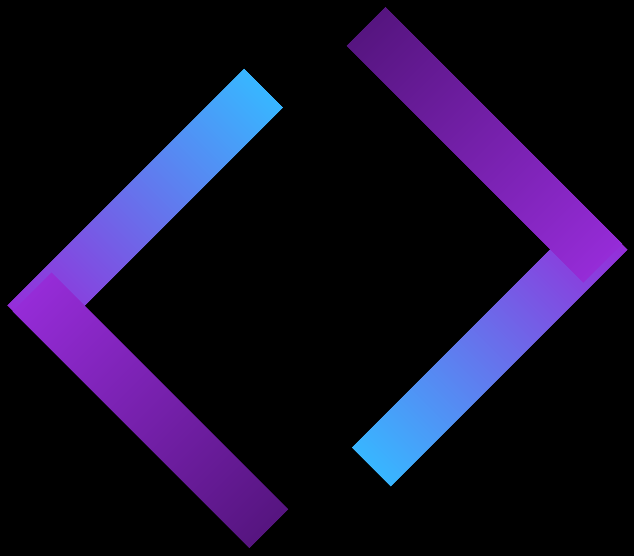
APOSTILA PET C3

C/C++

Conhecimento retido
é conhecimento
perdido.

PET C3
Ciências Computacionais



PETCode

C/C++

PET C3

Ciências Computacionais

PETCode

Indíce

PETCode

Indíce

1.Introdução à Linguagem C

1.1. O que é programação em C

C é uma das linguagens de programação mais antigas, originalmente desenvolvida por Dennis Ritchie em meados dos anos 70, com o intuito de evoluir a linguagem da época "B". A motivação para criar essa linguagem era desenvolver uma forma mais eficiente e portátil de programação para criar o sistema operacional UNIX.
Mesmo criada nos anos 70, essa linguagem só foi padronizada em 1989 com a especificação ANSI C e em 1990 com a ISO C. Desde então, tornou-se uma das linguagens mais influentes e amplamente utilizadas em todo o mundo, sendo a base para muitas outras linguagens modernas como C++, Java e C#.

1.2. Características da linguagem

C é considerada uma linguagem de nível médio - imagine como se fosse um carro com câmbio manual. Assim como um carro manual dá mais controle ao motorista (você decide quando trocar de marcha, como usar o freio motor, etc.), C dá mais controle ao programador sobre como o computador deve funcionar, mas sem exigir que você precise conhecer cada parafuso da máquina (como seria em linguagens de baixo nível).

Principais características:

- **Portabilidade:** Os programas escritos em C podem ser compilados e executados em diferentes sistemas com poucas ou nenhuma modificação. É como um livro traduzido para muitos idiomas - a história (o programa) é a mesma, mas pode ser lida (executada) em diferentes países (sistemas operacionais).

- **Eficiência**: Programas em C são extremamente rápidos e usam poucos recursos. Imagine um chef que consegue preparar um prato delicioso usando apenas os ingredientes essenciais, sem desperdício - é assim que C trabalha com os recursos do computador.
- **Flexibilidade**: Permite desde operações de alto nível (como cálculos matemáticos complexos) até operações de baixo nível (como manipulação direta de memória). É como um canivete suíço que serve tanto para tarefas simples (abrir uma carta) quanto complexas (consertar equipamentos).
- **Ampla Utilização**: C é usada em sistemas operacionais (Linux, Windows), dispositivos embarcados (micro-ondas, carros), jogos, bancos de dados e muito mais. É como o aço na construção civil - está presente nas fundações de quase tudo no mundo digital

2.Instalação e Configuração no VS Code

2.1 Instalação do compilador C

Para começar a programar em C, você precisa de um compilador - um programa que traduz seu código para a linguagem que o computador entende.

Windows:
1. Baixe o MinGW-w64 em
https://sourceforge.net/projects/mingw-w64/

   Execute o instalador e selecione:
   - Architecture: x86_64
   - Threads: posix
   - Exception: seh

2. Marque a opção "Add to PATH" durante a instalação

Linux (Debian/Ubuntu):

```
sudo apt update
sudo apt install build-essential gdb
```

macOS:

```
xcode-select --install
```

2.2 Configurando o VS Code

1. **Instale as extensões necessárias:**
   - C/C++ (Microsoft)
   - C/C++ Extension Pack
   - Code Runner

2. **Configure o ambiente:**
   - Crie uma pasta para seu projeto
   - Abra essa pasta no VS Code (File > Open Folder)
   - Crie um arquivo ola_mundo.c

2.3 Executando seu programa

1. **Compilação manual:**
   - Abra o terminal integrado (Ctrl+`)
   - Digite:

```
gcc ola_mundo.c -o ola_mundo
./ola_mundo
```

2.4 Atalhos úteis

Função	Atalho Windows/Linux	Atalho macOS
Executar programa	Ctrl+Alt+N	Control+Option+N
Depurar programa	F5	F5
Abrir terminal	Ctrl+Shift+´	Control+`
Compilar	Ctrl+Shift+B	Command+Shift+B

3. Fundamentos da Programação em C

3.1 Sintaxe Básica

A estrutura fundamental de um programa em C segue este padrão:

```c
#include <stdio.h> // Bibliotecas

int main(){     //Função principal
    printf("Olá, mundo!\n");    //Saída de texto
    return 0; //Valor de retorno da função main
}
```

Elementos essenciais:

- Todo programa C começa executando a função main()
- Instruções terminam com ponto-e-vírgula (;)
- Chaves {} delimitam blocos de código
- #include adiciona bibliotecas
- return 0 indica execução bem-sucedida

3.2 Variáveis e Tipos de Dados Declaração de Variáveis

Podemos pensar nas variáveis como “caixinhas” da memória do computador usadas para guardar informações temporariamente enquanto o programa está em execução. Cada variável tem um nome, um tipo, que determina o que ela pode armazenar e um valor, que pode sofrer alterações durante o programa.

Pense em variáveis como potinhos rotulados. Cada rótulo indica o que pode ser colocado dentro. Um potinho com o rótulo "int" só aceita números inteiros, enquanto um com o rótulo "char" guarda apenas uma letrinha. A declaração de variáveis segue a lógica abaixo, confira a tabelaX para exemplos:

```
tipo nome_da_variavel;

tipo nome_da_variavel = valor_inicial;
```

Tipo	Tamanho(bytes)	Descrição	Exemplo
int	2 ou 4	Números Inteiros	int idade = 25;
float	4	Números decimais (precisão simples)	float altura = 1,75f;
double	8	Números decimais (precisão dupla)	double pi = 3,141592;
char	1	Um caractere	char letra = ‘A’;
void	-	Sem tipo (usado em funções)	-

```c
#include <stdio.h>

int main() {
    // Declaração com inicialização
    int quantidade = 10;
    float preco = 15.99f;
    char categoria = 'A';

    // Declaração sem inicialização
    int codigo;
    double temperatura;

    // Atribuição posterior
    codigo = 1001;
    temperatura = 23.7;

    printf("Produto %d: R$ %.2f (Categoria %c)\n",
           codigo, preco, categoria);
    printf("Estoque: %d unidades\n", quantidade);
    printf("Temperatura atual: %.1f°C\n", temperatura);

    return 0;
}
```

3.3 Constantes e Operadores

3.3.1 Constantes

São valores que não podem ser alterados durante a execução, são como variáveis, mas que não mudam. Depois que você define o valor de uma constante, ele permanece o mesmo durante toda a execução do programa.

Pense em uma etiqueta de “proibido mexer” colada num potinho com valor dentro. É como o valor de pi (3.14), que sempre será o mesmo, não importa o que aconteça no programa. Podemos declarar da seguinte maneira:

```
const float PI = 3.14159f;
const int DIAS_SEMANA = 7;
#define MAX_ITENS 100  // Constante usando pré-processador
```

ou sem ponto e vírgula no final e sem declarar o tipo, usando “define” para valores pré-processados:

```
#define PI 3.14
```

3.3.2. Operadores:

Operadores são símbolos usados para realizar operações com os valores (variáveis ou constantes) no seu programa. Podem ser matemáticas, comparações, atribuições, etc. Operadores são como as ferramentas que você usa com seus potinhos de variáveis, você pode somar, comparar, multiplicar, equivale à misturar ingredientes numa receita. Eles podem ser de quatro tipos:

1. Aritméticos:

```
int a = 10, b = 3;

a + b;   // 13 (adição)
a - b;   // 7 (subtração)
a * b;   // 30 (multiplicação)
a / b;   // 3 (divisão inteira)
a % b;   // 1 (resto da divisão)
a++;     // Incremento (a = 11)
b--;     // Decremento (b = 2)
```

2. Relacionais (comparação):

Usados para comparar valores. Retornam verdadeiro (1) ou falso (0).

```
a == b;  // 0 (false) - igualdade
a != b;  // 1 (true) - diferença
a > b;   // 1 (true)
a < b;   // 0 (false)
a >= b;  // 1 (true)
a <= b;  // 0 (false)
```

3. Lógica:

Usados para combinar condições em expressões lógicas (verdadeiro/falso).

```
(a > 5) && (b < 5);  // AND lógico (1)
(a > 5) || (b > 5);  // OR lógico (1)
!(a == b);           // NOT lógico (1)
```

4. Atribuição:

Usados para atribuir ou atualizar valores de variáveis.

```
int x = 5;
x += 3;  // Equivalente a x = x + 3 (8)
x -= 2;  // x = x - 2 (6)
x *= 4;  // x = x * 4 (24)
x /= 3;  // x = x / 3 (8)
x %= 5;  // x = x % 5 (3)
```

Sintaxe básica:

A ordem que o C segue para avaliar expressões segue a mesma da matemática:

Precedência	Categoria	Operadores	Associatividade
1	Parênteses	()	-
2	Pós-fixados	expr++, expr--	Esquerda para direita
3	Unários	++expr, --expr, +, -, !, ~	Direita para esquerda
4	Multiplicação e Divisão	*, /, %	Esquerda para direita
5	Soma e Subtração	+, -	Esquerda para direita
6	Operadores relacionais simples	<, <=, >, >=	Esquerda para direita
7	Igualdade	==, !=	Esquerda para direita
8	Lógico AND	&&	Esquerda para direita
9	Lógico OR	`	-
10	Condicional (ternário)	? :	Direita para esquerda
11	Atribuição	=, +=, -=, *=, /=, etc.	Direita para esquerda
12	Vírgula (expressões separadas)	,	Esquerda para direita

4. Condicionais

Condicionais permitem que o programa tome decisões com base em determinadas condições. Imagine que você está jogando um jogo. Se o jogador tiver mais de 10 pontos, ele passa de fase. Senão, ele precisa tentar de novo. O programa precisa "verificar" isso e é aí que entram as estruturas condicionais.

4.1. If/else

É a estrutura condicional mais básica. Ela executa um bloco de código somente se uma condição for verdadeira, se não ela para ou segue outro caminho:

```
if (idade >= 18) {
    printf("Entrada permitida.\n");
} else {
    printf("Entrada negada.\n");
}
```

4.2. If/else else if

Permite testar vários caminhos:

```c
int nota = 7;

if (nota >= 9) {

    printf("Ótimo!\n");

} else if (nota >= 7) {

    printf("Bom.\n");

} else {

    printf("Precisa melhorar.\n");

}
```

4.3. switch

Usado quando você quer testar vários valores exatos de uma variável.

```c
int opcao = 2;

switch (opcao) {

    case 1:

        printf("Você escolheu 1\n");

        break;

    case 2:

        printf("Você escolheu 2\n");

        break;

    default:

        printf("Opção inválida\n");

}
```

4.4. Arrays

Um array (ou vetor) é uma estrutura que permite armazenar vários valores do mesmo tipo em uma única variável, usando índices para acessar cada valor. Imagine um conjunto de caixas numeradas (0, 1, 2, ...). Cada caixa guarda um valor, e você pode acessar qualquer uma informando o número da caixa.

Imagine que você é professor e precisa listar as notas dos seus alunos, nesse caso você vai declarar um array com a quantidade de alunos que você tem, onde cada aluno terá um índice. Para uma sala com 5 alunos:

```
int notas[5]; // Aqui estamos declarando um array com cinco espaços

int notas[5] = {8, 7, 10, 9, 0}; // O mesmo array com esses valores
```

4.5. Strings

Você já sabe como listar valores através de arrays, mas agora pense que você precisa listar os nomes dos seus alunos, para isso, vamos usar strings. Assim como no array os valores são guardados em caixas numeradas, uma string guarda as letras/caracteres formando palavras e frases.

Usando o exemplo do professor que precisa listar os nomes dos alunos, será necessário declarar uma string para cada aluno:

```
char nomes[5][50]; // Array para guardar 5 nomes, cada um com até
49 caracteres, o 0 não conta!
```

Extra! Vamos juntar os dois códigos e mostrar como eles poderiam funcionar da seguinte maneira:

Guardar o nome do aluno e sua nota, depois listar o nome do aluno e a nota:

```c
#include <stdio.h>

int main() {

    char nomes[5][50]; // Array para guardar 5 nomes

    float notas[5];    // Array para guardar 5 notas

    // Leitura dos nomes e notas

    for (int i = 0; i < 5; i++) {

        printf("Digite o nome do aluno %d: ", i + 1);

        fgets(nomes[i], sizeof(nomes[i]), stdin);

        printf("Digite a nota do aluno %d: ", i + 1);

        scanf("%f", &notas[i]);

        getchar(); // Limpa o buffer do teclado após o scanf

    }

    // Listagem dos nomes e notas

    printf("\nLista de alunos e notas:\n");

    for (int i = 0; i < 5; i++) {

        printf("%d - %sNota: %.2f\n", i + 1, nomes[i], notas[i]);

    }

    return 0;
}
```

Note que tem elementos nesse código que ainda não falamos, o FOR. Não se preocupe, isso é uma estrutura de loop e falaremos mais adiante.

4.6. Ponteiros

Agora que já vimos como podemos guardar os valores vamos conversar um pouco sobre como acessá-los, uma forma de acessar é através de ponteiros. Ponteiros são variáveis que armazenam o endereço de memória de outra variável, isso permite acessar e manipular diretamente os dados tornando o código mais flexível. Imagine que as strings e os arrays são documentos em uma pasta e os ponteiros são pessoas que realizam alguma modificação, apagam ou adicionam informações e depois guardam novamente.

Usando o exemplo acima, vamos imaginar que um aluno teve a nota colocada de forma indevida na lista e agora precisamos alterá-la:

```c
#include <stdio.h>

int main() {

    char nomes[5][50];

    float notas[5];

    // Leitura dos nomes e notas
    for (int i = 0; i < 5; i++) {

        printf("Nome do aluno %d: ", i + 1);

        fgets(nomes[i], sizeof(nomes[i]), stdin);

        printf("Nota do aluno %d: ", i + 1);

        scanf("%f", &notas[i]);

        getchar();
    }

    // Modifica a nota do aluno 1 usando ponteiro:
   // unidade *p = &variável
    float *p = &notas[0];

    printf("\nDigite a nova nota do aluno 1: ");

    scanf("%f", p);

    // Lista nomes e notas
    printf("\nAlunos e notas:\n");

    for (int i = 0; i < 5; i++) {

        printf("%d - %sNota: %.2f\n", i + 1, nomes[i], notas[i]);
    }
}
```

4.7. Funções

Pense o seguinte, você está ensinando uma pessoa a fazer um bolo, você ensina por etapas, primeiro a quebrar um ovo, depois a medir os ingredientes, para então ligar o forno e misturar tudo para assar. Se fossemos fazer um código para isso seria um tanto grande, já que vamos ensinar cada passo, porém quando ensinamos a alguém ela guarda em sua memória aquilo que aprendeu, no código isso não acontece, mas em C temos o chamado funções que são blocos de código que realizam alguma tarefa específica e podem ser reutilizada várias vezes no programa apenas chamando seu nome.

```c
#include <stdio.h>
// Função para quebrar o ovo
void quebrar_ovo() {
    printf("Quebrando o ovo...\n");
}
// Função para medir os ingredientes
void medir_ingredientes() {
    printf("Medindo os ingredientes...\n");
}
// Função para ligar o forno
void ligar_forno() {
    printf("Ligando o forno...\n");
}
// Função para misturar tudo
void misturar() {
    printf("Misturando tudo...\n");
}
// Função para assar o bolo
void assar() {
    printf("Assando o bolo...\n");
}
int main() {
    quebrar_ovo();
    medir_ingredientes();
    ligar_forno();
    misturar();
    assar();
    printf("Bolo pronto!\n");
    return 0;
}
```

5. Estruturas de Controle

Estruturas de controle permitem que o programa repita ações ou altere o fluxo de execução conforme necessário. Imagine que você está ensinando alguém a fazer uma receita e precisa repetir um passo várias vezes, como mexer a massa até ficar homogênea. Ou talvez queira executar diferentes etapas dependendo de uma escolha do usuário. As estruturas de controle, como os laços de repetição e comandos de desvio, ajudam o programa a organizar e controlar o que deve ser feito, quantas vezes e em que ordem.

5.1. Loops

Essa estrutura de controle permite que uma ação seja repetida várias vezes até que um ponto seja atingido, como bater a massa até que ela atinja o ponto X.

```c
#include <stdio.h>

int main() {

    for (massa = 0; massa <= X; massa++) {

        misturar();
    }

    return 0;
}
```

5.2. Recursividade

Essa estrutura de controle é conhecida por chamar ela mesma para resolver um problema, no exemplo do bolo, podemos usar a recursão para fazer uma ação que necessita de repetição, como colocar 2 xícaras de farinha:

```c
#include <stdio.h>

void xicara_de_farinha() {

    printf("botando farinha na xícara...\n");
}

// Função recursiva para botar a farinha X vezes
void xicara_de_farinha_recursivo(int farinha, int X) {

    if (farinha > X) {

        return;
    }

    xicara_de_farinha();

    xicara_de_farinha_recursivo(farinha + 1, X);
}
int main() {

    int X;

    printf("Quantas xícaras de farinha vai na receita? ");

    scanf("%d", &X);

    xicara_de_farinha_recursivo(1, X);

    return 0;
}
```

6. Estruturas de Dados

Estruturas de dados são formas de organizar e armazenar informações dentro do programa. Pense nelas como caixas, filas e pilhas que usamos no dia a dia. Cada tipo de estrutura é mais adequada para um problema específico. Por exemplo, se você quiser armazenar itens em ordem, pode usar uma lista é mais eficiente do que uma pilha, já se a ideia for guardar pratos empilhados em cima uns dos outros uma pilha seria suficiente Agora para organizar pessoas esperando no banco, uma fila é melhor. E para visualizar sua árvore genealógica ou uma tomada de decisões com base em perguntas de duas possíveis respostas usamos uma árvore.

6.1. Lista

Uma lista é uma coleção de elementos que podem ser acessados em sequência. Em C, podemos simular listas usando arrays ou listas encadeadas. Imagine uma lista de compras, onde cada item ocupa uma posição.

```c
#include <stdio.h>

int main() {

    int lista[5] = {10, 20, 30, 40, 50};

    for (int i = 0; i < 5; i++) {

        printf("Item %d: %d\n", i, lista[i]);

    }

    return 0;

}
```

6.2. Fila

Uma fila (queue) segue a regra FIFO (First In, First Out – o primeiro a entrar é o primeiro a sair). Imagine pessoas esperando na fila do cinema. Quem chega primeiro, compra o ingresso primeiro.

```c
#include <stdio.h>

#define TAM 5

int main() {

    int fila[TAM];

    int frente = 0, tras = 0;

    // Inserindo elementos na fila

    fila[tras++] = 1;

    fila[tras++] = 2;

    fila[tras++] = 3;

    // Removendo o primeiro da fila

    printf("Saiu da fila: %d\n", fila[frente++]);

    return 0;

}
```

6.3. Pilhas

Uma pilha segue a regra LIFO (Last In, First Out – o último a entrar é o primeiro a sair). Pense em uma pilha de pratos. O último prato colocado é o primeiro que você retira.

```c
#include <stdio.h>

#define TAM 5

int main() {

    int pilha[TAM];

    int topo = -1;

    // Empilhando (push)

    pilha[++topo] = 10;

    pilha[++topo] = 20;

    pilha[++topo] = 30;

    // Desempilhando (pop)

    printf("Saiu da pilha: %d\n", pilha[topo--]);

    return 0;

}
```

6.4. Árvores

Uma árvore é uma estrutura hierárquica, onde cada elemento é chamado de nó, e pode ter "filhos". Pense em uma árvore genealógica, onde cada pessoa pode ter filhos, e esses filhos também podem ter outros filhos.

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

struct No {
    int valor;
    struct No* esq;
    struct No* dir;
};

// Criar novo nó
struct No* novoNo(int valor) {
    struct No* no = (struct No*)malloc(sizeof(struct No));
    no->valor = valor;
    no->esq = no->dir = NULL;
    return no;
}

int main() {
    struct No* raiz = novoNo(10);
    raiz->esq = novoNo(5);
    raiz->dir = novoNo(15);
    printf("Raiz: %d\n", raiz->valor);
    printf("Esquerda: %d\n", raiz->esq->valor);
    printf("Direita: %d\n", raiz->dir->valor);
    return 0;
}
```

7. Modularidade e Bibliotecas

Quando os programas crescem, eles podem ficar difíceis de entender e manter se estiverem em um único arquivo. Para resolver isso, usamos modularidade: dividir o programa em partes menores e organizadas. Usando o exemplo do bolo citado no capítulo 5, pense que ao invés de escrever as funções (misturar, assar, quebrar_ovo…) em todos os códigos que você for usar, você pode só chamar ele referenciando um arquivo.

7.1. Criação e uso de bibliotecas

Em C, podemos criar bibliotecas para guardar funções e reutilizá-las em diferentes programas. Podem ser de dois tipos, bibliotecas padrão que já vêm com o C (ex.: stdio.h, math.h, string.h) e bibliotecas próprias, criadas pelo programador para organizar melhor o código. Para usar uma biblioteca, é só declará-la no início do código, logo após o include <stdio.h>:

```c
#include <stdio.h>
#include “quebrar_ovo”
#include “fritar_ovo”

int main() {
    printf("Vamos fazer um ovo frito\n");
    quebrar_ovo();
    fritar_ovo();
    return 0;
}
```

Porém, além do arquivo principal com o nome “quebrar_ovo.c” e “fritar_ovo.c” devemos fazer um arquivo .h chamado de arquivo cabeçalho, seria mais ou menos assim:

1. quebrar_ovo.h:

```c
#ifndef QUEBRAR_OVO_H

#define QUEBRAR_OVO_H

void quebrar_ovo();

#endif
```

2. fritar_ovo.h:

```c
#ifndef FRITAR_OVO_H

#define FRITAR_OVO_H

void fritar_ovo();

#endif
```

7.2. Organização do código fonte

Sabendo que um código com bibliotecas precisa de no mínimo 3 arquivos para rodar, é necessário realizar a organização do código fonte, a mais comum é:

1. Arquivos .h (cabeçalho) → contém declarações (funções, constantes, estruturas).
2. Arquivos .c → contêm implementações (código das funções).
3. Arquivo main.c → ponto de entrada do programa.

7.3. Modularização e reutilização de código

Modularização significa dividir o programa em módulos menores, cada um responsável por uma parte do problema. Já a reutilização de código significa criar funções e bibliotecas que possam ser usadas em diferentes programas sem precisar reescrever tudo. Isso facilita a leitura e entendimento do código, permite que várias pessoas trabalhem no mesmo projeto sem conflitos, evita repetição de código e ajuda na manutenção e correção de erros.

8. Manipulação de Arquivos

Os programas que fizemos até agora trabalham apenas enquanto estão em execução. Quando o programa termina, todas as informações são perdidas. Mas e se quisermos guardar os dados para usar depois?

É aí que entram os arquivos! Eles permitem salvar e carregar informações de forma permanente.

8.1. Leitura e escrita de arquivos

Em C, trabalhamos com arquivos usando a biblioteca “include <stdio.h>” e para abrir um arquivo, usamos a função fopen(). Ela precisa de dois parâmetros:

1. O nome do arquivo (ex: "dados.txt");
2. O modo de abertura (ex: "r", "w", "a");
   a. r = leitura (read);
   b. w = escrita (write) – cria o arquivo do zero (apaga o antigo, se existir);
   c. a = acrescentar (append) – escreve no final sem apagar o que já existe.

```c
FILE *arquivo;

arquivo = fopen("dados.txt", "w"); // abre para escrita

fprintf(arquivo, "Olá, mundo!\n"); // escreve no arquivo

fclose(arquivo); // fecha o arquivo
```

Importante: sempre feche o arquivo com fclose() depois de usá-lo!

8.2. Manipulação de arquivos de texto e binários

Podemos ler linha por linha com fgets(), ou palavra por palavra com fscanf().

Exemplo lendo um texto:

```c
#include <stdio.h>

int main() {

    FILE *arquivo;

    char linha[100];

    arquivo = fopen("dados.txt", "r");

    if (arquivo == NULL) {

        printf("Erro ao abrir o arquivo.\n");

        return 1;

    }
    while (fgets(linha, 100, arquivo) != NULL) {

        printf("%s", linha);
    }

    fclose(arquivo);

    return 0;
}
```

Esse programa abre o arquivo dados.txt e imprime seu conteúdo na tela.

8.3. Arquivos binários

Enquanto os arquivos de texto guardam dados legíveis (como frases), os arquivos binários armazenam informações em formato puro (bits).

Eles são usados quando queremos guardar estruturas, números ou dados mais complexos sem perder precisão.

Exemplo escrevendo e lendo um número em binário:

```c
#include <stdio.h>

int main() {

    FILE *arquivo;

    int numero = 42, lido;

    // Escrevendo em binário

    arquivo = fopen("dados.bin", "wb");

    fwrite(&numero, sizeof(int), 1, arquivo);

    fclose(arquivo);

    // Lendo em binário

    arquivo = fopen("dados.bin", "rb");

    fread(&lido, sizeof(int), 1, arquivo);

    fclose(arquivo);

    printf("Número lido: %d\n", lido);

    return 0;
}
```

Aqui o número 42 foi escrito em formato binário e depois recuperado exatamente como estava.

Anexo – Lista de Exercícios Resolvidos

1. Introdução à Linguagem C

Exercício 1: Escreva um programa que exiba na tela a mensagem:

Olá, C! Meu primeiro programa.

✅ Solução:

```c
#include <stdio.h>

int main() {

    printf("Olá, C! Meu primeiro programa.\n");

    return 0;

}
```

2. Instalação e Configuração

Exercício 2: Compile e execute o programa acima com o gcc.

gcc ola.c -o ola

./ola

(Esse é prático, sem código extra)

3. Fundamentos da Programação em C

Exercício 3: Declare variáveis dos tipos int, float e char, atribua valores e exiba-os.

✅ Solução:

```c
#include <stdio.h>

int main() {

    int idade = 20;

    float altura = 1.75;

    char inicial = 'A';

    printf("Idade: %d\n", idade);

    printf("Altura: %.2f\n", altura);

    printf("Inicial: %c\n", inicial);

    return 0;

}
```

Exercício 4: Crie uma constante PI e calcule a área de um círculo de raio 5.

✅ Solução:

```c
#include <stdio.h>

#define PI 3.14159

int main() {

    float raio = 5;

    float area = PI * raio * raio;

    printf("Área do círculo: %.2f\n", area);

    return 0;

}
```

4. Condicionais

Exercício 5: Leia a idade de uma pessoa e informe se ela é maior ou menor de idade.

✅ Solução:

```c
#include <stdio.h>

int main() {

    int idade;

    printf("Digite a idade: ");

    scanf("%d", &idade);

    if (idade >= 18) {

        printf("Maior de idade.\n");

    } else {

        printf("Menor de idade.\n");

    }

    return 0;

}
```

Exercício 6: Leia uma nota de 0 a 10 e classifique como Ruim, Boa ou Excelente.

✅ Solução:

```c
#include <stdio.h>

int main() {

    int nota;

    printf("Digite a nota: ");

    scanf("%d", &nota);

    if (nota >= 9) {

        printf("Excelente!\n");

    } else if (nota >= 6) {

        printf("Boa.\n");

    } else {

        printf("Ruim.\n");

    }

    return 0;

}
```

Exercício 7: Guarde o nome de 3 alunos em um array de strings e mostre-os.

✅ Solução:

```c
#include <stdio.h>

int main() {

    char nomes[3][50];

    for (int i = 0; i < 3; i++) {

        printf("Digite o nome do aluno %d: ", i + 1);

        fgets(nomes[i], sizeof(nomes[i]), stdin);

    }

    printf("\nAlunos cadastrados:\n");

    for (int i = 0; i < 3; i++) {

        printf("%s", nomes[i]);

    }

    return 0;

}
```

5. Estruturas de Controle

Exercício 8: Use um for para imprimir os números de 1 a 10.

✅ Solução:

```c
#include <stdio.h>

int main() {

    for (int i = 1; i <= 10; i++) {

        printf("%d\n", i);

    }

    return 0;

}
```

Exercício 9: Crie uma função recursiva que calcule o fatorial de um número.

✅ Solução:

```c
#include <stdio.h>

int fatorial(int n) {

    if (n == 0) return 1;

    return n * fatorial(n - 1);

}

int main() {

    int n = 5;

    printf("Fatorial de %d = %d\n", n, fatorial(n));

    return 0;

}
```

6. Estruturas de Dados

Exercício 10: Crie um array que guarde 5 números e exiba-os.

✅ Solução:

```c
#include <stdio.h>

int main() {

    int lista[5] = {10, 20, 30, 40, 50};

    for (int i = 0; i < 5; i++) {

        printf("Item %d: %d\n", i, lista[i]);

    }

    return 0;

}
```

Exercício 11: Implemente uma pilha e insira 2 elementos, depois retire 1.

✅ Solução:

```c
#include <stdio.h>

#define TAM 5

int main() {

    int pilha[TAM];

    int topo = -1;

    pilha[++topo] = 10;

    pilha[++topo] = 20;

    printf("Saiu da pilha: %d\n", pilha[topo--]);

    return 0;

}
```

7. Modularidade e Bibliotecas

Exercício 12: Crie uma função em outro arquivo que imprima "Olá de outra biblioteca!" e chame no main.c.

✅ Solução simplificada:

saudacao.h

```h
#ifndef SAUDACAO_H

#define SAUDACAO_H

void saudacao();

#endif
```

saudacao.c

```c
#include <stdio.h>

#include "saudacao.h"

void saudacao() {

    printf("Olá de outra biblioteca!\n");

}
```

main.c

```c
#include <stdio.h>

#include "saudacao.h"

int main() {

    saudacao();

    return 0;

}
```

8. Manipulação de Arquivos

Exercício 13: Grave o nome de 3 amigos em um arquivo de texto.

✅ Solução:

```c
#include <stdio.h>

int main() {

    FILE *arquivo = fopen("amigos.txt", "w");

    fprintf(arquivo, "João\nMaria\nAna\n");

    fclose(arquivo);

    printf("Arquivo criado!\n");

    return 0;

}
```

Exercício 14: Leia o arquivo amigos.txt e mostre o conteúdo na tela.

✅ Solução:

```c
#include <stdio.h>

int main() {

    FILE *arquivo = fopen("amigos.txt", "r");

    char linha[100];

    while (fgets(linha, 100, arquivo)) {

        printf("%s", linha);

    }

    fclose(arquivo);

    return 0;

}
```

PETCode

Créditos:

Andrew de Jesus Garcia
Lara Letittja Sague Lopez Guardiola Velloso
Yan Karlo da Silva Veiga Vasconcellos Dutra

PET C3
Ciências Computacionais