Programação em C

Kayky Moreira Praxedes

Fevereiro 2026

1 Características da linguagem

C é uma linguagem procedural de médio nível (possui abstrações, mas possui muitos recursos de gestão manual do hardware) extremamente rápida (devido à proximidade com a *CPU*), com muita presença em sistemas operacionais e embarcados, e programação de alto desempenho.

Para a execução do código, **o arquivo *.c* tem de ser compilado** (o compilador mais utilizado é o *gcc*), gerando um arquivo executável.

A função principal do programa é o *main*, responsável por definir as ações realizadas pelo programa (função ativa) [01], as demais funções servindo como ferramentas (função passiva).

A biblioteca *stdio.h* é utilizada em muitas aplicações de C, pois possui funções para a utilização *input* e *output* (necessário para a esmagadora maioria dos programas).

Formatação do código:

Espaçamentos e comentários não alteram diretamente funcionamento do código (apenas melhoram a legibilidade e organização do programa). **Instruções são delimitadas com um ponto e vírgula (;), e funções e estruturas de controle** (blocos como condicionais e *loops*) **tem o seu conteúdo delimitados por chaves ({ })** [02].

```c
#include <stdio.h>  // Biblioteca para o uso de elementos de input e output

/* Comentário com
mais de uma linha */
// Comentário com apenas uma linha
int main(void){ // Função de execução do programa com retorno int e sem argumentos
    printf("Hello, World!\n"); // Instrução do stdio.h que imprime uma mensagem no terminal: Hello, World!
    return 0; // Delimitação de uma instrução (retorno) por ponto e vírgula (;) ao seu final
} // Domínio da função delimitado por chaves ({})
```

2 Tipos de dados

O tipo de dado indica as propriedades de uma variável, restringindo algumas operações [03] e indicando o espaço de memória reservado [04] (variando a depender da arquitetura). Em C, todos os dados são essencialmente numéricos (números binários), variando o especificador de tipo. Essa propriedade facilita operações de tipos diferentes, bem como sua conversão [05].

- ***char*:** Caracteres (dentro da tabela ASCII) [06]. Normalmente a memória reservada é de 1 *byte* (8 *bits*).
- ***int*:** Números inteiros. Normalmente a memória reservada é de 2 a 4 *bytes*.
- ***float*:** Números racionais. Normalmente a memória reservada é de 4 *bytes*.
- ***double*:** Também são números racionais, mas com o dobro de capacidade do *float* (nesse caso, 8 *bytes*).

Cada tipo possui um formatador associado a si (*%c* para *char*, *%d* para *int*, *%f* para *float*. *%lf* para *double*, etc.), esse servindo para definir explicitamente o tipo de dado na extração e inserção um em/para uma *string* (evitar erros na conversão).

3 Armazenamento de informações

O armazenamento das informações voláteis é feito através de variáveis (em C são rótulos para endereços de espaços de memória associados a tipos). Primeiro, declara-se a variável (aloca seu espaço de memória), depois atribui-se um valor a ela (condizente com o tipo da variável), podendo ser um valor direto (explícito), ou indireto (resultado de retorno de uma função, valor de outra variável, etc.) [07].

```c
#include <stdio.h>

int main(){ // void implícito nos argumentos
    char a; // Declaração de uma variável
    a = 'c'; // Atribuição de um valor direto a variável
    int b = 20; // Declaração com atribuição diretamente
    int c = 30, d = 40 + 20; // Declaração múltipla com atribuição direta e expressão
    printf("a = %c, b = %d, c = %d, d = %d\n", a, b, c, d); // output: a = c, b = 20, c = 30, d = 60
    return 0; // O \n é utilizado para redirecionar o terminal para a linha de baixo (quebra linha)
}
```

Constantes:

Elementos cujo valor após sua declaração não pode ser modificado, sendo esse o único momento onde é possível atribuir um valor à ela.

Elementos locais e globais:

Se um elemento (variável ou constante) **é declarado dentro de um bloco, trata-se de um elemento local** (a memória dessa variável só fica reservada apenas durante a execução do bloco, e essa é acessível diretamente apenas pelo bloco onde foi declarada e seus sub-blocos), **se não, trata-se de um elemento global** (seu endereço de memória fica alocado durante toda a execução do código e esse pode ser acessado por qualquer função).

Sombreamento de elementos:

Elementos pertencentes aos mesmos blocos e sub-blocos não podem ter o mesmo nome, mas podem ter elementos com nomes iguais em blocos diferentes. Nesse caso, **se um dado global e um local tiver mesmo nome e houver uma chamada dele, o dado local que será processado**.

```c
#include <stdio.h>

const int constante_global = 10;
int variavel_global = 20;

int main(){
    const int constante_local = 30;
    int variavel_local = 40;
    int variavel_global = 50; // Sombreamento da variável global
    printf("constante_global = %d\nvariavel_global = %d\n", constante_global, variavel_global);
    /* output: constante_global = 10
               variavel_global = 50 */
    for(int i = 0; i < 10; i++){
        // Operação qualquer
    }
    int i = 5; // Como a variável i só existia no for, seu nome pode ser reaproveitado agora
    return 0;
}
```

4 Operações

As informações contidas nas variáveis podem ser manipuladas através de operações predefinidas na linguagem, essas seguindo uma ordem de prioridade que, a grosso modo, seguem a hierarquia: [08]

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

Elementos que tomam uma decisão a depender da validade de sua condição. Em C padrão não existe um tipo dados booleano (representa se um dado é verdadeiro ou falso) [09]. Todavia, **a linguagem entende o elemento 0 como falso e qualquer outro valor de qualquer outro tipo verdadeiro**.

if/else:

Estrutura de escolha binária (apenas duas opções, verdadeiro ou falso). Uma condição é definida no *if* que, se verdadeira, realiza a ação do bloco, mas se for falsa, passa-se para o próximo teste (*else if* ou *else*).

```c
#include <stdio.h>

int main(){
    int valor;
    scanf(" %d", &valor); // Recebe um valor no input e o envia para o endereço da variável
    if(valor == 10){
        printf("Eh 10!\n");
        return 10;
    } else if(valor == 20 || valor == 30) return 30; // valor != 10
    else { // valor != 10 && valor != 20 && valor != 30
        if('b') // Como o valor no escopo ('b') é diferente de 0, será executado sempre
            printf("Acao alcançada!\n");
        else
            return 1; // Condição nunca alcançada
    }
    return 0;
}
```

Switch case:

Estrutura que executa diferentes ações a depender do valor no escopo. Todos os *cases* necessitam obrigatoriamente um *break* (menos o *default*), se não, todos os cases abaixo também são executados (*fall-through*).

```c
#include <stdio.h>

int main(){
    int valor;
    scanf(" %d", &valor);
    switch (valor){
        case 10: // valor == 10
        // Ação 1
        break;
        case 30: case 40: // valor == 30 || valor == 40
        // Ação 3
        break;
        default: // valor != 10 && valor != 30 && valor != 40
        // Ação padrão
        break; // break facultativo
    }
    return 0;
}
```

goto:

Redireciona a execução do código para uma outra linha, essa definida por um *label*.

```c
#include <stdio.h>

int main(){
    int a = 10;
    goto LABEL;
    a *= 2; // a ficaria igual a 20, mas a instrução é pulada pelo goto
    LABEL:
    printf("%d\n", a); // output: 10
    return 0;
}
```

Seu uso é desencorajado, visto que deixa o programa desorganizado (dificulta o *debugging* e a manutenção) e pode 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 as ações de seu bloco até que uma condição de parada (ou um *break*) **seja alcançada**.

while:

Realiza uma ação enquanto a condição do seu escopo for "verdadeira" (diferente de 0), nem entrando no bloco se ela já for verdadeira. Existe uma variação dessa instrução, ***do while*, que executa o bloco antes de entrar no *loop*, repetindo-a enquanto a condição for verdadeira** (pelo menos uma vez irá executar a ação).

```c
#include <stdio.h>

int main(){
    int valor;
    scanf(" %d", &valor);
    while(valor != 10){ // Se valor == 10 nem entra
        printf("Valor diferente de 10\n");
        scanf(" %d", &valor);
    }
    do{
        printf("%d\n", ++valor); // output: 11
    } while (valor <= 10); // Realiza a ação pelo menos uma vez, mesmo se valor >= 20 antes do bloco
    return 0;
}
```

for:

Duração do *loop* delimitada no seu escopo. Nesse é definida uma variável e seu valor de início [10], sua condição de parada e uma operação, tudo separado por pontos e virgulas (;).

```c
#include <stdio.h>

int main(){
    int u = 0;
    for (int i = 0; u < 20; i++){ // Incrementa interna do for (i) de 1 em 1
        u += 2;
        printf("%d ", u);
    } printf("\n"); // output: 2 4 8 10 12 14 16 18 20
    for(u; u >= 10; u -= 3) // Decrementa variável externa ao for (u) de 3 em 3
        printf("%d ", u);
    printf("\n"); // output: 20 17 14 11
    return 0;
}
```

Ferramentas para alterar o fluxo de *loops*:

O *break* sai do bloco de execução instantaneamente (válido para *loops* e *switch*). **O *continue* ignora o resto das instruções abaixo dele no *loop***, começando a próxima repetição (funciona apenas em *loops*).

7 Funções

Funções são conjuntos de instruções montadas pelo programador para realizarem uma ação ao serem chamadas no código. Elas tem de ser escritas acima das funções que as utilizam (o *main*, portando, sendo a última), ter seu tipo de retorno definido (*void* caso não retorne nada) e podem admitir argumentos (variáveis locais que são declaradas no escopo que copiam os dados passados na chamada da função [11]).

Protótipos:

Ferramenta que pré-compilam as funções, permitindo seu posicionamento livre no código. É semelhante ao processo de declarar uma variável e definir seu valor depois, sendo necessário declarar em seu escopo apenas o tipo das variáveis. Os protótipos ainda devem ser declarados acima de funções que chamam sua função diretamente.

Bibliotecas:

As bibliotecas são códigos que possuem conjuntos de ferramentas e funções prontas. São disponibilizadas pela própria linguagem e podem ser adicionadas por meio do comando *#include <nome da biblioteca.h>* (verificar apêndice B). **Seu uso evita redundância de programação** (não há necessidade de criar funções do zero, economizando tempo) **e *bugs*** (as funções já foram testadas e otimizadas, sendo mais eficientes e previsíveis) [12].

```c
#include <stdio.h>  // Biblioteca p/ input e output

int dobro(int); // Prototipo que tem que ser posicionado acima, pois é chamado em funcaoSemPrototipo
void funcaoSemPrototipo(int valor){
    printf("%d\n", dobro(valor));
}
int funcaoComPrototipo(); // Protótipo que pode ser posicionado abaixo
int main(){
    int a; double b;
    // Funções da biblioteca stdio.h
    scanf("%d %lf", &a, &b); // Assumindo um input: 10 25
    printf("2*a = %d, b/2 = %.2lf\n", a*=2, b/=2); // output: 2*a = 20, b/2 = 12.50
    // Funções próprias
    funcaoSemPrototipo(funcaoComPrototipo()); // Mesmo que: funcaoSemPrototipo(25);
    return 0; // output: 50
}
int dobro(int valor){
    return (valor * 2);
}
int funcaoComPrototipo(){ // void implícito no argumento
    return 25;
}
```

Recursão:

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

```c
#include <stdio.h>

void fibonacci_recursivo(int antecessor, int atual, int termo){
    printf("%d, ", antecessor);
    if(termo > 0) // Chamada recursiva dos n primeiros termos da sequência
    fibonacci_recursivo(atual, antecessor + atual, --termo);
    else printf("%d\n", atual); // Fim da chamada
}
```

```c
int main(){
    fibonacci_recursivo(1,1,15);
    return 0;
}
```

8 Ponteiros

Ponteiros são elementos especiais que armazenam em si o endereço de outras variáveis, podendo alterá-las, mesmo sem uma chamada direta da mesma. Para criar um ponteiro, basta colocar um asterisco ($*$) ao lado do tipo de dado durante a declaração. **Podem ser criados ponteiros para quaisquer tipos, até ponteiros de um ponteiro**, através de um duplo asterisco ($**$) e assim por diante. O endereço de memória das variáveis pode ser acessado pelo prefixo &.

```c
#include <stdio.h>

int main(){
    int a = 10;
    int* b; // Declaração de um ponteiro de int
    b = &a; // b recebe o endereço de a (*b = a)
    (*b)++; // Mesmo que a++;
    printf("%d\n", a); // output: 11
    int* c = &a; // Declaração e atribuição de um ponteiro (c = &a, logo: *c = a)
    // Para declarar múltiplos ponteiros, cada um deve ter a quantidade respectiva de asteriscos
    int** d, * e; // d é um ponteiro para ponteiro de int, e é um ponteiro de int
    d = &b; // d = &b: *d = b = &a: **d = a
    return 0;
}
```

Passagem por valor x Passagem por referência:

Como já foi dito, no escopo de uma função são geradas variáveis locais (válidas apenas naquele bloco e em seus sub-blocos) que copiam o valor passado na sua chamada.

Se for passado um valor, ele será copiado, não interagindo nem alterando diretamente a variável original, sendo esse processo chamado de passagem por valor. Todavia, **se for passado um endereço de uma variável, ele é copiado para o ponteiro local da função, passando a interagir diretamente com o valor original**, esse processo sendo a passagem por referência.

```c
#include <stdio.h>

void divisao(int valor, int* referencia){ // Recebe um elemento int e um endereço de int
    valor /= 2; // Copia local de a
    *referencia /= 2; // Ponteiro com o endereço de b
}
int main(){
    int a = 10, b = 10;
    divisao(a, &b); // Envia o valor de a e o endereço de b
    printf("a = %d, b = %d\n", a, b); // output: a = 10, b = 5
    return 0;
}
```

9 *Arrays*

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

strings:

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

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.

```c
#include <stdio.h>

void funcao_com_array(int arr[]) { // Array no escopo
    for(int i = 0; i < 3; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
void funcao_com_ponteiro(int *ptr) { // Ponteiro no escopo
    for(int i = 0; i < 3; i++)
        printf("%d ", ptr[i]); // Poderia ser *(ptr + i)
    printf("\n");
}
int main(){
    int a;
    scanf("%d", &a);
    int array_vazio[a]; // Declaração de array com tamanho não definido e com lixo de memória
    int array1[5] = {0}, array2[5] = {1, 2, 3}; // array1 = {0, 0, 0, 0, 0}, array2 = {1, 2, 3, 0, 0}
    int array3[] = {4, 5, 6}; // Tamanho inferido (3 elementos)
    int* ptr = array2; // Ponteiro para o primeiro elemento do array
    // (ptr[n] == array2[n] == *(ptr + n) == *(array2 + n))
    funcao_com_array(array2); // output: 1 2 3
    funcao_com_array(ptr); // output: 1 2 3
    funcao_com_ponteiro(ptr); // output: 1 2 3
    funcao_com_ponteiro(array3); // output: 4 5 6
    return 0; // Todas as chamadas são válidas
}
```

Em geral, **múltiplos elementos de um *array* só podem ser inicializados simultaneamente durante sua declaração**, depois a modificação sendo individual (processo normalmente feito através de *loops*).

Matrizes:

Sendo um vetor uma matriz de dimensão 1, **matrizes de n dimensões são conjuntos de matrizes de dimensão $n-1$**. Em sua declaração é obrigatório definir o tamanho de todas as dimensões de uma vez. **Outra forma de representar matrizes é através de ponteiros múltiplos, mas esses não se misturam**.

```c
#include <stdio.h>

void funcao_com_matriz(int matriz[][5]){ // Necessidade de indicar o tamanho de cada array do conjunto
    printf("{");
        for(int i = 0; i < 10; i++) printf("%d%s", *(*matriz + i), i == 9 ? "": ", ");
    printf("}\n");
} // Aceita apenas matrizes (memória sequencial)
void funcao_ponteiro(int** ptr){ // Ponteiro no escopo (poderia ser int* array[])
    for(int i = 0; i < 3; i++){
        printf("{");
            for(int j = 0; j < 3; j++)
            printf("%d%s", ptr[i][j], j == 2 ? "": ", ");
            printf("}%c ", i == 2 ? '\0' : ',');
    } printf("\n");
} // Aceita apenas ponteiros (memória não necessariamente sequencial)
int main(){
    int matriz_1[3][10]; // Lixo de memória
    int matriz_2[3][3] = {0}; // {{0, 0, 0}, {0, 0, 0}, {0, 0, 0}} ou {0, 0, 0, 0, 0, 0, 0, 0, 0}
    int matriz_3[3][3] = {1, 2, 3}; // {{1, 2, 3}, {0, 0, 0}, {0, 0, 0}}
    int matriz_4[3][3] = {{9, 8, 7}, {6, 5, 4}}; // {{9, 8, 7}, {6, 5, 4}, {0, 0, 0}}
    int matriz_5[2][5] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; // {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}}
    int* ptr_array[3] = {matriz_4[0], matriz_4[1], matriz_4[2]}; // {{9, 8, 7}, {6, 5, 4}, {0, 0, 0}}
    int** ptr_duplo = ptr_array; // Diferente de {9, 8, 7, 6, 5, 4, 0, 0, 0}
    funcao_com_matriz(matriz_5); // output:{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
    funcao_ponteiro(ptr_duplo); // output: {9, 8, 7}, {6, 5, 4}, {0, 0, 0}
    funcao_ponteiro(ptr_array); // output: {9, 8, 7}, {6, 5, 4}, {0, 0, 0}
    return 0;
```

```c
    /* Erros de declaração:
    int* ptr_array[3] = matriz_3;
    int* ptr_array_1[3] = {{1, 2, 3}, {0, 0, 0}, {0, 0, 0}};
    int** ptr_duplo = matriz_3;
    int** ptr_duplo = {matriz_5[1], matriz_5[2], matriz_3[3]};
    int** ptr_duplo = {{1, 2, 3}, {0, 0, 0}, {0, 0, 0}};
    Cada array deve ser passada individualmente, ou então passadas múltiplas arrays já declaradas */
}
```

10 Alocação dinâmica

Existem diferentes tipos de alocação ((verificar apêndice C)). **A alocação estática é a gerenciada pelo sistema**, ocorrendo em situações como declaração de variáveis, chamadas de função, etc. **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) o espaço depois da sua utilização [15]. As funções para realizar a alocação dinâmica estão definidas na biblioteca *stdlib.h* [16].

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

int main(){
    int a = 10, b;
    int* ptr = (int *)malloc(sizeof(int)); // Aloca memória para int (com lixo de memória)
    *ptr = 10;
    free(ptr); // Desaloca a memória reservada antes de reutilizar o ponteiro
    ptr = &a;
    scanf("%d", &b);
    int* array1 = (int*)malloc(b * sizeof(int)); // Array de tamanho não definido com lixo de memória
    int* array2 = (int*)calloc(10, sizeof(int)); // Array de tamanho fixo com todos os elementos zerados
    free(array1);
    free(array2);
    return 0;
}
```

11 *Structs*

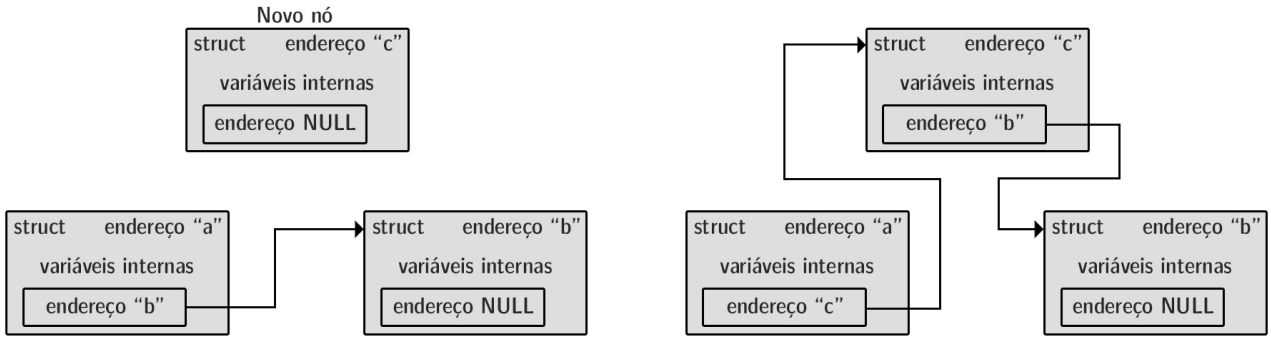
Uma *struct* é um tipo derivado de dados que pode ser definido como um grupo de variáveis de múltiplos tipos relacionados entre si [17]. Simula algo como um "objeto" de linguagens que utilizam *POO*, mas mais primitivos (não suporta funções internas, abstrações, encapsulamento e/ou polimorfismo).

```c
#include <stdio.h>

// Definição da "struct carta"
struct carta{
    char *naipe, *face;
    int valor;
};
// Pode-se definir um nome menor para declarar ao invés de "struct carta"
typedef struct carta carta;
typedef struct carta CartaDeBaralho;
int main(){
    struct carta a; // Declaração de uma variável do tipo "struct carta"
    a.naipe = "Paus"; a.valor = 9; a.face = "Nove";
    // Declarações usando os nomes do type def
    carta b = {"Ouros", "Rei", 10}; // Declaração e inicialização ordenada dos elementos
    CartaDeBaralho c = {.face = "As", .valor = 11, .naipe = "Copas"}; // Declaração nomeada (ordem livre)
    return 0;
}
```

12 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, ele é "encaixado" entre elementos, o antecessor apontando para ele e ele passando a apontar 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).

```c
#include <stdio.h>
#include <stdlib.h>
typedef struct lista { // Definição da "struct" diretamente com o typedef
    int elemento;
    struct lista* proximo; // Ponteiro para o próximo nó
} Lista;
typedef Lista* ListaPtr; // Facilitador de sintaxe para ponteiros (vai usar bastante)
int main(){
    ListaPtr minha_lista = NULL; // Uso de um ponteiro pois será usada alocação dinâmica
    int valor;
    scanf("%d", &valor);
    while (valor != 0){
        ListaPtr novo_no = (ListaPtr)malloc(sizeof(Lista));
        if (novo_no == NULL) return 1; // Falha na alocação
        // Exemplo de pilha (elemento adicionado no início da lista)
        novo_no->elemento = valor; // Equivalente à (*novo_no).elemento = valor;
        novo_no->proximo = minha_lista;
        minha_lista = novo_no;
        scanf("%d", &valor);
    } // loop para desalocar a memória
    while(minha_lista != NULL){ // Cada nó tem de ser liberado individualmente
        ListaPtr temp = minha_lista; // Guarda nó atual para liberar depois
        minha_lista = minha_lista->proximo; // Avança para o próximo nó
        free(temp); // Liberar a memória do nó atual
    } return 0;
}
```

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

```c
int add_crescente(ListaPtr* lista, int valor) {
    ListaPtr novo_no = calloc(1, sizeof(Lista)), atual = *lista, prev = NULL;
    if (!novo_no) return -1;
    novo_no->elemento = valor;
    // Procura a posição até o final da lista ou até o valor ser maior do que o novo
    while (atual != NULL && atual->elemento < valor) {
        prev = atual;
```

```
        atual = atual->proximo;
    } novo_no->proximo = atual; // atual->elemento >= valor ou atual == null (último elemento)
    if (prev == NULL) *lista = novo_no; // Inserção no primeiro elemento
    else prev->proximo = novo_no; // Inserção depois do prev
    return 0; // Antes: [..., prev, atual, atual->proximo, ...]
}             // Depois : [..., prev, novo_no, atual, atual->proximo, ...]
```

Vale explicitar que o que é alterado é o endereço que o elemento aponta. Por isso quando a alteração é feita por um ponteiro local, a lista ainda sim é alterada.

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).

13 Arquivos externos

A linguagem C possui ferramentas que permite a leitura e escrita persistente (permanece salvo após a execução do programa) **em arquivos externos**. Cria-se um ponteiro para arquivos ($FILE*$), para a navegação e manipulação do arquivo, a depender do tipo de acesso [18]. Posteriormente, desaloca-se o ponteiro (encerra-se o uso do arquivo garantindo o salvamento). **Os arquivos podem ser gravados e lidos em dois tipos diferentes**:

- **Texto:** Todo conteúdo é uma *string*, necessitando de uma conversão para o formato de dado desejado.

```
#include <stdio.h>

int main() {
    int conta[100] = {0}; double saldo[100] = {0};
    FILE* filePointer;
    // Tenta abrir o arquivo para a leitura de texto (.txt, .dat, etc.)
    if((filePointer = fopen("dados.dat", "r")) == NULL) return -1; // Erro ao abrir o arquivo
    for(int i = 0; !feof(filePointer); i++) // Lê até o final do arquivo (EOF)
        // Os dados estão dispostos 'int''espaço''double''espaço''int''espaço''double'...
        // Leitor de texto (lê, armazena os dados e avança o ponteiro para a próxima sequencia)
        fscanf(filePointer, " %d %lf", &conta[i], &saldo[i]); // Espaço antes do %d para a limpeza do buffer
    fclose(filePointer); // Desaloca o ponteiro e fecha o arquivo (salvamento)
    // Tenta criar ou sobrescrever arquivo para a escrita de texto
    if((filePointer = fopen("novo.txt", "w")) == NULL) return -1;
    for(int j = 0; conta[j] != 0; j++)
        fprintf(filePointer, "%d - %.1lf\n", conta[j], saldo[j]); // Escritor de texto
    // A string fica: "int - double"'\n'"int - double"\n'"int - double"'\n'...
    fclose(filePointer);
    // Tenta abrir ou criar arquivo para a leitura e append de texto
    if((filePointer = fopen("novo.txt", "a+")) == NULL) return -1;
    fscanf(filePointer, "%d - %lf", &conta[20], &saldo[20]); // Lê o conteúdo (cuidado com a formatação)
    fprintf(filePointer, "%d - %.31f\n", conta[20], saldo[20]); // Anexa os dados no final
    fclose(filePointer);
    return 0;
}
```

- **Binário:** A informação é armazenada numericamente, operando em blocos de memória (os valores permanecendo os mesmos, não sendo convertidos em *strings*).

```c
#include <stdio.h>

typedef struct {
    int conta;
    double saldo;
} Dados;
int main() {
    Dados dados[4] = {{1001, 1000.50}, {1002, -505.10}, {1010, 27}, {0, 0}}, lidos[4] = {{0, 0}};
    FILE* filePointer;
    // Tenta criar ou sobrescrever arquivo para a escrita de binário
    if((filePointer = fopen("dados.bin", "wb")) == NULL) return -1;
    for(int i = 0; dados[i].conta != 0; i++)
        // fwrite(endereço, tamanho de cada objeto, quantos dados escrever de uma vez, ponteiro)
        fwrite(&dados[i], sizeof(Dados), 1, filePointer);
    // fwrite(dados, sizeof(Dados), 3, filePointer); sem o loop pegaria os dados[0 - 2] de uma vez
    fclose(filePointer);
    // Tenta abrir arquivo para a leitura de binário
    if((filePointer = fopen("dados.bin", "rb")) == NULL) return -1;
    for(int i = 0; !feof(filePointer); i++)
        fread(&lidos[i], sizeof(Dados), 1, filePointer);
    fclose(filePointer);
    for(int i = 0; lidos[i].conta != 0; i++) printf("%d - %.2lf\n", lidos[i].conta, lidos[i].saldo);
    // output: "1001 - 1000.50"'\n'"1002 - -505.10"'\n'"1010 - 27.00"'\n'
    return 0;
}
```

14 Pré-processamento

É possível realizar ações e definir elementos antes de o programa ser compilado, por meio de comando # *"diretiva"*. As aplicações vão desde a inclusão de arquivos externos até a definição de constantes simbólicas (globais), compilação condicional, etc. [19]

```c
// #include:
/* Inclui arquivos para a execução do programa */
#include <stdio.h>    // Biblioteca padrão
#include "mylib.h"    // Biblioteca autoral
// #define e #undef:
/* O primeiro define e copia macros (constantes e funções) e o segundo os apaga (diretivas locais e externas)*/
#define PI 3.14
#define NOME_PROGRAMA "Calculadora Financeira"
#define PROGRAMA NOME_PROGRAMA // Dois macros com o mesmo valor
#define AREA_CIRCULO(raio) (PI * (raio) * (raio))
// Seria o mesmo que fazer: double area_circulo(double raio){ return PI * (raio) * (raio);}
#undef DEZ // Definido em mylib.h (biblioteca externa)
// #if, #elif e #else:
/* Verifica a condição de uma diretiva */
#ifndef DEZ // if (!defined(NOME)), também tem if defined(NOME)
    #warning "DEZ não definido!"
    #define DEZ 15  // Lança uma mensagem no terminal
#endif
#if (DEZ > 20)
    #undef DEZ
    #define DEZ 25
#elif (DEZ < 15) // else if de diretivas
    #undef DEZ
    #define DEZ 5
#else
    #undef DEZ
    #define DEZ 10
#endif
```

Compilação de múltiplos arquivos:

Para compilar múltiplos códigos interligados em C, é necessário a criação de um arquivo fonte (arquivo *.c* contém a implementação das funções) **, um arquivo cabeçalho/biblioteca** (arquivo *.h* que contem

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).

```c
/* mylib.h */
#ifndef MYLIB_H
    #define MYLIB_H
#define DEZ 10 // Constantes públicas
typedef struct { // Estruturas públicas
    int x;
    int y;
} Ponto;
#endif
double media(double* , int); // Protótipo de funções

/* mylib.c */
#include "mylib.h"

double media(double* valores, int quantidade){  // Declaração da função
    double total = 0;
    for(int i = 0; i < quantidade; i++) total += valores[i];
    return total / quantidade;
}

/* main */
#include <stdio.h>
#include "mylib.h>

int main(){
    double* valores = {DEZ, 20, 25, 30}; // Uso de variável global da biblioteca externa
    printf("Média = %.2lf\n", media(valores, 4)); // Uso de função da biblioteca externa
    Ponto ponto = {10, 20}; // Declaração de struct da biblioteca externa
    return 0;
}
```

15 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**.

```c
#include <stdio.h>

int main(int argc, char* argv[]){
    printf("A última mensagem enviada pela execução do programa pelo terminal é: %s\n", argc ==1 ? "Não foi
    ↪  enviado nada" : argv[argc - 1]);
    return 0;
}
```

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] - **Estruturas de controle podem ser usadas sem chaves, executando apenas a instrução imediatamente subsequente**. É recomendado sempre o uso de chaves, a fim de evitar comportamentos inesperados.

Tipos de dados

[03] - **Cada tipo possui uma maneira de representar ausência de valor** (elemento nulo), por exemplo, para elementos numéricos, o elemento nulo é o 0, para *char* é o $'\backslash 0'$, para ponteiros, *arrays* e *strings* é *NULL*, etc.

[04] - **Por vezes o espaço de memória padrão para tipos numéricos 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, **é possível aumentar o tamanho de memória para esses dados**, através da adição do prefixo *long* ao na declaração variáveis. O formatador da variável passa a ter um *l* após à porcentagem (%) (*long int* $\rightarrow$ *%ld*, por exemplo).

[05] - Os métodos de conversão de tipos de dados são:

- **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).
- **Conversão implícita:** Automática pelo compilador. Normalmente usado para converter um tipo menor para um tipo maior (*float* para *double*, por exemplo).

É melhor sempre realizar a conversão explicitamente (torna o código mais claro e evita *bugs*).

```c
#include <stdio.h>
int main(){
    // Casos de conversão:
    double pi = 3.1415926535;
    int i_pi = (int) pi; // i_pi = 3
    int* intPtr;
    char* charPtr = (char*) intPtr;
    unsigned int u = 5;
    int s = -5;
    if(s < (int) u){ /*Realiza uma ação */}
    /* Sem a conversão, ambos seriam tratados como unsigned. Assim, em binário:
    -5 = (1) 111111 11111111 11111111 11111011 = 4.294.967.291 (unsigned) > 10 */
    // Casos onde ocorre a conversão implícita:
    char a = 'A';        // char (65 na tabela ASCII)
    int b = a;           // char -> int implícito ((int) a == 65)
    float c = b;         // int -> float implícito ((float) b == 65.0)
    double d = c;        // float -> double implícito ((double) c == 65.0)
    int e = d;           // double -> int implícito ((int) d == 65)
    double op1 = b/10;   // op1 = (double)(65/10) = (double)6 = 6.0
    double op2 = b/10.0; // op2 = ((double)65)/10.0 = 65.0/10.0 = 6.5
    return 0;
}
```

[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).

Armazenamento de informações

[07] - **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.

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* (ˆ), *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 (+ =, − =, ∗ =, / =, % =, & =, | =, ˆ =, <<=, >>=). 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] - Atualmente existe uma biblioteca que permite a utilização de elementos do tipo booleano (*stdbool.h*), variáveis desse tipo podendo assumir os valores "*true*" ou "*false*".

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] - Em C padrão, **a quantidade de variáveis presente no escopo é predefinida e fixa**, todavia, existe uma biblioteca que permite a utilização de uma quantidade variável de parâmetros na função (*stdarg.h*).

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

double media(double total, int i, ...){ // É preciso PELO MENOS UM argumento fixo
    va_list pointer; // Declara um "ponteiro" para a lista de argumentos variáveis
    va_start(pointer, i); // Coloca o ponteiro no último argumento fixo (nesse caso o i)
    for (int j = 0; j < i; j++) // Passa por cada argumento a partir de i até o final da lista
    total += va_arg(pointer, double); // Acessa o próximo argumento da lista (tamanho de um double)
    va_end(pointer); // Zera o ponteiro (evita erros de acesso à memória)
    return(total / i);
}
int main(){
    double w = 38.5, x = 22.5, y = 1.7, z = 10.2;
    printf("%.3lf\n", media(0, 2, w, x)); // output: 30.500
    printf("%.3lf\n", media(0, 3, w, x, y)); // output: 20.900
    printf("%.3lf\n", media(0, 4, w, x, y, z)); // output: 18.225
    return 0;
}
```

[12] - **As funções mais importantes da biblioteca *stdio.h* são o *scanf*** (recebe uma *string* no *input* (terminal), converte os dados da *string*, organizados da maneira definida no escopo do *scanf*, nos tipos definidos pelo formatador e grava os valores no endereço de variáveis) **e o *printf*** (recebe informações de vários tipos, converte elas em *string*, através do formatador, e imprime tudo no *output* padrão (terminal)).

O *scanf* pode acabar funcionando de maneira incorreta por conta do *buffer* (área da memória usada para guardar dados de *input* provisoriamente, enquanto eles estão sendo processados) **estar cheio antes de sua execução**. Para contornar esse problema, basta colocar dentro as aspas um espaço antes do primeiro formatador. 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 feito com cuidado**, visto que o processo deixa variáveis alocadas e ativas dentro de cada recursão. Para uma função com n variáveis, cara 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. 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 para o tipo de ponteiro desejado é feita automaticamente**. Dessa maneira a explicitação do tipo de ponteiro é facultativo em C (em C++ é obrigatório).

Structs

[17] - **Não é possível predefinir 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* predefinidos 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 ou extern *auto*, *static* ou *extern*.

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

```c
#include <stdio.h>

int contExtern = 0; // extern int contExtern = 0;
void inicializacao() {
    static int contStatic = 0;
    int contAuto = 0; // auto int contAuto = 0;
    contExtern++, contStatic++, contAuto++;
    printf("%d, %d, %d\n", contExtern, contStatic, contAuto);
}
int main() {
    inicializacao(); // output: 1, 1, 1
    inicializacao(); // output: 2, 2, 1
    inicializacao(); // output: 3, 3, 1
    return 0;
}
```

Programa pelo terminal

[21] - Para o seguinte código, tendo um *.txt* que será redirecionado como entrada com o seguinte conteúdo:

```c
#include <stdio.h>

int main() {
    int a = 1, total = 0;
    while (a != 0){
        scanf(" %d", &a);
        total += a;
    }
    printf("O total da soma é: %d\n", total);
    printf("Mais um valor para a soma: ");
    scanf(" %d", &a);
    printf("O total da soma é: %d\n", total + a);
    return 0;
}
```

- "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*, gerando um erro (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).

Apêndice B

stdio.h

Biblioteca que permite utilização e manipulação do *input* e *ouput* de um programa, permitindo o código receber e enviar dados de/para diversas fontes.

```c
#include <stdio.h>

typedef struct {
    int inteiro;
    float racional;
} grupo;
int main(){
    /* Ferramentas para input e output padrão (terminal) */
    int int_a; char char_b, vet_d[100], vet_e[100]; double double_f;
    scanf(" %d %c", &int_a, &char_b); // Recebe valores input e armazena nos endereços (input assumido: 10 abc)
    printf("int_a = %d, pi = %.2lf, Nome: %s\n", int_a, 3.141592, "Maria"); // Imprime uma string no terminal
    // output: int_a = 10, pi = 3.14, Nome = Maria
    sprintf(vet_e, "char_b = %c, num = %.4lf, Nome: %s\n", char_b, 2.15, "Joao"); // printf para array de char
    printf("vet_e: %s", vet_e); // output: vet_e: char_b = a, num = 2.1500, Nome: Joao
    /* Ao enviar múltiplos inputs, eles ficam armazenados em fila no buffer. Para o input assumido, ainda estão
    no buffer os caracteres 'b', 'c' e '\n' */
    char_b = getchar(); // Pega o proximo caracter da fila do buffer ('b').
    putchar(char_b); // Imprime um char
    char linha[100];
    fgets(linha, 100, stdin); // Limpa o buffer (100 caracteres pra ter certeza).
    // fgets e getchar são muito usados para limpeza de buffer (principalmente ao se mexer com strings).
    puts(linha); // Imprime uma string colocando um '\n' ao final (output: bc'\n''\n')
    // Recebe uma string de tamanho <= 100 de um local (stdin = terminal) e armazena em um array de char
    fgets(vet_d, 50, stdin); // O '\n' também é pego no input (supondo input: Maranhao)
    sscanf(vet_e, "char_b = %c, num = %lf", &char_b, &double_f); // scanf de string
    printf("vet_d = %schar_b = %c e double_f = %.2lf\n", vet_d, char_b, double_f);
    // output: vet_d = Maranhao'\n' char_b = a e double_f = 2.15

    /* Ferramentas para arquivos como input e output */
    // Arquivos de texto
    int vet_a[] = {10,20,30,40,50}, vet_b[10] = {0};
    FILE* filePtr; // Cria um ponteiro para arquivos
    if((filePtr = fopen("texto.txt", "w+")) == NULL) return -1; // Tenta abrir um arquivo de texto para leitura
    ↪  e escrita
    for(int i = 0; i < (sizeof(vet_a)/sizeof(vet_a[0])); i++)
        fprintf(filePtr, "%d - %d\n", i, vet_a[i]); // Salva dados no .txt no formato especificado
    rewind(filePtr); // Volta o ponteiro pro início do arquivo
    for(int i = 0, tmp = 0; !feof(filePtr); i++)
        fscanf(filePtr, "%d - %d", &tmp, &vet_b[i]); // Salva os dados num array
    fclose(filePtr); // Fecha e salva o arquivo
    for(int i = 0; vet_b[i] != 0; i++)
        printf("%d %s", vet_b[i], vet_b[i + 1] == 0 ? "" : "- ");
    putchar('\n'); // output: 10 - 20 - 30 - 40 - 50

    // Arquivos binário
    grupo vet_struc_c[] = {{1,6},{2,5},{3,4},{4,3},{5,2},{6,1}}, vet_struc_d[30] = {{0,0}};
    if((filePtr = fopen("binario.bin", "ab")) == NULL) return -1; // Tenta abrir ou criar um arquivo binário
    ↪  para anexação de dados
    for(int i = 0; i < (sizeof(vet_struc_c)/(sizeof(vet_struc_c[0]))); i += 3)
        fwrite(&vet_struc_c[i], sizeof(grupo), 3, filePtr); // Passa 3 elementos pro arquivo binário de cada
        ↪  vez
    // {c[0], c[1], c[2]}, {c[3], c[4], c[5]}
    fclose(filePtr); // Modos a e ab não permitem rewind
    if((filePtr = fopen("binario.bin", "rb")) == NULL) return -1;
    for(int i = 0; !feof(filePtr); i++)
        fread(&vet_struc_d[i], sizeof(grupo), 1, filePtr); // Lê um elemento do arquivo binário de cada vez
    fclose(filePtr);
    for(int i = 0; vet_struc_d[i].inteiro != 0; i++)
        printf("%d %.2lf %s", vet_struc_d[i].inteiro, vet_struc_d[i].racional, vet_struc_d[i + 1].inteiro == 0
        ↪  ? "" : "- ");
    putchar('\n'); // output: 1 6.00 - 2 5.00 - 3 4.00 - 4 3.00 - 5 2.00 - 6 1.00
    return 0; // Cada execução, o último output se repete (por conta do append)
}
```

stdlib.h

Biblioteca com ferramentas mais gerais (alocação manual de memória, conversão de tipos, ordenação e busca, etc.).

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

/* É preciso uma função para definir o método de organização do qsort e para o bsort. Para uma organização
crescente, por exemplo, ele retorna (*(int*) a - *(int*) b), e troca as casa quando o resultado for > 0 */
int crescente(const void* a, const void* b) {
    return (*(int*) a - *(int*) b);
} // Caso quisesse a organização decrescente: return (*(int*) a - *(int*) b)
int main(){
    /* Funções para alocação dinâmica */
    int* intptr = (int*)malloc(sizeof(int)); // Aloca memória (com lixo)
    int* intarr = (int*)calloc(10, sizeof(int)); // Aloca e ZERA a memória alocada
    intarr = (int*)realloc(intarr, 20 * sizeof(int)); // Redimensiona memória alocada
    free(intptr);
    free(intarr);

    /* Funções para conversão de tipos */
    char* cPtr1,* cPtr2;
    int a = atoi("99"); // Converte string com APENAS CARACTERES NUMÉRICOS para int.
    // Existem funções semelhantes para outros tipos (atof: string → float, atol: string → long, etc.)
    double b = strtod("51.2% foram admitidos", &cPtr1); //atol com detecção de erro (b = 51.2)
    /* Caso haja elementos não numéricos após o número (obrigatório o número na frente), o resto é convertido
    em string e armazenado em um ponteiro de char (array daria erro) ou eliminado se colocado NULL */
    // O número determina a base (decimal, hexadecimal, binária, etc.). Base = 0 converte automaticamente
    long bin = strtol("100101abc", &cPtr2, 2); // bin = 37 em decimal, cPtr = abc
    long hex = strtol("0x123", NULL, 0); // hex = 291 em decimal
    printf("b = %.2lf, bin = %ld e %b, hex = %ld e %x\n", b, bin, (int) bin, hex, (int) hex);
    // "b = 51.20, bin = 37 e 100101, hex = 291 e 123"
    int c = abs(-25); // c = 25 | |-25|
    div_t res = div(70, 30); // Divisão inteira com quociente e resto (res.quot = 2, res.rem = 10)

    /* Funções para ordenação e busca */
    int numeros[] = {42, 13, 7, 99, 1, 25};
    int n = sizeof(numeros) / sizeof(numeros[0]);
    // Organiza os elementos de uma lista seguindo uma ordem definida pela função na última posição
    qsort(numeros, n, sizeof(int), crescente); // numeros = {1, 7, 13, 25, 42, 99}
    for(int i = 0; i < n; i++) printf("%d ", numeros[i]);
    putchar('\n'); // output: 1 7 13 25 42 99
    int chave = 25;
    // Faz uma busca binária em uma lista de organização variada e retorna o valor da chave se achá-la
    int* resultado = bsearch(&chave, numeros, n, sizeof(int), crescente);
    if (resultado != NULL) printf("Encontrado: %d\n", *resultado); // output: Encontrado: 25
    return 0;
}
```

math.h

Biblioteca com funções de operações e conversões matemáticas.

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

int main() {
    printf("98.0001: %.2lf, 10.8: %.2lf\n", ceil(98.0001), floor(10.8)); // output: 98.0001: 99.00, 10.8: 10.00
    printf("10.00: %.2lf, %.2lf\n", floor(10.00), ceil(10.00)); // output: 10.00: 10.00, 10.00
    printf("A raiz quadrada de 8 é: %.4lf\n", sqrt(8)); // output: A raiz quadrada de 8 é: 2.8284
    printf("A raiz cúbica de 27 é: %.4lf\n", pow(27, 1.0/3)); // output: A raiz cúbica de 27 é: 3.0000
    printf("(2.3)^(7.5) é: %.4lf\n", pow(2.3, 7.5)); // output: (2.3)^(7.5) é: 516.3673
    printf("%.3lf, %.3lf, %.3lf\n",sin(3.1416/2),cos(3.1416/3),tan(3.1416/4)); // output: 1.000, 0.500, 1.000
    printf("ln(e) é: %.4lf\n", log(2.71828)); // output: ln(e) é: 1.0000
    printf("log10(1000) é: %.4lf\n", log10(1000)); // output: log10(1000) é: 3.0000
    return 0;
}
```

ctype.h e *string.h*

Bibliotecas para a **manipulação de elementos do tipo *char* e *string***, respectivamente (muito usados em conjunto).

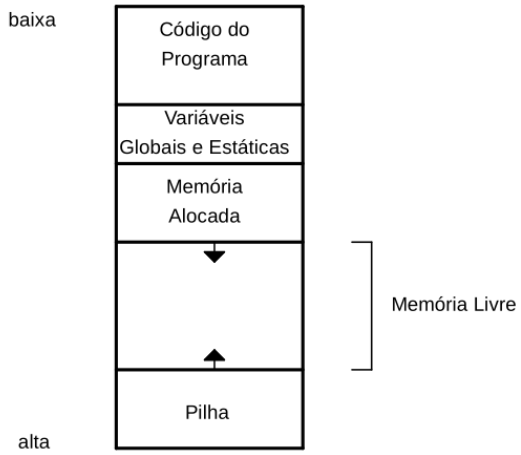
```c
#include <stdio.h>
#include <ctype.h>
#include <string.h>

int main() {
    /* Funções do ctype.h */
    // Funções de comparação. Retornam 1 se o char tiver as propriedades, caso contrário, retornam 0
    // Número decimal
    printf("'8': %s\n", isdigit('8') ? "" : "não "); // 1
    printf("'.': %s\n", isdigit('.') ? "" : "não "); // 0
    // Letra do alfabeto
    printf("'b': %s\n", isalpha('b') ? "" : "não "); // 1
    printf("'4': %s\n", isalpha('4') ? "" : "não "); // 0
    // Letra maiúscula
    printf("'c': %s\n", isupper('c') ? "" : "não "); // 0
    printf("'F': %s\n", isupper('F') ? "" : "não "); // 1
    // Letra minúscula
    printf("'k': %s\n", islower('c') ? "" : "não "); // 1
    // Letra do alfabeto ou número
    printf("'A': %s\n", isalnum('A') ? "" : "não "); // 1
    printf("'7': %s\n", isalnum('7') ? "" : "não "); // 1
    printf("'&': %s\n", isalnum('&') ? "" : "não "); // 0
    // Espaço
    printf("'\\n': %s\n", isspace('\n') ? "" : "não "); // 1
    printf("' ': %s\n", isspace(' ') ? "" : "não ");   // 1
    printf("'_': %s\n", isspace('_') ? "" : "não ");   // 0
    // Funções de modificação. Caso sejam não seja possível modificar, é retornado o mesmo caracter
    printf("'u' toupper é '%c'\n", toupper('u'));
    printf("'7' toupper é '%c'\n", toupper('7'));
    printf("'W' tolower é '%c'\n", tolower('W'));
    printf("'m' tolower é '%c'\n", tolower('m'));

    /* Funções do string.h */
    char* str1 = "Feliz aniversário", str2[20], str3[20], str4[20];
    // Retorna a quantidade de caracteres da string (contando o '\0')
    printf("%ld\n", strlen(str1)); // output: 18
    // Copia a string do segundo vetor para o primeiro
    strcpy(str2, str1); // str2 = "Feliz aniversário\0"
    printf("str2 = %s\n", str2); // output: str2 = Feliz aniversário
    // Copia os n primeiros caracteres (se n > tamanho da str, copia tudo)
    strncpy(str3, str2, 5); str3[5] = '\0'; // == "Feliz\0"
    strncpy(str4, str2, 20); // str4 = "Feliz aniversário\0"
    printf("str 3 = %s - str4 = %s\n", str3, str4);
    //  output: str 3 = Feliz - str4 = Feliz aniversário
    char* str5 = "Feliz ano novo", str6[20] = "";
    // Anexa a string do segundo vetor ao final do primeiro
    strcat(str6, str5); // str6 = "Feliz ano novo\0"
    str6[strlen(str6)] = ' '; // == "Feliz ano novo "
    // Anexa os n primeiros caracteres
    strncat(str6, str5, 5); // str6 = "Feliz ano novo Feliz"
    str6[strlen(str6) + 1] = '\0'; // == "Feliz ano novo Feliz\0"
    printf("str6 = %s\n", str6); // output: str6 = Feliz ano novo Feliz
    // Compara as strings e retorna 0 se são iguais
    printf("São %s\n", !strcmp("Abc", "Abc") ? "iguais" : "diferentes"); // 0
    printf("São %s\n", !strcmp("Abc", "Abd") ? "iguais" : "diferentes"); // 1
    // Compara os n primeiros caracteres de duas strings
    printf("São %s\n", !strncmp("Abc", "Abd", 2) ? "iguais" : "diferentes"); // 0
    // Retorna a a string a partir da primeira ocorrência do caracter (retorna NULL se não achar)
    char* ptr7 = strchr("Uma maquina voadora", 'q'); // == "quina voadora\0"
    char* ptr8 = strrchr("Uma maquina voadora", 'z'); // == NULL
    // A mesma ação do strchr, mas para achar uma string
    char* ptr9 = strstr("O bebê saiu dai", "iu"); // == "iu dai\0"
    printf("%s\n", ptr9);
    return 0;
}
```

Apêndice C

Estrutura da memória (*stack/heap*):

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

A arquitetura mais comuns organizam a memória começando no armazenamento do código do programa e das variáveis globais e estáticas (elementos fixos) na região mais baixa (endereço de memória menor), a região mais acima sendo a área dinâmica (*heap*), a memória livre e mais acima a pilha (*stack*) (os últimos 3 variando sua região alocada e as informações de seus endereços).



- ***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 (novas informações em um 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**, cabendo ao programador 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.

Fontes

1. DEITEL, Harvey M.; DEITEL, Paul J. Como programar em C. 2. ed. Rio de Janeiro: LTC, 1994.
2. ZIVIANI, Nivio. Projeto de algoritmos com implementações em Pascal e C. 4. ed. São Paulo: Pioneira, 1999.
3. 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.
4. PEIXOTO, Daniela Cristina Cascini. Disciplina: Lógica de programação. Curso de graduação em Engenharia de Computação – CEFET-MG, 2024.
5. CAMPOS, Luciana Maria de Assis. Disciplina: Programação orientada a objetos. Curso de graduação em Engenharia de Computação – CEFET-MG, 2024.
6. BATISTA, Natália Cosse. Disciplina: Algoritmos e estruturas de dados. Curso de graduação em Engenharia de Computação – CEFET-MG, 2025.