_69[123: 69]
    p01 --> 124_74[124: 74]
    p10 --> 126_14[126: 14]
    p11 --> 127_31[127: 31]
```


ALOCAÇÃO DE ARRAYS

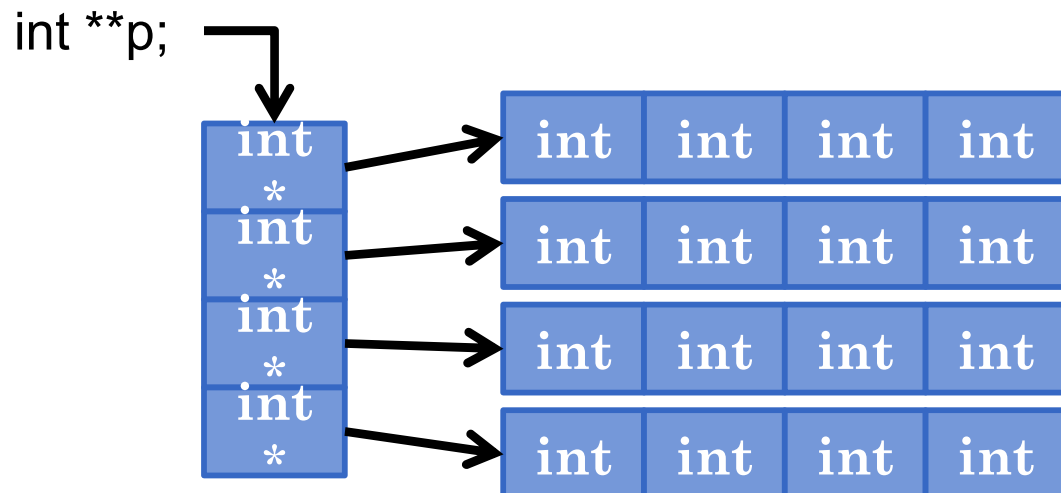
- Em um ponteiro para ponteiro, cada nível do ponteiro permite criar uma nova dimensão no array.

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

1º malloc:
cria as linhas

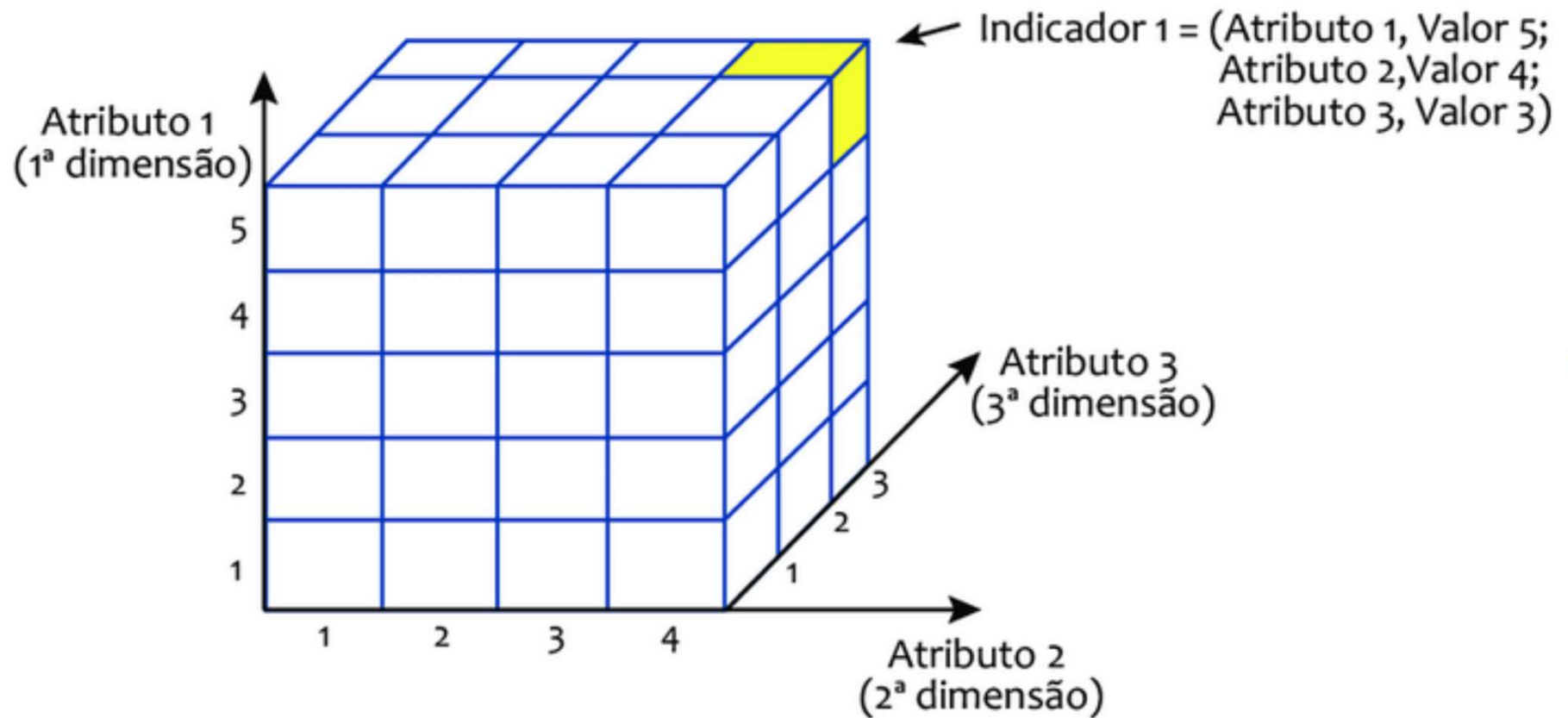


2º malloc:
cria as colunas



ALOCAÇÃO DE ARRAYS: EXEMPLO

- 3 dimensões:



DESALOCAÇÃO DE ARRAYS

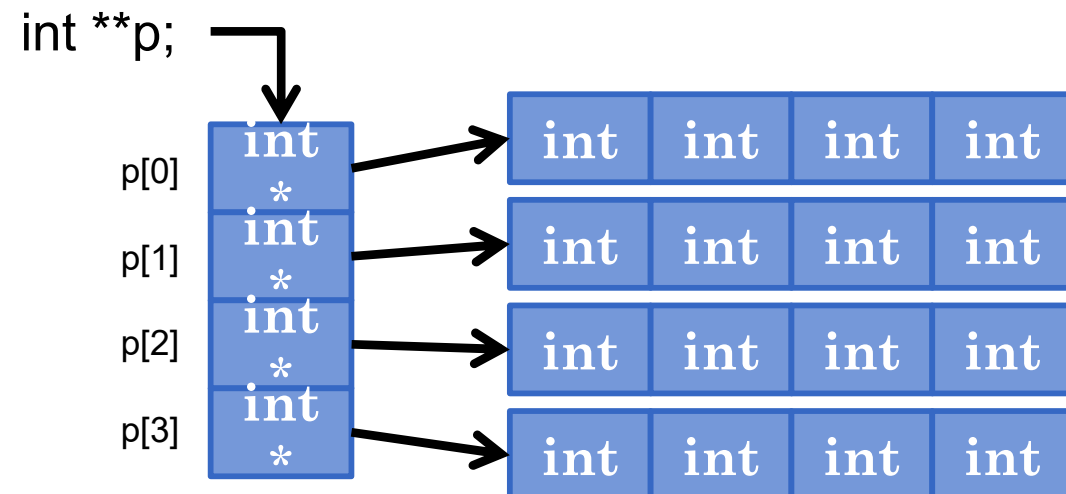
- Diferente dos arrays de uma dimensão, para liberar um array com mais de uma dimensão da memória, é preciso liberar a memória alocada em cada uma de suas dimensões, na ordem inversa da que foi alocada

DESALOCAÇÃO DE ARRAYS

```
int **p; //2 "*" = 2 níveis = 2 dimensões
int i, j, N = 2;
p = (int**) malloc(N*sizeof(int*));

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

```
for (i = 0; i < N; i++)
    free(p[i]);
free(p);
```



MATERIAL COMPLEMENTAR

○ Vídeo Aulas

- Aula 60: Alocação Dinâmica pt.1 – Introdução
 - Aula 61: Alocação Dinâmica pt.2 – Sizeof
 - Aula 62: Alocação Dinâmica pt.3 – Malloc
 - Aula 63: Alocação Dinâmica pt.4 – Calloc
 - Aula 64: Alocação Dinâmica pt.5 – Realloc
 - Aula 65: Alocação Dinâmica pt.6 – Alocação de Matrizes
-
- <https://programacaodescomplicada.wordpress.com/indicadores/linguagem-c/>



LINGUAGEM C: ALOCAÇÃO DINÂMICA

70

Contém slides originais gentilmente
disponibilizados pelo Prof. André R. Backes (UFU)