

# RECURSÃO, ALOCAÇÃO DE MEMÓRIA & TIPO ABSTRATO DE DADOS

---

Prof. André Backes | @progdescomplicada

# RECURSÃO

---

# Recursão

- Na linguagem C, uma função pode chamar outra função.
  - A função `main()` pode chamar qualquer função, seja ela da biblioteca da linguagem (como a função `printf()`) ou definida pelo programador (função `imprime()`).
- Uma função também pode chamar a si própria
  - A qual chamamos de ***função recursiva***.

# Recursão

- A recursão também é chamada de definição circular. Ela ocorre quando algo é definido em termos de si mesmo.
- Um exemplo clássico de função que usa recursão é o cálculo do fatorial de um número:
  - $3! = 3 * 2!$
  - $4! = 4 * 3!$
  - $n! = n * (n - 1)!$

# Recursão

$$0! = 1$$

$$1! = 1 * 0!$$

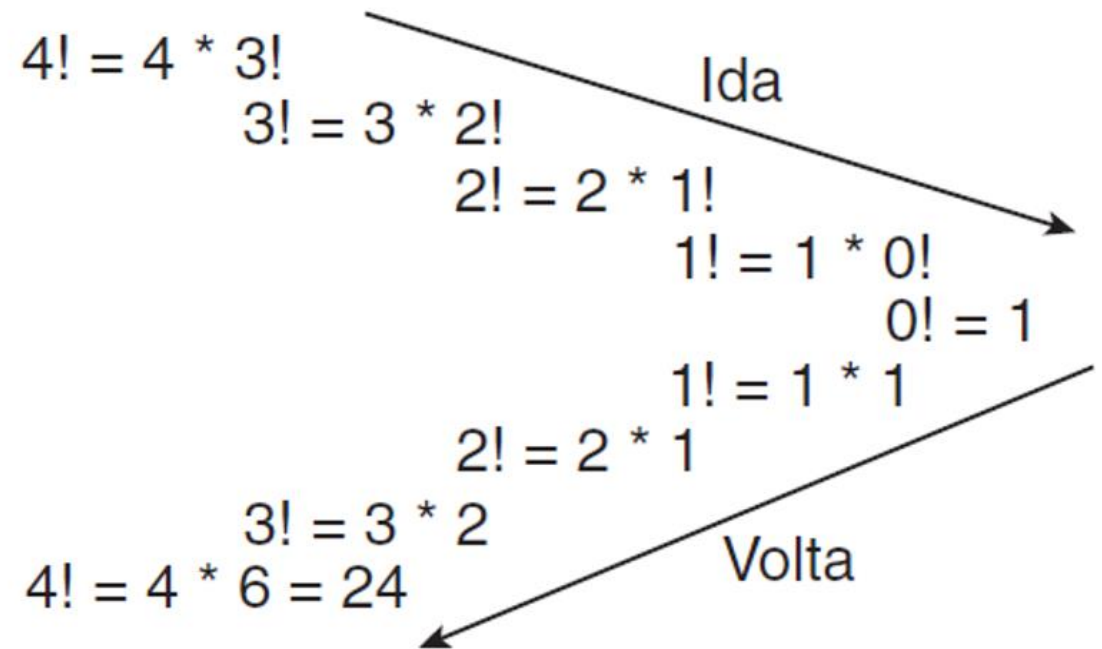
$$2! = 2 * 1!$$

$$3! = 3 * 2!$$

$$4! = 4 * 3!$$

$$n! = n * (n - 1)! : \text{fórmula geral}$$

$$0! = 1 : \text{caso-base}$$



# Recursão

## Com Recursão

```
int fatorial(int n){  
    if (n == 0)  
        return 1;  
    else  
        return n * fatorial(n-1);  
}
```

## Sem Recursão

```
int fatorial (int n){  
    if (n == 0)  
        return 1;  
    else{  
        int i;  
        int f = 1;  
        for(i = 1; i <= n; i++)  
            f = f * i;  
        return f;  
    }  
}
```

# Recursão

- Em geral, formulações recursivas de algoritmos são frequentemente consideradas “mais enxutas” ou “mais elegantes” do que formulações iterativas.
- Porém, algoritmos recursivos tendem a necessitar de mais espaço do que algoritmos iterativos.

# Recursão

- Todo cuidado é pouco ao se fazer funções recursivas.
  - Critério de parada: determina quando a função deverá parar de chamar a si mesma.
  - O parâmetro da chamada recursiva deve ser sempre modificado, de forma que a recursão chegue a um término.

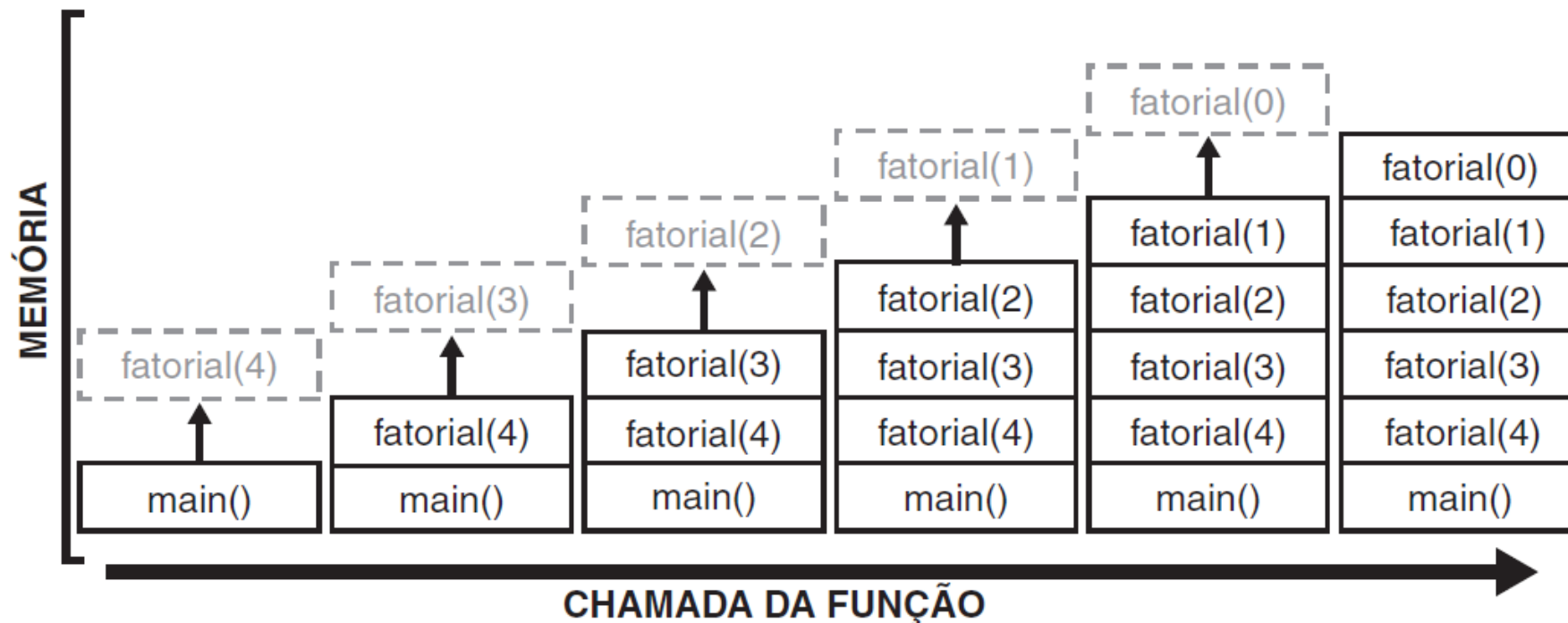
```
int fatorial (int n){  
    if (n == 0) //critério de parada  
        return 1;  
    else /*parâmetro de fatorial sempre muda*/  
        return n*fatorial(n-1);  
}
```



# Recursão

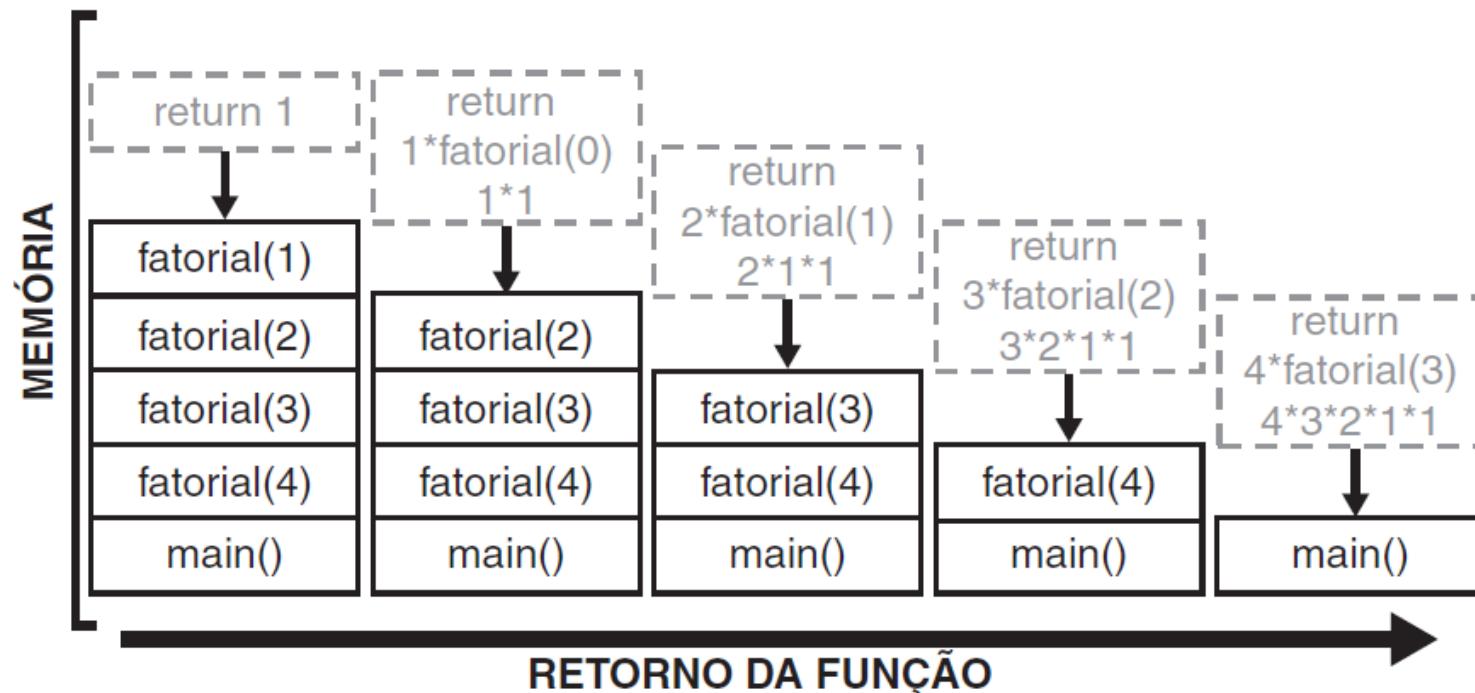
- O que acontece na chamada da função fatorial com um valor como  $n = 4$ ?

```
int x = fatorial(4);
```



# Recursão

- Uma vez que chegamos ao caso-base, é hora de fazer o caminho de volta da recursão.



# Fibonacci

- Essa seqüência é um clássico da recursão
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
- A sequência de Fibonacci é definida como uma função recursiva utilizando a fórmula ao lado
- Sua solução recursiva é muito elegante ...

$$F(n) = \begin{cases} 0, & \text{se } n = 0 \\ 1, & \text{se } n = 1 \\ F(n-1) + F(n-2), & \text{outros casos} \end{cases}$$

# Fibonacci

## Sem Recursão

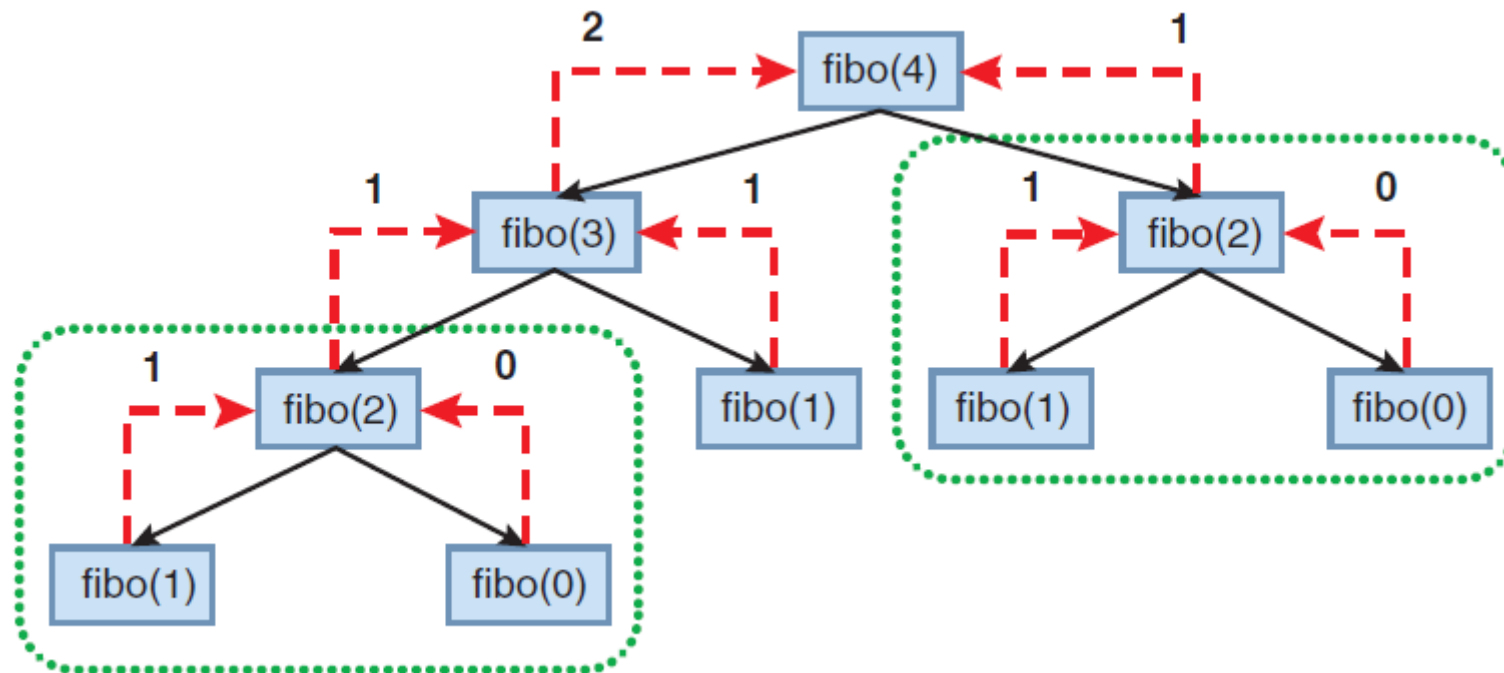
```
int fibo(int n){  
    int i, t, c, a = 0, b = 1;  
    for(i = 0; i < n; i++){  
        c = a + b;  
        a = b;  
        b = c;  
    }  
    return a;  
}
```

## Com Recursão

```
int fiboR(int n){  
    if (n == 0 || n == 1)  
        return n;  
    else  
        return fiboR(n-1) + fiboR(n-2);  
}
```

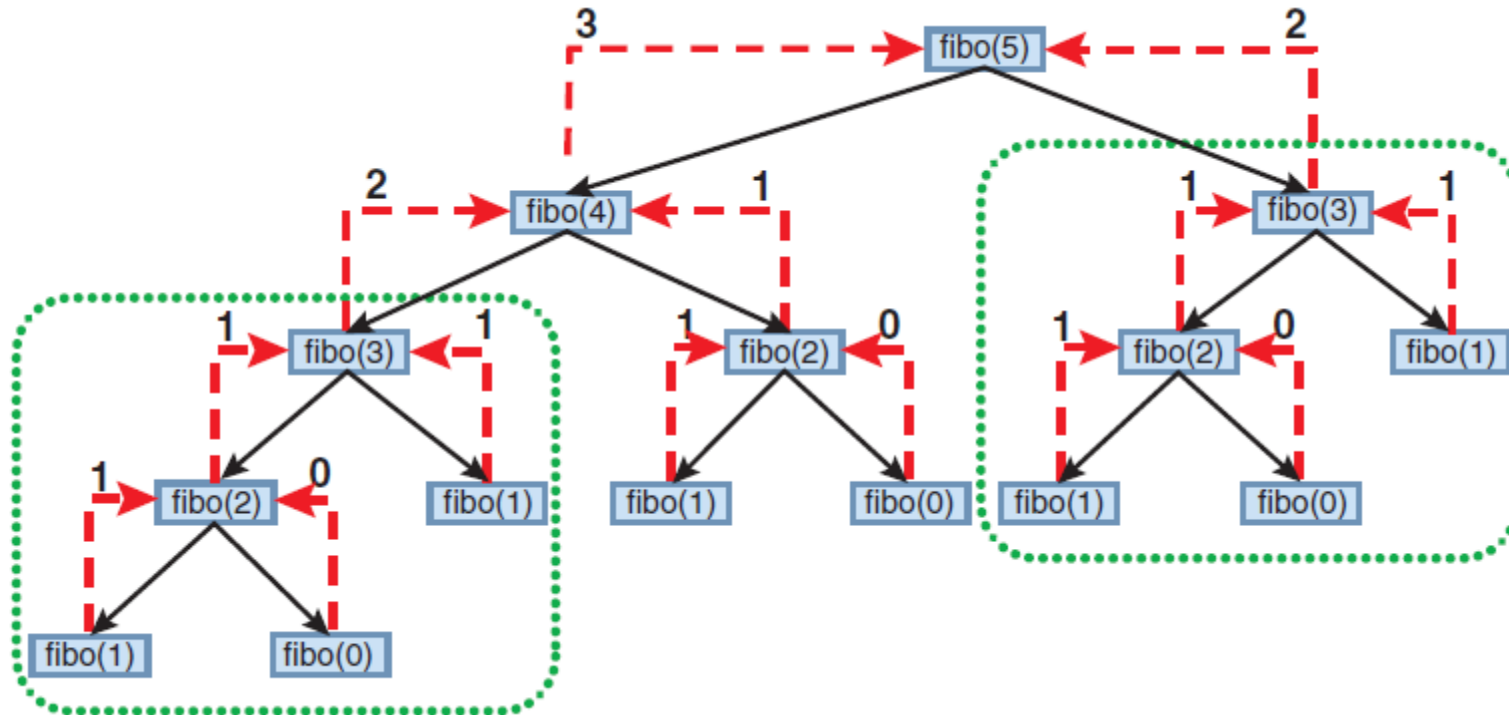
# Fibonacci

- ... mas como se verifica na imagem, elegância não significa eficiência



# Fibonacci

- Aumentando para **fibonacci(5)**



# ALOCAÇÃO DE MEMÓRIA

---

# Problema

- 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.



# Problema

- 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

# Alocação Dinâmica | 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.

# Alocação Dinâmica | Definição

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 | Vantagens

- Quantidade de memória é alocada sob demanda
  - Apenas quando o programa precisa
- Menos desperdício de memória
  - Espaço é reservado até liberação explícita
  - Depois de liberado, estará disponível para outros usos
  - 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`:
  - `malloc()`
  - `calloc()`
  - `realloc()`
  - `free()`

# Função malloc()

- **malloc()**

- Serve para alocar memória e tem o seguinte protótipo:

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

- Funcionamento

- Dado o número de bytes que queremos alocar (**num**), aloca a memória e retorna um ponteiro **void\*** para o primeiro byte alocado.

# Função 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);
```

# Função 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()

- 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;
}
```

# 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

# Função 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;  
}
```

# Função calloc()

- Serve para alocar memória, mas possui um protótipo um pouco diferente:

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

- Funcionamento
  - 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.

# Função calloc() | Exemplo

```
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;  
}
```

# Função realloc()

- Serve para realocar memória e tem o seguinte protótipo:

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

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

# Função 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.
- 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;
}
```

# Função realloc() | Observações

- 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.



# Função free()

- Diferente das variáveis definidas durante a escrita do programa, as variáveis alocadas dinamicamente não são liberadas automaticamente pelo programa.
  - 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);
```

# Função 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.

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

- 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 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 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.

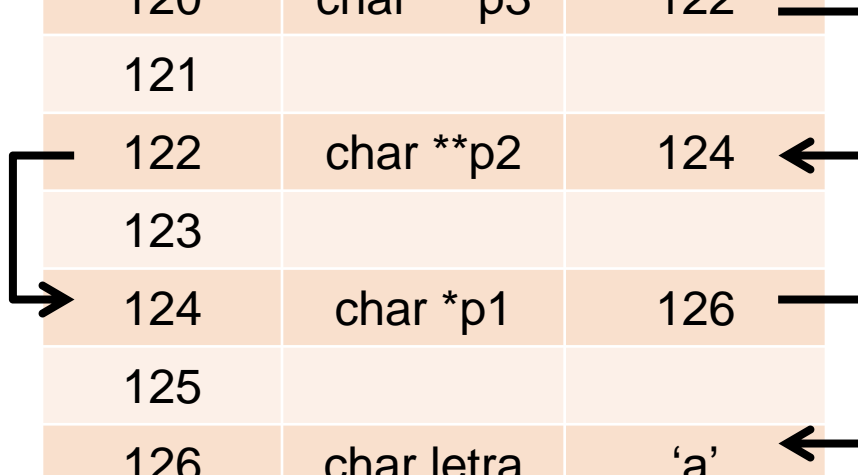
# Alocação de arrays

- 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]
```

# 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		

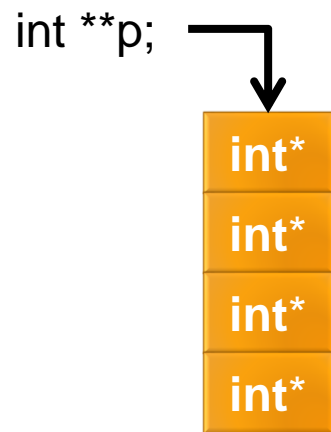


# Alocação de arrays

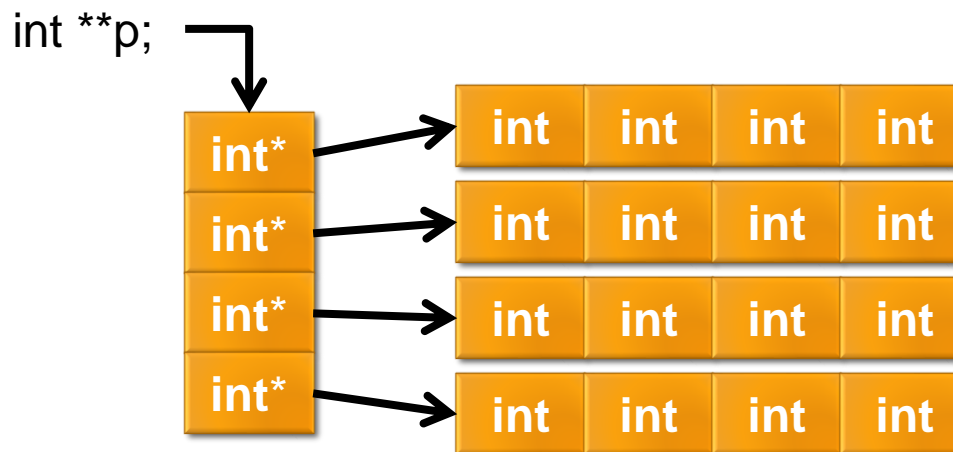
- Em um ponteiro para ponteiro, cada nível do ponteiro permite criar uma nova dimensão no array.

```
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

- Para liberar um array com mais de uma dimensão da memória
  - Liberar a memória alocada em cada uma de suas dimensões
  - Na ordem inversa da que foi alocada

```
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);
```

# Alocação de struct

- Assim como os tipos básicos, também é possível fazer a alocação dinâmica de estruturas.
- As regras são exatamente as mesmas para a alocação de uma **struct**.
- Podemos fazer a alocação de
  - uma única **struct**
  - um array de **structs**

# Alocação de struct

- Para alocar uma única **struct**
  - Um ponteiro para **struct** receberá o **malloc()**
  - Utilizamos o **operador seta** para acessar o conteúdo
  - Usamos **free()** para liberar a memória alocada

```
struct cadastro{
    char nome[50];
    int idade;
};

int main(){
    struct cadastro *cad = (struct cadastro*) malloc(sizeof(struct cadastro));
    strcpy(cad->nome, "Maria");
    cad->idade = 30;

    free(cad);

    return 0;
}
```

# Alocação de struct

- Para alocar um array de **struct**
  - Um ponteiro para **struct** receberá o **malloc()**
  - Utilizamos os **colchetes [ ]** para acessar o conteúdo
  - Usamos **free()** para liberar a memória alocada

```
struct cadastro{
    char nome[50];
    int idade;
};

int main(){
    struct cadastro *vcad = (struct cadastro*) malloc(10*sizeof(struct cadastro));

    strcpy(vcad[0].nome, "Maria");
    vcad[0].idade = 30;

    strcpy(vcad[1].nome, "Cecilia");
    vcad[1].idade = 10;

    strcpy(vcad[2].nome, "Ana");
    vcad[2].idade = 10;

    free(vcad);

    return 0;
}
```

# TIPO ABSTRATO DE DADOS | TAD

---

# Tipo de dado | Definição

- Define o conjunto de valores (domínio) e operações que uma variável pode assumir
  - O tipo **char** da linguagem C suporta valores inteiros que vão de -128 até +127
  - Também suporta operações de soma, subtração etc.
  - Não possui nenhum tipo de estrutura sobre seus valores

# Estrutura de dados | Definição

- Consiste em um conjunto de tipos de dados em que existe algum tipo de relacionamento lógico estrutural
  - É apenas uma forma de armazenar e organizar os dados de modo que eles possam ser usados de forma eficiente
- Na linguagem C temos **array**, **struct**, **union** e **enum**, todas criadas a partir dos tipos de dados básicos



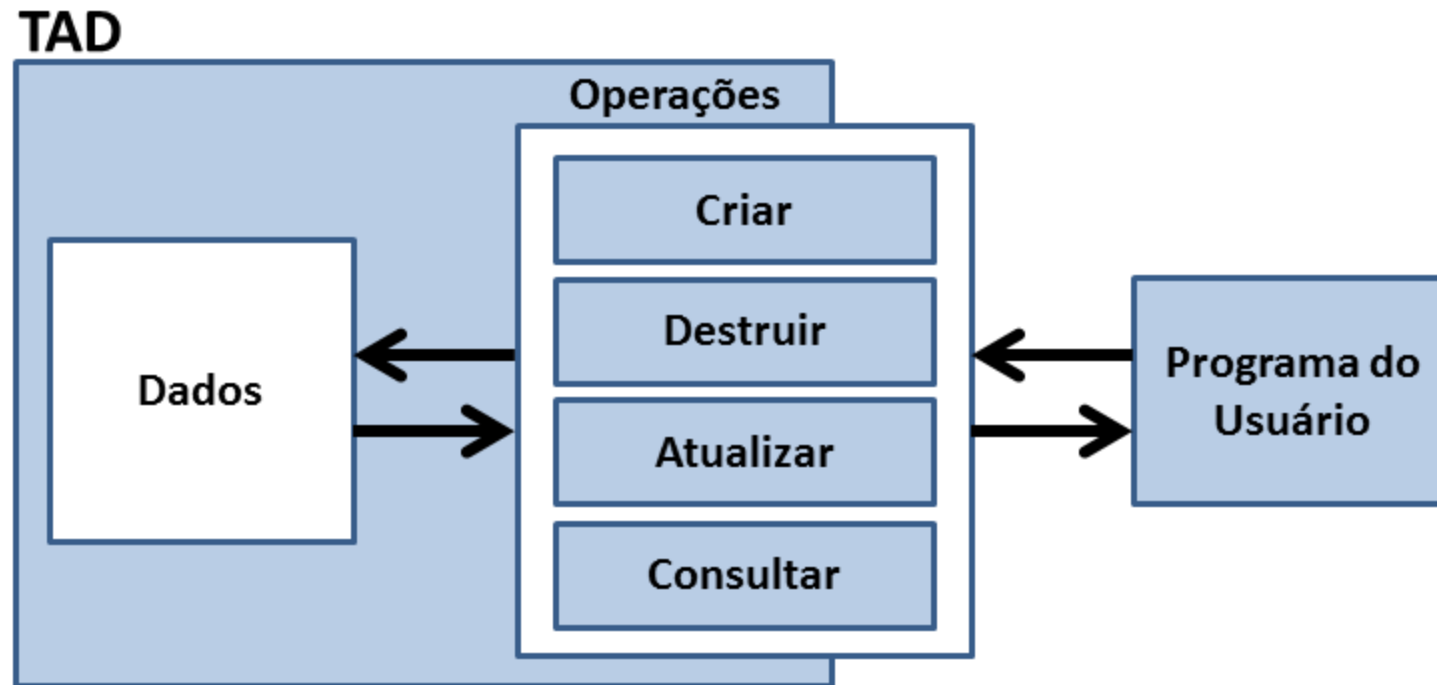
# Tipo abstrato de dados | TAD

- Conjunto de dados estruturados e as operações que podem ser executadas sobre esses dados
  - Basicamente, é um conjunto de valores com seu comportamento definido por operações implementadas na forma de funções.
- Ele é construído a partir dos
  - tipos básicos (**int**, **char**, **float** e **double**)
  - ou dos tipos estruturados (**array**, **struct**, **union** e **enum**)

# Tipo abstrato de dados | TAD

- TADs são entidades puramente teóricos. São usados para
  - simplificar a descrição de algoritmos abstratos
  - classificar e avaliar as estruturas de dados
  - descrever formalmente certos tipos de sistemas
- Surge da necessidade de
  - uma melhor estruturação dos dados
  - especificar quais operações estarão disponíveis para manipular os dados

# Tipo abstrato de dados | TAD



# Tipo abstrato de dados | TAD

- Se você já trabalhou com arquivos na linguagem C, então muito provavelmente você já teve o seu primeiro contato com um TAD
- Trata-se do tipo **FILE**

```
typedef struct{
    int          level;          // nível do buffer
    unsigned     flags;          // flag de status do arquivo
    char         fd;             // descritor do arquivo
    unsigned char hold;          // retorna caractere se sem buffer
    int          bsize;          // tamanho do Buffer
    unsigned char *buffer;        // buffer de transferência de dados
    unsigned char *curp;          // ponteiro atualmente ativo
    unsigned     istemp;          // indicador de arquivo temporário
    short        token;          // usado para validação
} FILE;
```

# Tipo abstrato de dados | TAD

- O tipo FILE é uma estrutura contendo informações sobre um arquivo necessárias para realizar as operações de entrada/saída
  - o descritor do arquivo
  - a posição atual dentro do arquivo
  - indicador de fim de arquivo
  - um indicador de erro
  - etc

# Tipo abstrato de dados | TAD

- A única maneira de trabalhar com arquivos em linguagem C é declarando um ponteiro de arquivo
  - **FILE \*arq;**
- A única maneira de acessar o conteúdo do ponteiro FILE é por meio de das operações definidas em sua interface
  - fopen()
  - fclose()
  - fputc()
  - fgetc()
  - feof()
  - etc

# Tipo abstrato de dados | Vantagens

- Encapsulamento
  - ao ocultarmos a implementação, nós fornecemos um conjunto de operações possíveis para o TAD
  - isso é tudo o que o usuário precisa saber para fazer uso do TAD
  - o usuário não precisa de nenhum conhecimento técnico de como a implementação trabalha para usá-lo, tornando o seu uso muito mais fácil

# Tipo abstrato de dados | Vantagens

- Segurança
  - o usuário não tem acesso direto aos dados
  - Isso evita que ele manipule os dados de uma maneira imprópria
- Flexibilidade
  - podemos alterar o TAD sem alterar as aplicações que o utilizam
- Reutilização
  - a implementação da TAD é feita em um módulo diferente do programa do usuário



# Tipo Opaco

- Se o tipo **FILE** é na verdade uma estrutura, por que não podemos simplesmente declarar uma variável ao invés de um ponteiro para ela?

```
typedef struct{
    int         level;           // nível do buffer
    unsigned    flags;          // flag de status do arquivo
    char        fd;             // descritor do arquivo
    unsigned char hold;         // retorna caractere se sem buffer
    int         bsize;          // tamanho do Buffer
    unsigned char *buffer;      // buffer de transferência de dados
    unsigned char *curp;        // ponteiro atualmente ativo
    unsigned    istemp;         // indicador de arquivo temporário
    short       token;          // usado para validação
} FILE;
```

# Tipo Opaco

- Isso acontece porque o tipo FILE é um tipo opaco do ponto de vista dos usuários da biblioteca
  - apenas a própria biblioteca conhece o conteúdo do tipo e consegue manipulá-lo
  - seus valores só podem ser acessados por funções específicas

# Tipo Opaco

- Um tipo de dado é opaco quando ele é incompletamente definido em uma interface. Em linguagem C, isso é feito utilizando dois arquivos e os princípios de modularização
  - **Arquivo .c:** declara o tipo de dados que deverá ficar oculto do usuário
    - `struct file`
  - **Arquivo .h:** declara o tipo que irá representar os dados ocultos do arquivo .C e que somente poderá ser declarado pelo usuário na forma de um ponteiro
    - `typedef struct file FILE;`

# Criando uma TAD

- Vamos criar um TAD que represente um ponto definido por suas coordenadas x e y
- Ponto.h
  - define o tipo opaco e as funções disponíveis para se trabalhar com o TAD
- Ponto.c
  - as chamadas as bibliotecas necessárias
  - define o tipo do nosso TAD

```
///Arquivo Ponto.h
typedef struct ponto Ponto;
//Cria um novo ponto
Ponto* Ponto_cria(float x, float y);
//Libera um ponto
void Ponto_libera(Ponto* p);
//Acessa os valores "x" e "y" de um ponto
int Ponto_acessa(Ponto* p, float* x, float* y);
//Atribui os valores "x" e "y" a um ponto
int Ponto_atribui(Ponto* p, float x, float y);
//Calcula a distância entre dois pontos
float Ponto_distancia(Ponto* p1, Ponto* p2);

///Arquivo Ponto.c
#include <stdlib.h>
#include <math.h>
#include "Ponto.h" //inclui os Protótipos
struct ponto{//Definição do tipo de dados
    float x;
    float y;
};
```

# Criando uma TAD

- Ponto.c
  - Implementa as funções definidas no arquivo Ponto.h

```
Ponto* Ponto_cria(float x, float y){
    Ponto* p = (Ponto*) malloc(sizeof(Ponto));
    if(p != NULL){
        p->x = x;
        p->y = y;

        return p;
    }

    void Ponto_libera(Ponto* p){
        free(p);
    }

    int Ponto_acessa(Ponto* p, float* x, float* y){
        if(p == NULL)
            return 0;
        *x = p->x;
        *y = p->y;
        return 1;
    }
```

# Estrutura de dados vs. TAD

	Estrutura	Tipo Abstrato de Dado
Definição do tipo	<pre>struct ponto{     float x;     float y; };</pre>	<pre>//Arquivo Ponto.h typedef struct ponto Ponto; //Arquivo Ponto.C struct ponto{     float x;     float y; };</pre> <p><b>Tipo opaco</b></p>
Declaração da variável	<pre>struct ponto p;</pre>	<pre>Ponto *p;</pre>
Acesso ao seu conteúdo	<pre>p.x = 10; p.y = 21;</pre>	<pre>int Ponto_atribui(Ponto* p, float x, float y){     if(p == NULL)         return 0;     p-&gt;x = x;     p-&gt;y = y;     return 1; }</pre>

# Material Complementar | Vídeo Aulas

- Aula 51: Recursão pt.1 – Definição:
  - [youtu.be/T2gTc5u-i1o](https://youtu.be/T2gTc5u-i1o)
- Aula 52: Recursão pt.2 – Funcionamento:
  - [youtu.be/FH5ICr-RVWE](https://youtu.be/FH5ICr-RVWE)
- Aula 53: Recursão pt.3 – Cuidados:
  - [youtu.be/o3MPTEc3LD8](https://youtu.be/o3MPTEc3LD8)
- Aula 54: Recursão pt.4 – Soma de 1 até N:
  - [youtu.be/YEeYk9uEqEI](https://youtu.be/YEeYk9uEqEI)

# Material Complementar | Vídeo Aulas

- Aula 60: Alocação Dinâmica pt.1 – Introdução:
  - [youtu.be/ErOmueylikM](https://youtu.be/ErOmueylikM)
- Aula 61: Alocação Dinâmica pt.2 – Sizeof:
  - [youtu.be/p2ihD9uDZs4](https://youtu.be/p2ihD9uDZs4)
- Aula 62: Alocação Dinâmica pt.3 – Malloc:
  - [youtu.be/iU9CL5d-P5U](https://youtu.be/iU9CL5d-P5U)
- Aula 63: Alocação Dinâmica pt.4 – Calloc:
  - [youtu.be/34uZMXVQd08](https://youtu.be/34uZMXVQd08)
- Aula 64: Alocação Dinâmica pt.5 – Realloc:
  - [youtu.be/vEMTGkANXM4](https://youtu.be/vEMTGkANXM4)
- Aula 65: Alocação Dinâmica pt.6 – Alocação de Matrizes:
  - [youtu.be/W4vbwEJn11U](https://youtu.be/W4vbwEJn11U)



# Material Complementar | Vídeo Aulas

- Aula 01 – TAD (Tipo Abstrato de Dado)
  - <http://youtu.be/bryesHll0vY>
- Aula 02 – Modularização e TAD
  - <http://youtu.be/IKwEQgV6nZk>

# Material Complementar | GitHub

- <https://github.com/arbackes>

## Popular repositories

Livro\_Python

Public

☆ 118    🍴 12

Estrutura-de-Dados-em-C

Public

Códigos fontes de diversas estrutura de dados implementadas em linguagem C

● C    ☆ 49    🍴 4

Estrutura-de-Dados-em-Python

Public

Códigos fontes de diversas estrutura de dados implementadas em linguagem Python

● Python    ☆ 9    🍴 2

Codigos-Fontes-Diversos-em-C

Public

Códigos fontes de diferentes projetos implementados em linguagem C

● C    ☆ 7    🍴 1