

# Linguagem C

Kayky Moreira Praxedes

Fevereiro 2026

## 1 Características da linguagem

Trata-se de uma linguagem procedural de médio nível extremamente rápida (devido à sua proximidade com o processador) com forte ênfase em **gerenciamento manual de memória**, por meio de **estruturas encadeadas** e utilização de **ponteiros**.

Para a execução do código, o arquivo .c tem de ser **compilado** (o compilador mais utilizado é o *gcc*), gerando um arquivo executável. As **ações realizadas pelo programa** são definidas na função principal **main**, sendo a única função obrigatória do programa [01].

### Bibliotecas:

As bibliotecas são códigos que possuem **conjuntos de ferramentas e função prontos** (arquivos terminados em .h). Podem ser adicionadas por meio do comando `#include <nome da biblioteca>` para bibliotecas oficiais ou `#include "nome da biblioteca"` para códigos desenvolvidos pelo próprio programador (na mesma pasta que o *main*).

Para princípios de programação, praticamente todo código irá conter a biblioteca **stdio.h**, responsável pela utilização de elementos de **input** e **output** (como *scanf*, *printf*, etc.), necessário para a esmagadora maioria dos programas.

### Formatação do código:

C não é uma linguagem de indentação necessária, ou seja, o **espaçamentos de qualquer tipo não definem o domínio de ação de estruturas de controle ou funções nem alteram o funcionamento de instruções** ( $a + b = a + b$ ), tendo apenas a função de melhorar a legibilidade e organização do código.

Para delimitar corretamente a execução das **instruções**, utiliza-se o ponto e vírgula (;), que indica o término de uma instrução. Já para **funções e estruturas de controle** (como condicionais e loops), o domínio de ação é definido por meio de blocos delimitados por chaves ({} ) [02].

```
1 #include <stdio.h>
2
3 int main(void){
4
5     // Código
6
7 }
```

## 2 Armazenamento de informações

O armazenamento das informações é feito através de **variáveis**. Essas são **nomes que referenciam endereços de espaços de memória** associados a tipos de dados (que definem o seu tratamento e o espaço reservado a elas).

Toda variável tem de ser **declarada** (criada) antes de sua utilização, o nome sendo escolhido a critério do programador (idealmente um nome explicativo, mas sucinto). Após ser declarada, pode ser **atribuído a ela um valor**, sendo esse um **valor estático**, o **resultado de retorno** de uma função ou mesmo o **valor de outra variável** [03].

```

1 #include <stdio.h>
2
3 int main(void){
4
5     int a;           // Declaração de variável
6     a = 10;         // Atribuição de valor
7     int b = 20;     // Declaração e atribuição direta
8     int c = 30, d = 40; // Declaração múltipla com atribuição
9
10    // Resto do código
11    return 0;
12 }
```

O valor atribuído deve ser **condizente com o tipo da variável**, caso contrário é gerado um erro.

## Variáveis x Constantes:

- **Variáveis:** Seu valor pode ser modificado. Quando essa é declarada dentro de um bloco, são chamadas de **variáveis locais** e só são reconhecidas **dentro dos blocos e sub-blocos onde foram criados**, sendo necessário a utilização de **ponteiros** para a sua alteração em blocos externos (como outras funções).

Se declaradas fora de blocos, são **variáveis globais**. Elas podem ser acessadas e modificadas por **qualquer função dentro do programa**.

- **Constantes:** Uma vez declarado, o seu valor é constante e inalterável. Se forem declaradas dentro de blocos, são **constantes locais**, fora, **constantes globais**, seguindo a mesma regra de validade das variáveis.

## Sombreamento de elementos:

Variáveis locais **não podem ter o mesmo nome no mesmo bloco**. Se um dado global e um local tiverem mesmo nome e forem chamados por uma função, o **dado local que será processado**.

```

1 #include <stdio.h>
2
3 const int constante_global = 10;
4 int variavel_global = 20;
5
6 int main(void){
7
8     const int constante_local = 30;
9     int variavel_local = 40;
10    int variavel_global = 50;      // Sombreamento da variável global
11
12    // Resto do código
13    return 0;
14 }
```

## 3 Tipos de dados

Em C, **todos os tipos básicos de dados são numéricos** essencialmente (todos os dados são *bits* que possuem um especificador de tipo que define como irá ser apresentado e como as operações às modificam) [04] [05].

- **char:** São caracteres dentro do código ASCII [06]. Normalmente elementos *char* possuem 1 *byte* (1 *byte* = 8 *bits*).
- **int:** Números inteiros. Normalmente elementos *int* possuem 2 a 4 *bytes* de espaço de memória reservados.
- **float:** Números racionais. Normalmente elementos *float* possuem 4 *bytes* de espaço reservado de memória.
- **double:** Também são números racionais, mas com o dobro de capacidade do *float* (8 *bytes*).

Algumas funções precisam determinar o tipo de dados que serão trabalhados, (como *scanf*, *printf*, etc.). Para que a função possa identificar cada tipo, esses possuem um “código” (formatador) associado a si (%c para *char*, %d para *int*, %f para *float*. %lf para *double*, etc.) [07].

## 4 Operações

As informações contidas nas variáveis podem ser manipuladas através de operações pré-definidas na linguagem, essas seguindo uma ordem de prioridade [08]. A grosso modo, a ordenação pode ser definida como:

1. Parênteses.
2. Operadores aritméticos, lógicos, comparativos, etc.
3. Operadores de atribuição.

É bom definir explicitamente a ordem das operações por parênteses para evitar *bugs* e comportamentos inesperados.

## 5 Elementos de tomada de decisão

Em C padrão **não existe um tipo dados booleano** (representa se um dado é verdadeiro ou falso) [09]. A linguagem, todavia, entende como verdadeiro qualquer valor diferentes de 0 (independentemente do tipo), caso contrário é falso.

### *if/else:*

Se a condição é verdadeira, realiza-se a ação do *if*. Se for falsa, passa-se para o próximo teste (*else if* ou *else*).

```
1 #include <stdio.h>
2
3 int main(void){
4
5     if(condição_1){
6         // Ação 1
7     }
8     else if(condição_2){
9         // Ação 2
10    }
11    else{
12        // Ação padrão
13    }
14    // Resto do código
15    return 0;
16 }
```

### *Switch case:*

Estrutura que cobre **inúmeras possibilidades de valor** que a condição pode assumir. **Todos os casos no switch necessitam obrigatoriamente um break**, menos o *default* (caso padrão, com *break* facultativo), caso contrário todas as condições abaixo da escolhida também são executadas (*fall-through*).

```
1 #include <stdio.h>
2
3 int main(void){
4
5     switch (condição)
6     {
7         case 10:           // condição = 10
8             // Ação 1
9             break;
10        case 20:          // condição = 20
11            // Ação 2
12            break;
13        case 30: case 40: // condição = 30 ou 40
14            // Ação 3
15            break;
16        default:
17            // Ação padrão
18            break;
19     }
20     // Resto do código
21     return 0;
22 }
```

## **goto:**

Permite o código ser redirecionado para um *label* (elemento que aponta para uma linha específica do código).

```
1 #include <stdio.h>
2
3 int main(void){
4
5     goto LABEL;
6     // Código
7     LABEL:
8     // Resto do código
9     return 0;
10 }
```

O uso desse recurso em códigos é **desencorajado**, já que deixa o **programa desorganizado**, dificultando o *debugging* e a manutenção, além de poder **causar falhas lógicas** (como avançar para áreas do código que utilizam uma variável que devia ter sido declarada mas essa instrução foi pulada).

## **6 Elementos de repetição (*loops*)**

Realizam repetidamente uma ação até que uma condição de parada ou um *break* sejam alcançados.

### ***while* e *do while*:**

Semelhante ao *if*, o *while* realiza uma ação enquanto a sua condição for “verdadeira” (diferente de 0), nem entrando no bloco se a condição for falsa. A única diferença do *while* para o *do while* é que o segundo **realiza a ação antes de entrar no loop**, repetindo-a enquanto a condição for verdadeira (pelo menos uma vez irá executar a ação).

```
1 #include <stdio.h>
2
3 int main(void){
4
5     while (condição_1){
6         // Ação 1
7     }
8
9     do{
10        // Ação 2
11    } while (condição_2);
12    // Resto do código
13    return 0;
14 }
```

### ***for*:**

No seu escopo define-se uma variável (que definirá sua condição) e seu valor de início [10], sua condição de parada e a operação a ser realizada na variável, tudo separado por pontos e vírgulas (;).

```
1 #include <stdio.h>
2
3 int main(void){
4
5     for (int i = 0; i < 10; i++){
6         // Ação
7     }
8     // Resto do código
9     return 0;
10 }
```

### ***break* e *continue*:**

O *break* sai do bloco de execução instantaneamente (válido para *loops* e *switch*). O *continue* ignora o resto das instruções do *loop* a partir dele, começando a próxima repetição (apenas para *loops*).

## 7 Funções

Para se declarar uma nova função em C, ela é tem de ser escrita **acima da função *main***, caso contrário gera um erro de compilação (para poder colocá-la em qualquer lugar é necessário que ela tenha um protótipo), define-se seu **tipo de retorno** (*void* caso não retorne nada), seu **nome** e são declaradas as **variáveis locais** que serão receberão valores na chamada [11].

As funções básicas mais importantes da biblioteca *stdio.h* são o *scanf* (recebe um valor de *input* no terminal e grava esse dado no endereço da variável desejada) [12] e o *printf* (imprime uma mensagem no terminal).

### Protótipos:

Ferramenta que **identifica erros pré-compilando as funções** (seu uso é facultativo e permitem o posicionamento de funções em outros lugares, como abaixo do *main*). Em seu escopo é necessário declarar apenas o tipo das variáveis.

### Comentários:

Os comentários em C são feitos **linha a linha** por meio de barras duplas (//) ou **dentro de um intervalo** delimitado por barras e asteriscos /\* \*/.

```
1 #include <stdio.h>
2
3 /*
4 Comentário com
5 mais de uma linha
6 */
7
8 void funcao(int, double); // Protótipo da função
9
10 int main(void){
11     int a;
12     scanf(" %d", &a);
13     printf("Valor lido: %d\n", a);
14     // Resto do código
15     return 0;
16 }
17
18 void funcao(int inteiro, double racional){
19     // Código da função
20 }
```

### Recursão:

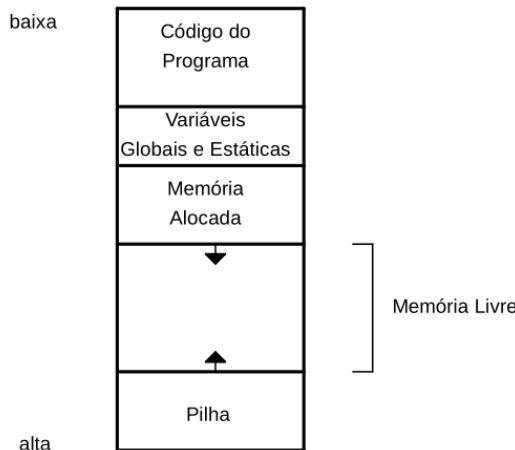
Trata-se de um tipo especial de operação onde uma função realiza uma **chamada de si mesma**, gerando algo semelhante à um *loop*. Quando a condição de parada é atingida, é encerrado o processo recursivo, permitindo que as chamadas retornem seus resultados gradualmente [13].

```
1 #include <stdio.h>
2
3 void fibonacci_recursivo(int antecessor, int atual){
4     printf("%d, ", antecessor);
5     if((atual + antecessor) <= 20) // Chamada recursiva dos termos até 20
6         fibonacci_recursivo(atual, antecessor + atual);
7     else printf("%d\n", atual); // Fim da chamada
8 }
9
10 int main(void){
11     fibonacci_recursivo(1,1);
12     return 0;
13 }
```

## 8 Estrutura da memória (*stack/heap*)

Todo o programa necessita de utilizar a memória para a sua execução (seja para armazenar as instruções, as variáveis, as funções e todos os demais elementos que compõe o programa).

Comummente, a memória é dividida em, da região de memória mais baixa (endereço menor) para a mais alta, área reservada ao **código do programa** (fixo), área reservada à **variáveis globais e estáticas**, a **área dinâmica (*heap*)**, a **memória livre** e a **pilha (*stack*)**, sendo as duas primeiras fixas (definidas desde o início do código, utilizando a mesma quantidade de memória antes e depois sua execução) e as demais variáveis.



- **Stack:** Responsável pelas **informações temporárias das funções** (parâmetros, variáveis locais, endereço de retorno das funções, etc.). O armazenamento de novos dados geralmente é feito colocando um novo bloco de informações (*frame*) no “**topo**” da **pilha** (endereço menor). Ela é **gerenciada automaticamente pelo sistema** (os blocos são adicionados quando necessário e são removidos ao fim de sua execução, liberando a memória).
- **Heap:** Região de **controle manual** da memória por parte do programador, cabendo a ele **controlar a alocação e liberação de memória diretamente**. Ao armazenar um novo dado, geralmente ele é salvo na base do *heap* (passa para um endereço maior).
- **Memória livre:** Região em comum onde *stack* e *heap* armazenam seus dados a depender da necessidade.

## 9 Ponteiros

As variáveis são armazenadas em um **endereço de memória**, o qual pode ser encontrado colocando o prefixo & em seu nome. Ponteiros são elementos especiais que **armazenam em si o endereço de outras variáveis e redirecionam o programa para esse endereço**.

Para criar um ponteiro, basta colocar um asterisco (\*) ao lado do tipo após declarar a variável. Podem ser criados ponteiros para quaisquer tipos, até **ponteiros de um ponteiro**, através de um duplo asterisco (\*\*). E assim por diante.

```
1 #include <stdio.h>
2
3 int main(void){
4
5     int a = 10;
6     int* b;           // b é um ponteiro para int
7     b = &a;          // b recebe o endereço de a
8     int* c = &a;      // Atribuição direta do endereço de a para c
9     int** d;         // d é um ponteiro para ponteiro de int
10    d = &b;          // d recebe o endereço de b => *d = b = &a => **d = a
11
12    // Resto do código
13 }
```

Qualquer alteração feita em *\*b* é armazenada diretamente no endereço de memória de *a*.

## Passagem por valor x Passagem por referência:

No escopo de uma função são geradas variáveis locais (válidas apenas naquele bloco e em seus sub-blocos).

Se é passado uma variável na sua chamada **o valor dessa variável é copiado para a variável local**, (não interage diretamente com a variável original). Esse processo recebe o nome de **passagem por valor**.

Quando é passado o endereço de uma variável, **o endereço é copiado para o ponteiro local da função** (passando a interagir diretamente com a variável original). Esse processo recebe o nome de **passagem por referência**.

```
1 #include <stdio.h>
2
3 void divisao(int, int*); // Cria um objeto int (recebe int) e um ponteiro de int (recebe endereço int)
4
5 int main(void){
6
7     int a = 10, b = 10;
8     divisao(a, &b); // Envia o valor de a e o endereço de b
9     // a continua 10, b vira 5
10    // Resto do código
11    return 0;
12 }
13
14 void divisao(int valor, int* referencia){
15     valor /= 2;           // Copia local de a
16     *referencia /= 2;    // Ponteiro com o endereço de b
17 }
```

## 10 Arrays

*Arrays* ou vetores são conjuntos de tamanho pré determinado fixo de variáveis de mesmo tipo. Para um *array* de tamanho  $n$  seu primeiro elemento está localizado na posição 0, e seu último na posição  $n - 1$ .

### Arrays e ponteiros:

Um *array* pode funcionar como **um ponteiro para o endereço do seu primeiro elemento** (decai para um ponteiro), de modo que a sua utilização como um ponteiro ou o contrário é aceita pela linguagem.

```
1 #include <stdio.h>
2
3 void funcao_com_array(int arr[]){ // Array no escopo
4     // Código da função
5 }
6 void funcao_com_ponteiro(int *ptr){ // Ponteiro no escopo
7     // Código da função
8 }
9
10 int main(void){
11
12     int a;
13     scanf("%d", &a);
14     int array_vazio[5]; // Tamanho não definido e lixo de memória
15     int array_limpo[5] = {0}; // Tamanho definido com os elementos zerados
16     int array1[5] = {1, 2, 3}; // {1, 2, 3, 0, 0}
17     int array2[] = {4, 5, 6}; // Tamanho inferido (3 elementos)
18     int* ptr = array1; // Aponta para o primeiro elemento do array
19     // ptr[n] == array1[n] = *(ptr + n) == *(array1 + n)
20     funcao_com_array(array1);
21     funcao_com_array(ptr);
22     funcao_com_ponteiro(ptr);
23     funcao_com_ponteiro(array2); // Todas as chamadas são válidas
24 }
```

Em geral, múltiplos elementos de um *array* só podem ser inicializados simultaneamente durante sua **declaração**, sendo necessário posteriormente uma **modificação ou atribuição individual** em cada espaço do conjunto (processo feito através de *loops*).

## Matrizes:

Tratam-se de conjuntos de *arrays*. Além das propriedades, em sua declaração é **obrigatório definir o tamanho de todos todas as dimensões de uma vez** (menos no escopo de funções, onde o primeiro colchete ([ ]) pode ser deixado vazio). O mesmo é válido para matrizes de  $n$  dimensões.

```
1 #include <stdio.h>
2
3 void funcao_com_matriz(int matriz[][][10]){} // Matriz no escopo
4 void funcao_ponteiro_array(int* array[]){} // Ponteiro de array no escopo
5 void funcao_ponteiro_duplo(int** pont{}) // Ponteiro duplo no escopo
6
7 int main(void){
8
9     int matriz_1[3][10]; // Lixo de memória
10    int matriz_2[3][3] = {0}; // Todos os valores zerados
11    int matriz_3[3][3] = {1, 2, 3}; // Demais valores zerados
12    int matriz_4[2][5] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
13    int matriz_5[2][5] = {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}};
14
15    // Ponteiro de array
16    int* ptr_array_1[3] = {matriz_3[0], matriz_3[1], matriz_3[2]};
17    int* ptr_array_2[3] = matriz_3;
18    int* ptr_array_3[3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
19    // Ponteiro duplo
20    int** ptr_duplo_1 = ptr_array_1;
21    int** ptr_duplo_2 = matriz_3;
22    funcao_com_matriz(ptr_duplo_1);
23    funcao_ponteiro_array(ptr_array_1);
24    funcao_ponteiro_duplo(matriz_1); // Todas as chamadas são válidas
25
26    // Resto do código
27    return 0;
28 }
```

## String:

São *arrays* de *char*. Também possuem tamanho pré-definido, todavia, seu fim é definido por um sinal ('\0') que pode ser colocado em outras posições, podendo **aumentar ou diminuir o tamanho da string**.

## 11 Alocação dinâmica

Ao se declarar uma variável, esta passa a ocupar um espaço de memória na pilha e é gerenciada automaticamente pelo sistema. Esse processo é chamado de **alocação estática**. Já a **alocação dinâmica** é realizada e gerenciada diretamente pelo programador em cada parte do processo, desde alocar (reservar) o espaço de memória no *heap* [14], até adicionar e/ou modificar os dados na memória e desalocar (liberar) a o espaço depois da sua utilização [15]. As funções de alocação dinâmica estão definidas na biblioteca *stdlib.h* [16].

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void){
5
6     int a = 10, b;
7     // Aloc a memória para int (com lixo de memória)
8     int* ptr = (int *)malloc(sizeof(int));
9     *ptr = 10;
10    // Desaloca a memória alocada antes de reutilizar o ponteiro
11    free(ptr);
12    ptr = &a;
13    scanf("%d", &b);
14    // Array de tamanho variado com lixo de memória
15    int* array1 = (int*)malloc(b * sizeof(int));
16    // Array de tamanho fixo com todos os elementos zerados
17    int* array2 = (int*)calloc(10, sizeof(int));
18    free(array1);
19    free(array2);
20
21    // Resto do código
22    return 0;
23 }
```

## 12 Structs

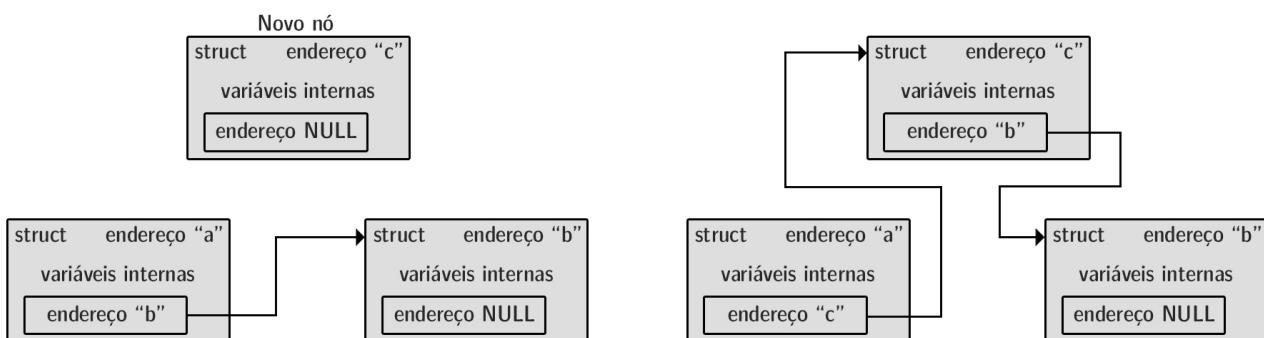
Trata-se de um tipo derivado que pode ser definido como um **grupo de variáveis relacionados entre si**. Simula algo como um “**objeto**” linguagens que suportam *POO*, possuindo atributos [17], mas infinitamente mais primitivos, não suportando funções internas, nem capacidade de realizarem abstrações, encapsulamento e/ou polimorfismo.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  // Definição da "struct carta"
5  struct carta{
6      char *naipe, *face;
7      int valor;
8  };
9  // Define-se um nome fácil de declarar ao invés de declar "struct carta"
10 typedef struct carta carta;
11 typedef struct carta CartaDeBaralho;
12
13 int main(void){
14     // Declaração de uma variável do tipo "struct carta" sem inicialização
15     struct carta carta_generica;
16     // Declaração e inicialização ordenada dos elementos
17     struct carta a = {"Ouros", "Rei", 10};
18     // Declaração e inicialização nomeada (livre)
19     struct carta b = {.face = "As", .valor = 11, .naipe = "Copas"};
20     // Modificação dos campos
21     a.naipe = "Paus";
22     a.valor = 9;
23     a.face = "Nove";
24     // Declarações usando os nomes do typedef
25     carta c = {"Copas", "Dois", 2};
26     CartaDeBaralho d = {"Espadas", "Valete", 10};
27
28 }
```

## 13 Elementos encadeados

Elementos encadeados combinam a utilização de ponteiros, alocação dinâmica e *structs* para criar **conjuntos mais flexíveis, organizados e seguros** (em alguns aspectos) **do que arrays** (esses com a organização envolvendo reescrita, a adição e remoção de elementos trabalhosa e possuindo tamanho invariável após declaração).

Dentro da *struct* existe um **ponteiro para uma *struct* do mesmo tipo que irá apontar para o próximo elemento da sequência**. Dessa maneira, ao adicionar um novo nó à lista, basta “encaixá-lo” entre dois elementos, o endereço do antecessor apontando para ele e ele apontando para o sucessor.



Esse método permite **flexibilidade de disposição dos espaços de memória** (os dados não precisam ser alocados sequencialmente), **tamanho variável das listas** e **nenhuma reescrita** (sem risco de perda de todos os dados em casos de *bugs* na realocação).

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 // Definição da "struct" diretamente com o typedef
4 typedef struct lista {
5     int elemento;
6     struct lista* proximo;      // Ponteiro para o próximo nó
7 } Lista;
8 // Facilitador de sintaxe para ponteiros (vai usar bastante)
9 typedef Lista* ListaPtr;
10
11 int main(void){
12     // Uso de um ponteiro pois será usada alocação dinâmica
13     ListaPtr minha_lista = NULL;
14     int valor;
15     scanf("%d", &valor);
16     while (valor != 0){
17         ListaPtr novo_no = (ListaPtr)malloc(sizeof(Lista));
18         if (novo_no == NULL) return 1; // Falha na alocação
19         // Exemplo de pilha (elemento adicionado no início da lista)
20         novo_no->elemento = valor;
21         novo_no->proximo = minha_lista;
22         minha_lista = novo_no;
23         scanf("%d", &valor);
24     } // Cada nó tem de ser liberado individualmente
25     while(minha_lista != NULL){
26         ListaPtr temp = minha_lista; // Guarda nó atual para liberar depois
27         minha_lista = minha_lista->proximo; // Avança para o próximo nó
28         free(temp); // Liberar a memória do nó atual
29     }
30     return 0;
31 }
```

O encadeamento permite inúmeros tipos de disposições dos dados, a depender da necessidade do programador. Pode-se, por exemplo, criar uma função para organizar automaticamente os novos elementos adicionados em ordem crescente.

```

11 int add_crescente(ListaPtr* lista, int valor) {
12     ListaPtr novo_no = (ListaPtr)malloc(sizeof(Lista));
13     if (novo_no == NULL) return -1;
14     novo_no->elemento = valor;
15     // Lista está vazia ou o novo valor é menor que o primeiro elemento
16     if (*lista == NULL || (*lista)->elemento >= valor) {
17         novo_no->proximo = *lista;
18         *lista = novo_no;
19     } else { // Inserção no meio ou no final
20         // Uso de um ponteiro temporário para percorrer a lista
21         ListaPtr tmp = *lista;
22         while ((tmp->proximo != NULL && (tmp->proximo)->elemento < valor) {
23             tmp = tmp->proximo;
24         }
25         // tmp → novo_no → tmp->proximo (o último existindo ou sendo NULL)
26         novo_no->proximo = tmp->proximo;
27         tmp->proximo = novo_no;
28     }
29     return 0;
30 }
31 int main(void){
32     ListaPtr minha_lista = NULL;      // NULL
33     add_crescente(&minha_lista, 20); // [20]
34     add_crescente(&minha_lista, 10); // [10, 20]
35     add_crescente(&minha_lista, 15); // [10, 15, 20]
36     return 0;
37 }
```

Vale explicitar que quando dois ponteiros referenciam o mesmo endereço de memória, a **alteração feita por um também é vista pelo outro**.

No escopo da função é passado um **ponteiro com o endereço de início da lista**, que só deve ser alterado se for adicionado um novo primeiro item (caso do *if*). Por isso que no *else* foi usado um **ponteiro local** para se deslocar entre os elementos na lista, e não diretamente pelo comando *(\*lista)=(\*lista)->proximo* (preservando assim o endereço de início, que **seria alterado com o uso do comando**, e sendo possível manipular as informações dos endereços ligados à frente).

## Estruturas de dados:

A partir do encadeamento de elementos é possível construir algoritmos mais complexos e funcionais para a organização dos dados. As principais estruturas de dados são:

- **Filas:** Estruturas **sequenciais** com adição de **novos nós ao final** (dados recentes são mais difíceis de acessar, enquanto dados antigos mais fáceis).
- **Pilhas:** Estruturas **sequenciais** com adição de **novos nós no início** (dados recentes estão mais próximos, enquanto dados antigos mais distantes).
- **Árvores:** Estruturas **não sequenciais** de dados, que permite **ramificações a partir de uma base**. A depender do tipo da árvore, novos dados podem ser adicionados em posições intermediárias ou apenas nas “folhas” (fim das ramificações).

Pode-se pensar em algo semelhante à estrutura de um computador, onde uma pasta pode conter várias subpastas, permitindo uma organização por categorias, não sendo necessário passar por uma enormidade de arquivos anteriores para acessar o desejado).

## 14 Arquivos externos

A linguagem C possui um conjunto de ferramentas que permite a **leitura e escrita** em arquivos de maneira **persistente** em arquivos externos.

A partir da criação de um ponteiro para arquivos (*FILE\**), é possível a **navegação dentro de um arquivo** através da atribuição *ponteiro = fopen("nome do arquivo.tipo", "tipo de acesso")* [18]. Posteriormente, para **encerrar o uso do arquivo** (garantir o salvamento dos dados), usa-se o comando *fclose(ponteiro)*.

Os arquivos podem ser gravados e lidos em dois tipos diferentes:

- **Texto:** Todo conteúdo é *string*, necessitando de uma **conversão para o formato de dado desejado** para operação. Necessita de muita organização na gravação dos dados para evitar que ocorram *bugs* de leitura por conversões inválidas.

```
1 #include <stdio.h>
2
3 int main(void) {
4     int conta[100] = {0};
5     double saldo[100] = {0};
6     FILE* filePointer;
7     // Tenta abrir o arquivo para a leitura de texto (.txt, .dat, etc.)
8     if((filePointer = fopen("dados.dat", "r")) == NULL)
9         return -1; // Erro ao abrir o arquivo
10    for(int i = 0; !feof(filePointer); i++){ // Lê até o final (EOF)
11        // Os dados estão organizados: int double int double ...
12        // São separados por um caractere de espaço ou de nova linha.
13        // Leitor de texto (lê e avança o ponteiro para a próxima sequencia)
14        fscanf(filePointer, "%d %lf", &conta[i], &saldo[i]);
15        // Espaço antes do %d para a limpeza do buffer
16    }
17    fclose(filePointer); // Encerra o processo (garante salvamento dos dados)
18    // Tenta abrir o arquivo para a escrita de texto
19    if((filePointer = fopen("novo.txt", "w")) == NULL) return -1;
20    for(int j = 0; conta[j] != 0 ; j++)
21        // Escritor de texto (escreve e passa para o proximo espaço/linha)
22        fprintf(filePointer, "%d - %.1lf\n", conta[j], saldo[j]);
23    // Os dados ficam organizados: int - double int - double ...
24    fclose(filePointer);
25    // Tenta abrir o arquivo para a leitura e append de texto
26    if((filePointer = fopen("novo.txt", "a+")) == NULL) return -1;
27    // Lê o conteúdo (cuidado com a formatação) e anexa os dados no final
28    fscanf(filePointer, "%d - %lf", &conta[20], &saldo[20]);
29    fprintf(filePointer, "%d - %.3lf\n", conta[20], saldo[20]);
30    fclose(filePointer);
31    return 0;
32 }
```

- **Binário:** A informação é **armazenada numericamente**, operando em blocos de memória (os valores permanecendo os mesmos, não sendo convertidos em *strings*), permitindo inclusive o armazenamento e leitura de *structs* (deixando os dados mais organizados e blindados contra erros e *bugs*). Tende a ser mais eficiente que o texto para processamento.

```

1  #include <stdio.h>
2
3  typedef struct {
4      int conta;
5      double saldo;
6  } Dados;
7
8  int main(void) {
9      Dados dados[4] = {{1001, 1000.50}, {1002, -505.10}, {1010, 27}, {0, 0}};
10     Dados lidos[4] = {{0, 0}};
11     FILE* filePointer;
12     // Tenta abrir o arquivo para a escrita de binário
13     if((filePointer = fopen("dados.bin", "wb")) == NULL) return -1;
14     for(int i = 0; dados[i].conta != 0; i++)
15         /* fwrite(endereço do dado, tamanho de cada objeto, quantos dados
16            escrever de uma vez, ponteiro do arquivo) */
17         fwrite(&dados[i], sizeof(Dados), 1, filePointer);
18     /* Mesmo que: fwrite(dados, sizeof(Dados), 3, filePointer);
19        que pegaria dados[0], dados[1] e dados[2]*/
20     fclose(filePointer);
21     // Tenta abrir o arquivo para a leitura de binário
22     if((filePointer = fopen("dados.bin", "rb")) == NULL) return -1;
23     for(int i = 0; !feof(filePointer); i++)
24         fread(&lidos[i], sizeof(Dados), 1, filePointer);
25     fclose(filePointer);
26     return 0;
27 }
```

Existem mais funções disponíveis para propósitos variados, como o *fputs*, que permite adicionar uma *string* direto, *fgets* que lê uma linha inteira de uma vez, *rewind*, que recoloca o ponteiro no início do arquivo, etc.

## 15 Pré-processamento

A linguagem permite a **realização de algumas ações antes de o programa ser compilado** por meio de comando *#ação*. As aplicações vão desde a **inclusão de outros arquivos** até a **definição de constantes simbólicas** (globais), **compilação condicional**, etc. [19]

```

1  /* #include:
2   Inclui arquivos para a execução do programa */
3  #include <stdio.h>      // Biblioteca padrão
4  #include "mylib.h"       // Biblioteca autoral
5
6  /* #define e #undef:
7   O primeiro define e copia macros (constantes e funções pré-processadas)
8   e o segundo os apaga (esse funciona para diretivas locais e externas)*/
9  #define DEZ 10           // Diferentes tipos
10 #define NOME_PROGRAMA "Calculadora Financeira"
11 #define PROGRAMA NOME_PROGRAMA // Dois macros com o mesmo valor
12 #define AREA_CIRCULO(raio) (PI * (raio) * (raio))
13 //double area_circulo(double raio){ return PI * (raio) * (raio); }
14 #undef DEZ              // Definido em mylib.h (biblioteca externa).
15
16 // #if: Verifica a condição de uma diretiva
17 #ifndef PI              // if (!defined(NOME)), também tem if defined(NOME)
18     #warning "PI não definido" // Mensagem de warning no terminal
19     #define PI 3.14159265
20 #else
21     #if (DEZ > 20)        // Só suporta comparações int
22         #undef DEZ        // Definido localmente.
23     #elif (DEZ <= 2)       // (DEZ <= 2.0) ou (PI <= 2) daria erro
24         #define PI 3.14
25     #endif                // Delimita a atuação do condicional
26 #endif                  // Colocado após o último #if, #elif, #else
```

## Compilação de múltiplos arquivos:

Par a compilação de múltiplos códigos interligados em C, é necessário a criação de **um arquivo fonte** (arquivo *.c* contém a implementação das funções) **arquivo cabeçalho/biblioteca** (arquivo *.h* que conterá o protótipo das funções, estruturas e variáveis **públicas** [20]). Para incluir a biblioteca, o comando é `#include "caminho/do/arquivo.h"` (caso estejam na mesma pasta, basta colocar o nome do arquivo).

```
1  /* mylib.h */
2  #ifndef MYLIB_H
3  #define MYLIB_H
4
5  double media(double*, int); // Protótipo de funções
6  #define DEZ 10 // Constantes públicas
7  typedef struct { // Estruturas públicas
8      int x;
9      int y;
10 } Ponto;
11 #endif

1  /* mylib.c */
2  #include "mylib.h"
3
4  double media(double* valores, int quantidade){ // Declaração da função
5      double total = 0;
6      for(int i = 0; i < quantidade; i++) {
7          total += valores[i];
8      }
9      return total / quantidade;
10 }

1  /* Arquivo.c */
2  #include <stdio.h> // Biblioteca padrão
3  #include "mylib.h" // Biblioteca autoral
4
5  int main(void){
6      double* valores = {DEZ, 20, 25, 30}; // Macro de mylib.h
7      media(valores, 4); // Função de mylib.h
8      Ponto ponto = {10, 20}; // Estrutura de mylib.h
9      return 0;
10 }
```

## 16 Programa pelo terminal

O **fluxo padrão de I/O** (*input* e *output*) em C é o **terminal**, mas, utilizando ferramentas na **linha de comando** (varia em sintaxe a depender do sistema operacional), é possível **redirecionar esse fluxo para arquivos** (*.txt* por exemplo) e/ou até outros programas.

- **Redirecionamento de entrada:** *executável < entrada.tipo*. Todos os elementos de *input* do código, que seriam obtidas pelo terminal, são passados em um arquivo [21].
- **Redirecionamento de saída:** *executável > saída.tipo*. Todos os elementos de *output* do código, que seriam impressos no terminal, são passados em um arquivo.
- **Piping:** *executável\_output | executável\_input*. Todos os elementos de *output* do código primeiro código servem como *input* para o segundo programa.

Além de redirecionamento, podem ser adicionados **parâmetros no escopo do main** para a **passagem de parâmetros pelo terminal**.

```
1  #include <stdio.h>
2
3  int main(int argc, char* argv[]){
4
5      // Resto do código
6      return 0;
7 }
```

Se o programa for executado pelo terminal com esse escopo, os elementos, separados por espaço após ele são salvos no *array de strings argv*, e a quantidade de elementos é salva em *argc* [22].

# Apêndice A

## Características da linguagem

[01] - Por padrão, o tipo adotado no *main* é **int**, por conta da padronização de uso do **valor de retorno 0 como indicativo que a execução ocorreu de maneira bem sucedida**.

[02] - A delimitação por chaves é obrigatória **apenas para funções**. Estruturas de controle sem chaves **executam apenas a instrução imediatamente subsequente**. Boas práticas de programação recomendam **sempre o uso de chaves**, a fim de evitar comportamentos inesperados.

## Armazenamento de informações

[03] - Enquanto nenhum valor for atribuído à variável após ela ser declarada, ela não estará vazia, mas sim conterá **lixo de memória** (dados soltos que se encontravam no endereço da variável). Desse modo, **é necessário atribuir, mesmo que um valor nulo, um valor à variável** antes de utilizá-la.

## Tipos de dados

[04] - Os dados são essencialmente numéricos, mas o tratamento não é o mesmo para as funções, possuindo suas peculiaridades. O elemento nulo que cada um aceita, por exemplo, é diferente, sendo 0 para elementos numéricos, '\0' para *char* e *strings*, *NULL* para ponteiros, etc.

[05] - Existem métodos para converter certos tipos de dados em outros.

- **Conversão implícita:** Automática pelo compilador. Normalmente usado para **converter um tipo menor para um tipo maior** (*float* para *double*, por exemplo).
- **Conversão explícita:** Adição do prefixo (*novo tipo*) na frente da variável pelo programador. Normalmente usado para **converter um tipo maior para um tipo menor** (*double* para *int*, por exemplo).

```
1 #include <stdio.h>
2
3 int main(){
4     // Conversões implícitas comuns
5     //Atribuição:
6     char c = 'A';           // char (65 na tabela ASCII)
7     int i = c;             // char → int implícito (int 65)
8     float f = i;           // int → float implícito (float 65.0)
9     double d = f;          // float → double implícito (double 65.0)
10    // Operações aritméticas:      Operação   Pré Op   Resultado  Atribuição
11    double somal = i / 10;     // 65/10 →   65/10 →   6 →       6.0
12    double soma2 = i / 10.0;   // 65/10.0 → 65.0/10.0 → 6.5 →   6.5
13    printf("Implícitas: %c -> %d -> %.2f -> %.2f\n", c, i, f, d);
14    printf("Soma 1: %.2lf, Soma 2: %.2lf\n", somal, soma2);
15
16    // Conversões explícitas comuns
17    // Casting para tipo menor (perder precisão):
18    double pi = 3.1415926535;
19    int inteiro = (int)pi;
20    // Mudança de ponteiros:
21    int* ptrInt;
22    char* ptrChar = (char*) ptrInt;
23    // Tratamento com unsigned
24    unsigned int u = 10;
25    int b = -5;
26    if(b < (int)u) /* Faz uma ação */;
27    /* Sem a conversão, ambos são tratados como unsigned. Assim, em binário:
28    -5 = 11111111 11111111 11111111 11111011 = (unsigned) 4294967291 > 10 */
29
30 }
```

O prefixo *unsigned* **converte o bit de sinal de uma variável em um bit de valor**. Para uma variável com o valor  $-5$ , em binário (1) 1111111 11111111 11111111 11111011 (primeiro bit de sinal), o seu valor *unsigned* seria 4.294.967.291.

[06] - Como a tabela ASCII associa códigos numéricos aos caracteres, **operações aritméticas como adição e subtração podem ser realizadas com os elementos *char*** (especialmente útil para a formatação de caracteres e para cifragem).

[07] - Os tipos de dados, como comentado anteriormente, também delimitam o espaço de memória máximo a ser alocado na variável. Por vezes, para tipos numéricos, esse **espaço de memória não é suficiente**, de modo que algumas operações levam à um *overflow* ou *underflow* (resultado aritmético incorreto pois não havia memória o suficiente para armazenar o valor completo).

Para sanar esse problema, um recurso é adicionar o prefixo *long* ao tipo das variáveis (permitindo a utilização de um **espaço de memória maior** pelas variáveis). O código da variável passa a ter um *l* após à porcentagem (%) (*long int* → *%ld*, por exemplo).

## Operações

[08] - Tabela completa da ordem de operações:

1. **Parênteses ( )**.
2. **Elementos que acessam valores** (acesso a arrays ([ ]) e chamadas de função).
3. **Operadores unários** (inversor de sinal (-), NOT lógico (!), NOT bit a bit (^), incremento (++) , decremento (--), operador de endereço (&), desreferenciação de ponteiros (\*), operador *sizeof* e mudança de tipo (também chamado de *casting*) ((*new.type*))). São avaliados da direita para a esquerda
4. **Operadores aritméticos** (multiplicação (\*), divisão (/) e resto da divisão (%)) possuem prioridade maior que adição (+) e subtração (-)).
5. **Deslocamento de bits** (esquerda (<<) e à direita (>>)).
6. **Operadores relacionais** (maior (>), maior ou igual (>=)), menor (<), menor ou igual (<=), igual (==) e diferente (! =)).
7. **Operadores bit a bit** (*AND* bit a bit (&), *XOR* bit a bit (^), *OR* bit a bit (|)).
8. **Operadores lógicos** (*AND* lógico (&&) e *OR* lógico (||)).
9. **Operador condicional (?:)**. Esse funciona da seguinte maneira:  
**(condição) ? (valor se verdadeiro) : (valor se falso)**.
10. **Operadores de atribuição** (simples (=)) e compostas (+ =, - =, \* =, / =, % =, & =, | =, ^ =, <<=, >>=). Esses operadores são avaliados da direita para a esquerda.

Os operadores unários de incremento e decremento, quando à esquerda da variável, **realizam a operação e depois retornam o valor**. Quando à direita, **primeiro retornam o valor depois realizam a operação**.

## Elementos de tomada de decisão

[09] - Existe uma biblioteca (*stdbool.h*) que permite a utilização de elementos do tipo booleano (podendo assumir os valores “true” ou “false”), o que permite um código mais simples no quesito de entendimento humano.

## Elementos de repetição (*loops*)

[10] - A variável não tem de ser declarada no escopo, podendo ser declarada antes. Caso seja criada no escopo, ela será uma variável local no *for* (só existirá dentro dele).

## Funções

[11] - A quantidade de variáveis presente no escopo normalmente é pré-definido, todavia, existe uma biblioteca (*stdarg.h*) que permite a utilização de uma quantidade variável de parâmetros na função.

```
1 #include <stdio.h>
2 #include <stdarg.h>
3
4 // É preciso PELO MENOS UM argumento fixo
5 double media(double total, int i, ...){
6     // Declara um "ponteiro" para a lista de argumentos variáveis
7     va_list pointer;
8     // Coloca o ponteiro no último argumento fixo (nesse caso o i)
9     va_start(pointer, i);
10    // Passa por cada argumento a partir de i até o final da lista
11    for (int j = 0; j < i; j++)
12        // Acessa o próximo argumento da lista (tamanho de um double)
13        total += va_arg(pointer, double);
14    va_end(pointer);    // Zera o ponteiro (evita erros de acesso à memória)
15    return(total / i);
16 }
17
18 int main(void){
19     double w = 38.5, x = 22.5, y = 1.7, z = 10.2;
20     media( 0, 2, w, x );
21     media( 0, 3, w, x, y );
22     media( 0, 4, w, x, y, z );
23     return 0;
24 }
```

[12] - Principalmente ao se mexer com *strings*, o *scanf* pode acabar funcionando de maneira incorreta por conta do **buffer estar cheio antes de sua execução** (esse sendo a área de memória usada para guardar dados provisoriamente, enquanto eles estão sendo processados). Para contornar esse problema, dentro as aspas basta **colocar um espaço antes do identificador do tipo** a ser lido. Também pode ser usadas outras funções como um *getchar* vazio ou um *fgets* para o mesmo propósito.

[13] - Apesar de chamadas recursivas conseguirem facilitar consideravelmente algumas operações em códigos, seu uso deve ser cuidadoso, visto que o processo deixa **variáveis alocadas e ativas dentro de cada recursão**. Para uma função com  $n$  variáveis, cada chamada recursiva adiciona um novo *frame* à *stack* (criando mais  $n$  variáveis). O uso prolongado gera **perda de eficiência no código**. Se a memória for limitada, pode ocorrer um *stack-overflow* (memória completamente cheia).

## Alocação dinâmica

[14] - Em casos reais, é importante criar testes para **verificar se a alocação ocorreu corretamente**. Para isso, basta verificar se o ponteiro é diferente de *NULL* (caso contrário contém o endereço).

[15] - Caso a memória não seja liberada, ela **permanece alocada** (marcada como se estivesse ocupada), se tornando inacessível e não podendo ser utilizada novamente. Esse fenômeno recebe o nome de **vazamento de memória**.

[16] - As funções de alocação (*malloc* e *calloc*) por padrão retornam um ponteiro do tipo (*void\**), mas a **conversão é feita automaticamente**. Dessa maneira a explicitação do tipo de ponteiro é facultativo em C (em C++ é obrigatório), e seu uso é até desencorajado, pois pode causar comportamentos inesperados e *bugs* (como quando se tenta usar as funções sem importar a biblioteca *stdlib.h*).

## Structs

[17] - **Não é possível pré-definir valores de variáveis da structs em sua definição**. Essas só podem ser especificadas durante ou após declaração de uma variável do tipo da *struct*.

## Arquivos externos

[18] - Os modos de acesso são para **arquivos de texto** são:

- **r**: Abre o arquivo e permite **apenas a sua leitura**.
- **w**: Cria ou sobrescreve um arquivo, permitindo **apenas a escrita**.
- **a**: Abre ou cria (se não existir) um arquivo **apenas para anexar um conteúdo ao seu final**.
- **r+**: Abre um arquivo, permitindo **leitura e/ou escrita** em seu interior.
- **w+**: Cria ou sobrescreve um arquivo, **permitindo leitura e/ou escrita**.
- **a+**: Abre ou cria (se não existir) um arquivo para sua **leitura e/ou anexar um conteúdo ao seu final**.

Os equivalentes para **arquivos binários** são, respectivamente: rb, wb, ab, rb+, wb+, ab+.

## Pré-processamento

[19] - Existem *macros* pré-definidos na linguagem C, os mais usados sendo:

- **\_LINE\_**: *int* que contém o número da linha atual do código.
- **\_FILE\_**: *string* que contém o nome do arquivo.
- **\_DATE\_**: *string* que contém a data atual no modelo *Mmm dd aaaa* (Feb 7 2026).
- **\_TIME\_**: *string* que contém a hora atual no modelo *hh:mm:ss* (13:22:56).

[20] - Variáveis e constantes de um arquivo podem ser declaradas como *auto*, *static* e *extern*.

- **auto**: O tipo padrão implícito quando uma variável é declarada em uma função. O **seu valor só está salvo na memória durante a execução de sua função**.
- **static**: Define que uma variável terá o seu **espaço de memória alocado durante toda a execução do programa**. Também **impede que a variável seja acessada por códigos externos** (válido também para elementos auto).
- **extern**: Define que **uma variáveis e constantes podem ser utilizadas por em todo o programa** (códigos locais e externos). É o tipo implícito para elementos globais e **não pode ser usado dentro de funções**.

```
1 #include <stdio.h>
2
3 int contExtern = 0;           // extern int contExtern = 0;
4
5 void inicializacao() {
6     static int contStatic = 0;
7     int contAuto = 0;          // auto int contAuto = 0;
8     contExtern++, contStatic++, contAuto++;
9     printf("%d, %d, %d\n", contExtern, contStatic, contAuto);
10 }
11
12 int main() {
13     inicializacao(); // 1, 1, 1
14     inicializacao(); // 2, 2, 1
15     inicializacao(); // 3, 3, 1
16     return 0;
17 }
```

## Programa pelo terminal

[21] - Tendo um *.txt* com o conteúdo (“10 20 30 0 10”) que será redirecionado como entrada para o seguinte código:

```
1 #include <stdio.h>
2
3 int main(void) {
4     int a = 1, total = 0;
5     while (a != 0){
6         scanf(" %d", &a);
7         total += a;
8     }
9     printf("O total da soma é: %d\n", total);
10    printf("Mais um valor para a soma: ");
11    scanf(" %d", &a);
12    printf("O total da soma é: %d\n", total + a);
13
14 }
```

Considerando o conteúdo do *.txt* que será redirecionado como entrada para o código:

- “10 20 30 0 10”: Será executado o *scanf* do *loop* (até o 0), o primeiro *printf* e a próxima instrução receberá o valor 10, imprimindo o segundo *printf*.
- “10 20 30 0”: o segundo *scanf* receberia o valor de **EOF** (*End Of the File*). Seria gerado um erro no *scanf* (ele não alteraria o valor da variável). Isso pode ocorrer tanto no **redirecionamento de entrada** quanto no **piping**.
- “10 20 30”: Ocorreria o mesmo problema no *scanf* do caso anterior. Todavia, como o valor de *a* não é alterado, seria gerado um **loop infinito**.

[22] - Se não houver nenhum elemento após o nome do executável no terminal, *argc* = 1 e *argv[0]* está vazio (armazena o quebra linha ‘\n’ ao executar).

## Fontes

1. DEITEL, Harvey M.; DEITEL, Paul J. Como programar em C. 2. ed. Rio de Janeiro: LTC, 1994.
2. DEITEL, Harvey M.; DEITEL, Paul J. Java: como programar. 10. ed. São Paulo: Pearson, 2017.
3. ZIVIANI, Nivio. Projeto de algoritmos com implementações em Pascal e C. 4. ed. São Paulo: Pioneira, 1999.
4. BATISTA, Natália Cosse. Ponteiros e alocação dinâmica de memória. 2022. 40 f. slides (PDF) da disciplina Algoritmos e estruturas de dados. Centro Federal de Educação Tecnológica de Minas Gerais (CEFET-MG), 2025.
5. PEIXOTO, Daniela Cristina Cascini. Disciplina: Lógica de programação. Curso de graduação em Engenharia de Computação – CEFET-MG, 2024.
6. CAMPOS, Luciana Maria de Assis. Disciplina: Programação orientada a objetos. Curso de graduação em Engenharia de Computação – CEFET-MG, 2024.
7. BATISTA, Natália Cosse. Disciplina: Algoritmos e estruturas de dados. Curso de graduação em Engenharia de Computação – CEFET-MG, 2025.