UNIDADE III - ESTRUTURAS DE DADOS E ARQUIVOS EM LINGUAGEM C

INTRODUÇÃO DA UNIDADE

Prezado estudante, bem-vindo à Unidade III da disciplina Nesta etapa do nosso percurso acadêmico, mergulharemos em conceitos fundamentais que constituem a espinha dorsal da programação eficiente e estruturada: as estruturas de dados, manipulação de arquivos, funções e ponteiros em linguagem C.

Você já percorreu um caminho significativo no desenvolvimento de suas habilidades de programação, dominando conceitos básicos de sintaxe, estruturas de controle e lógica algorítmica. Agora, é momento de elevar seu conhecimento a um patamar mais avançado, onde a organização dos dados e a modularização do código se tornam elementos cruciais para a construção de sistemas robustos e eficientes.

Esta unidade representa um divisor de águas em sua formação como desenvolvedor. Kernighan e Ritchie (2011) enfatizam que a linguagem C proporciona recursos avançados para manipulação eficiente de dados e gerenciamento de memória, conferindo ao desenvolvedor controle direto sobre o funcionamento dos programas. Aqui, você descobrirá como organizar dados de forma inteligente usando arrays, como persistir informações através de arquivos, como estruturar seu código através de funções e como utilizar ponteiros para otimizar performance e funcionalidade.

A relevância deste conteúdo transcende os limites acadêmicos. No mercado de trabalho, profissionais que dominam estruturas de dados e técnicas avançadas de programação são altamente valorizados. Cormen et al. (2014) demonstram que a seleção apropriada de estruturas de dados constitui fator determinante entre algoritmos eficientes e aqueles que desperdiçam recursos computacionais. Esta competência é essencial para o desenvolvimento de sistemas embarcados, aplicações de tempo real e softwares que demandam alta performance.

Os conhecimentos que você adquiriu nas unidades anteriores sobre variáveis, operadores e estruturas de controle forneceram a base sólida necessária para compreender os conceitos mais avançados que abordaremos agora. Prepare-se para uma jornada de descoberta que expandirá significativamente suas capacidades como programador e o preparará para desafios mais complexos no desenvolvimento de sistemas.

DESENVOLVIMENTO DO CONTEÚDO

3.1 Estruturas de Dados Básicas (Homogêneas)

Introdução às Estruturas de Dados

As estruturas de dados representam formas organizadas de armazenar e manipular informações em um programa. Tenenbaum et al. (2013) conceituam estruturas de

dados como conjuntos organizados de informações caracterizados pelas operações específicas e pelos relacionamentos lógicos estabelecidos entre seus componentes. Em linguagem C, uma das estruturas de dados homogêneas mais fundamentais é o **array**.

O que são Estruturas de Dados?

Estruturas de dados são formas organizadas e específicas de armazenar, organizar e manipular informações dentro de um programa de computador. Elas são fundamentais para que possamos trabalhar de maneira eficiente com grandes volumes de dados, garantindo rapidez e facilidade no acesso, modificação e armazenamento dessas informações.

Tenenbaum et al. (2013) conceituam estrutura de dados como uma coleção organizada de dados, caracterizada não apenas pelos dados em si, mas também pelas operações que podem ser realizadas sobre eles e pela forma como esses dados se relacionam entre si.

Por que usamos Estruturas de Dados?

- **Organização:** Permitem armazenar dados relacionados de forma estruturada, evitando a criação de muitas variáveis soltas, que dificultam o entendimento e a manutenção do código.
- **Eficiência:** Facilitam o acesso e manipulação rápida dos dados, tornando programas mais eficientes.
- **Reutilização:** Estruturas bem definidas podem ser usadas em diferentes partes do programa ou até em diferentes projetos.
- **Resolução de Problemas:** Muitos problemas computacionais são resolvidos de forma natural usando estruturas adequadas, como filas, pilhas, listas e arrays.

Tipos de Estruturas de Dados

Existem diversos tipos de estruturas de dados, mas aqui focaremos nas mais básicas e homogêneas, ou seja, aquelas que armazenam dados do mesmo tipo, como:

- **Arrays (ou vetores):** armazenam elementos em sequência linear.
- **Matrizes:** arrays multidimensionais que organizam dados em linhas e colunas.

Array é o mesmo que vetor?

Sim! Em programação, especialmente em C, os termos **array** e **vetor** são usados quase como sinônimos, e ambos se referem a uma estrutura que armazena múltiplos elementos do mesmo tipo, organizados de forma sequencial na memória.

- **Array** é o termo oficial usado na linguagem C e em documentações técnicas.

- **Vetor** é uma palavra comum no português, muito usada em contextos educacionais para facilitar o entendimento.

Arrays e Vetores: Conceito Básico

Em C, o **array** é a estrutura mais simples e fundamental para armazenar múltiplos elementos do mesmo tipo, organizados sequencialmente na memória. Como o termo técnico usado na linguagem é *array*, na linguagem portuguesa e em contextos didáticos usamos frequentemente o termo **vetor** para facilitar a compreensão.

Array e **vetor** significam a mesma coisa: uma lista ordenada de elementos do mesmo tipo, acessíveis por índices que indicam a posição de cada elemento.

Imagine que você precise armazenar as notas de 50 alunos. Sem arrays, seria necessário criar 50 variáveis diferentes, como nota1, nota2, ..., nota50. Isso deixaria o código extenso, difícil de ler e manter. Com um array, podemos criar apenas uma variável, por exemplo, notas, que agrupa todas as 50 notas em um só lugar, acessível com índices que vão de 0 a 49.

SAIBA MAIS Eficiência de Memória com Arrays Os arrays em C são armazenados em posições contíguas de memória, o que proporciona acesso extremamente rápido aos elementos por meio de cálculos de endereço. Essa característica torna os arrays ideais para operações que exigem acesso sequencial ou aleatório frequente aos dados. Referência: https://www.gnu.org/software/gnu-c-manual/

Índices em Arrays (Vetores)

Para acessar cada elemento de um array, usamos **índices** — números que indicam a posição do elemento dentro da estrutura. Em C, os índices **começam sempre em zero**. Isso significa que:

- O primeiro elemento está na posição **0**, mas para o usuário, essa é a **primeira posição** do array;
- O segundo elemento está na posição **1**, que corresponde à **segunda posição**;
- E assim por diante, até o último elemento que está na posição **tamanho do array - 1**.

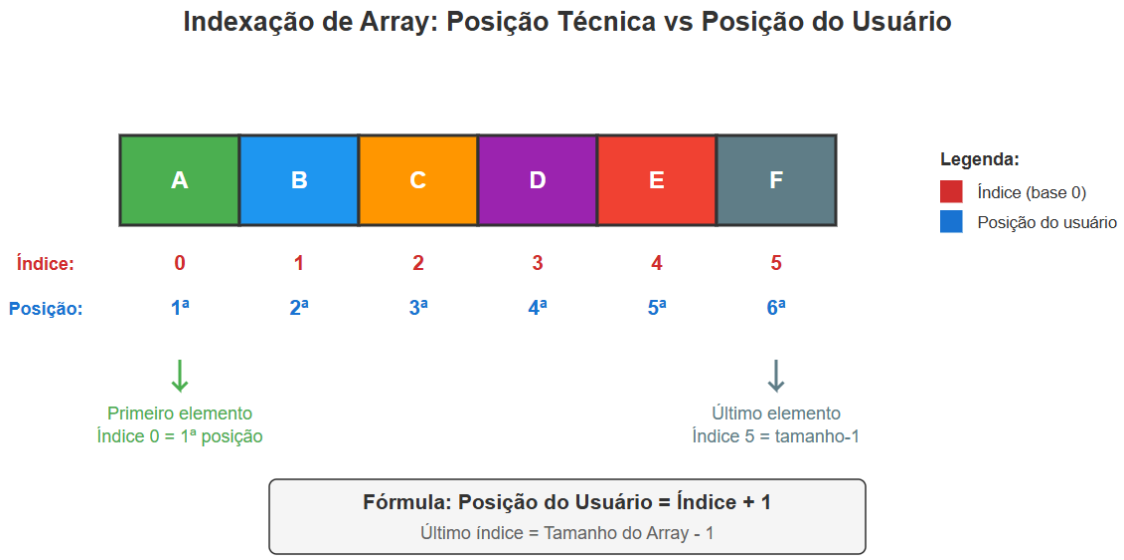


Figura 1

Fonte: próprio autor

Por exemplo, se você declara um array com 5 elementos:

int numeros[5];

os índices válidos são 0, 1, 2, 3 e 4. Para o programador, o índice 0 é o começo da contagem, mas para o usuário final, normalmente falamos que o primeiro elemento está na "posição 1".

Essa contagem começando do zero é uma característica histórica e técnica da linguagem C, que facilita o cálculo do endereço na memória para acessar qualquer elemento, já que a posição de um elemento é calculada a partir do endereço base mais o índice multiplicado pelo tamanho do tipo.

Importante: Tentar acessar um índice fora do intervalo válido, como numeros[5] ou numeros[10] neste exemplo, causa **comportamento indefinido**. Isso pode levar a:

- Acesso a dados errados;
- Quebra do programa (crash);
- Vulnerabilidades de segurança.

Por isso, é fundamental sempre garantir que os índices usados estejam dentro do limite do array, implementando validações quando necessário.

Arrays Unidimensionais

Um array unidimensional, também conhecido como vetor, é uma sequência linear de elementos do mesmo tipo. A declaração básica segue a sintaxe:

```
tipo nome_array[tamanho];
```

Exemplo prático

```
#include <stdio.h>


int main() {
   // Declaração e inicialização de um array (vetor) de inteiros
   int numeros[5] = {10, 20, 30, 40, 50};

   // Acesso aos elementos pelo índice
   printf("Primeiro elemento: %d\n", numeros[0]);
   printf("Último elemento: %d\n", numeros[4]);

   // Modificação de elementos
   numeros[2] = 35;

   // Iteração através do array
   for(int i = 0; i < 5; i++) {
      printf("Posição %d: %d\n", i, numeros[i]);
   }

   return 0;
}
```

É fundamental compreender que os índices em C começam em zero. Portanto, um array de tamanho 5 possui índices que vão de 0 a 4. Tentar acessar um índice fora desse intervalo resulta em comportamento indefinido, podendo causar erros graves no programa.

PONTO DE REFLEXÃO Limites de Arrays e Segurança Considere esta situação: você declara um array de 10 elementos, mas acidentalmente tenta acessar o índice 15. O que acontece? Como programador, que estratégias você pode implementar para

prevenir esses erros? Reflita sobre a importância da validação de índices em aplicações críticas.

Utilizando o laço for para acessar os elementos de um array (vetor)

Uma das formas mais comuns de acessar todos os elementos de um array em C é utilizando o laço de repetição **for**. Como os índices do array são números inteiros que vão de 0 até o tamanho do array menos 1, o *for* permite percorrer esse intervalo e acessar cada elemento sequencialmente.

Por exemplo, se temos um array chamado numeros com 5 elementos:

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

Podemos usar o *for* para **ler** todos os seus valores assim:

```
for (int i = 0; i < 5; i++) {
   printf("Elemento na posição %d: %d\n", i, numeros[i]);
}
```

Explicação detalhada:

- A variável i é o índice que começa em 0 (primeira posição do array);
- A condição i < 5 garante que o índice não ultrapasse o tamanho do array (que tem 5 elementos, com índices de 0 a 4);
- A cada repetição, i é incrementado em 1 (i++), passando para o próximo índice;
- Dentro do loop, numeros[i] acessa o elemento na posição atual do índice i;
- Assim, o programa imprime cada elemento do array na ordem.

Esse padrão é muito utilizado para **ler**, **modificar** ou **processar** elementos de arrays de forma prática e eficiente.

Arrays Multidimensionais

Arrays multidimensionais permitem organizar dados em estruturas mais complexas, como matrizes. O mais comum é o array bidimensional, útil para representar tabelas, grades ou matrizes matemáticas.

Também é importante que o aluno compreenda o uso de contadores para percorrer essas estruturas. **Por convenção amplamente adotada na programação, o contador i sempre se refere às linhas da matriz, enquanto o contador j se refere às colunas.** Esta padronização facilita a leitura e compreensão do código, permitindo que qualquer programador identifique imediatamente que matriz[i][j] representa o elemento localizado na linha i e coluna j da estrutura bidimensional.

Veja o exemplo abaixo:

```
#include <stdio.h>


int main() {
   // Matriz 3x3 representando um tabuleiro
   int tabuleiro[3][3] = {
      {1, 2, 3},
      {4, 5, 6},
      {7, 8, 9}
   };


   // Acesso e modificação de elementos
   printf("Elemento na posição [1][1]: %d\n", tabuleiro[1][1]);
   tabuleiro[0][0] = 0;


   // Percorrendo toda a matriz
   for(int i = 0; i < 3; i++) {
      for(int j = 0; j < 3; j++) {
         printf("%d ", tabuleiro[i][j]);
      }
      printf("\n");
   }


   return 0;
}
```

Para uma compreensão mais aprofundada sobre arrays multidimensionais e suas aplicações práticas, consulte a documentação oficial GNU C: https://www.gnu.org/software/gnu-c-manual/

IMPORTANTE Armazenamento de Arrays Multidimensionais Em C, arrays multidimensionais são armazenados em formato *row-major*, ou seja, linha por linha na memória. Um array [3][4] ocupa 12 posições consecutivas, onde os primeiros 4 elementos correspondem à primeira linha, os próximos 4 à segunda linha, e assim por diante. Essa organização impacta diretamente a performance de algoritmos que percorrem matrizes.

3.2 Manipulação de Arquivos

Conceitos Fundamentais

A manipulação de arquivos em C permite que programas persistam dados além da execução, possibilitando armazenamento permanente e compartilhamento de informações. Deitel e Deitel (2016) ressaltam a importância fundamental dos arquivos para aplicações que necessitam manter estado ou processar grandes volumes de dados.

Em C, trabalhamos com arquivos através de ponteiros para estruturas FILE, que representam fluxos de dados. A biblioteca padrão oferece funções específicas para abertura, leitura, escrita e fechamento de arquivos.

Abertura, Fechamento e Modos de Acesso

A função fopen() é responsável por abrir arquivos, retornando um ponteiro FILE ou NULL em caso de erro. Os modos de abertura determinam como o arquivo será manipulado:

```
#include <stdio.h>

#include <stdlib.h>


int main() {

   FILE *arquivo;


   // Abertura para escrita (cria ou sobrescreve)

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

   if (arquivo == NULL) {

      printf("Erro ao abrir arquivo para escrita!\n");

      return 1;

   }
```

```
    // Escrita no arquivo
    fprintf(arquivo, "Primeira linha do arquivo\n");
    fprintf(arquivo, "Segunda linha com número: %d\n", 42);

    // Fechamento obrigatório
    fclose(arquivo);

    // Abertura para leitura
    arquivo = fopen("dados.txt", "r");
    if (arquivo == NULL) {
        printf("Erro ao abrir arquivo para leitura!\n");
        return 1;
    }

    char linha[100];
    while (fgets(linha, sizeof(linha), arquivo) != NULL) {
        printf("Lido: %s", linha);
    }

    fclose(arquivo);
    return 0;
}
```

Os principais modos de abertura incluem:

- "r": leitura (arquivo deve existir)
- "w": escrita (cria ou sobrescreve)
- "a": anexar (adiciona ao final)
- "r+": leitura e escrita (arquivo deve existir)

- "w+": leitura e escrita (cria ou sobrescreve)

CASE COM O ESPECIALISTA Gerenciamento de Recursos em Aplicações Críticas Em sistemas embarcados ou aplicações de tempo real, o gerenciamento inadequado de arquivos pode causar vazamentos de recursos e instabilidade do sistema. Profissionais experientes sempre implementam verificações de erro robustas e garantem o fechamento adequado de todos os arquivos abertos, mesmo em situações de exceção.

Leitura e Escrita em Arquivos Texto em C

Manipular arquivos em C permite que um programa leia e escreva dados de forma persistente, ou seja, que as informações fiquem armazenadas mesmo depois que o programa termina. Para isso, a linguagem C oferece um conjunto de funções específicas que facilitam a manipulação de arquivos texto — arquivos onde os dados são armazenados em formato legível, como linhas e caracteres.

Funções Básicas para Arquivos Texto

- **fopen()**: abre um arquivo, podendo ser para leitura ("r"), escrita ("w") ou acréscimo ("a").
- **fclose()**: fecha o arquivo aberto, liberando recursos.
- **fgetc() / fputc()**: lê ou escreve um único caractere por vez.
- **fgets() / fputs()**: lê ou escreve uma linha ou uma string inteira (até um limite).
- **fscanf() / fprintf()**: funções formatadas que permitem ler e escrever dados com especificadores, como %d (inteiro), %f (float), %s (string), semelhantes ao scanf e printf.

Exemplo de Escrita e Leitura Estruturada em Arquivo Texto

```
#include <stdio.h>


int main() {
   FILE *arquivo;


   // Abrir arquivo para escrita (cria novo ou sobrescreve)
   arquivo = fopen("funcionarios.txt", "w");
   if (arquivo == NULL) {
      printf("Erro ao abrir arquivo para escrita.\n");
```

```
    return 1;
}

// Escrever dados no arquivo com formatação: nome idade salario
fprintf(arquivo, "Joao_Silva 30 3500.50\n");
fprintf(arquivo, "Maria_Santos 25 4200.00\n");
fprintf(arquivo, "Pedro_Costa 35 5500.75\n");

fclose(arquivo); // Sempre fechar o arquivo após o uso

// Abrir arquivo para leitura
arquivo = fopen("funcionarios.txt", "r");
if (arquivo == NULL) {
    printf("Erro ao abrir arquivo para leitura.\n");
    return 1;
}

char nome[50];
int idade;
float salario;

printf("Dados dos funcionários:\n");
// Ler dados formatados do arquivo enquanto existir linhas válidas
while (fscanf(arquivo, "%s %d %f", nome, &idade, &salario) == 3) {
    printf("Nome: %s, Idade: %d, Salário: %.2f\n", nome, idade, salario);
}

fclose(arquivo);
```

```
    return 0;
}
```

Explicações importantes:

- Usamos %s para ler strings sem espaços (por isso nomes com _), %d para inteiros e %f para números de ponto flutuante.
- fscanf() retorna o número de itens lidos corretamente; o laço continua enquanto forem 3, garantindo leitura completa.
- Sempre verificar se fopen() não retornou NULL para garantir que o arquivo foi aberto com sucesso.
- Fechar o arquivo com fclose() é fundamental para evitar perda de dados ou vazamento de recursos.

Processamento de Dados Texto - Exemplo Prático

Arquivos texto podem armazenar dados variados, como logs de acesso, onde podemos contar informações específicas:

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

int main() {
    FILE *log_file;
    char linha[200];
    int total_acessos = 0;
    int acessos_erro = 0;

    log_file = fopen("access.log", "r");
    if (log_file == NULL) {
        printf("Erro ao abrir arquivo de log!\n");
        return 1;
    }
```

```
    while (fgets(linha, sizeof(linha), log_file) != NULL) {
        total_acessos++;

        // Verifica se a linha contém códigos de erro HTTP comuns
        if (strstr(linha, " 404 ") || strstr(linha, " 500 ")) {
            acessos_erro++;
        }
    }

    fclose(log_file);

    printf("Total de acessos: %d\n", total_acessos);
    printf("Acessos com erro: %d\n", acessos_erro);
    printf("Taxa de erro: %.2f%%\n", (float)acessos_erro / total_acessos * 100);

    return 0;
}
```

Aqui, fgets() lê uma linha inteira do arquivo, e strstr() busca substrings específicas dentro dela. Esse tipo de processamento sequencial é comum em análise de arquivos texto.

Arquivos Texto x Arquivos Binários

- Arquivos **texto** armazenam dados como caracteres legíveis (ex: "123\n").
- Arquivos **binários** armazenam dados no formato bruto da memória (ex: um float diretamente).

O uso de arquivos texto facilita a leitura humana e edição manual, mas pode ser menos eficiente para grandes volumes de dados, onde arquivos binários são preferíveis.

Boas Práticas e Otimização

- Sempre **verifique erros** ao abrir arquivos.
- **Feche os arquivos** após terminar o uso.

- Para grandes volumes, considere ler e escrever em **blocos** com fread() e fwrite() para arquivos binários.
- Use buffering personalizado com setvbuf() para otimizar I/O em sistemas que exigem alto desempenho.

SAIBA MAIS Otimização de E/S em Arquivos Para aplicações que processam grandes volumes de dados, técnicas como buffering personalizado e leitura em blocos podem melhorar significativamente a performance. A função setvbuf() permite controlar o comportamento do buffer, enquanto fread() e fwrite() oferecem operações em bloco mais eficientes para dados binários. Referência: https://www.geeksforgeeks.org/c/c-programming-language/

3.3 Funções e Modularização

O que é Modularização?

Modularizar significa **dividir um programa grande em partes menores**, chamadas de **funções**, que realizam tarefas específicas. Essa prática é fundamental na programação estruturada e traz diversas vantagens:

- Deixa o código mais **organizado e fácil de entender**.
- Torna o programa **mais fácil de manter**, pois cada parte tem uma função clara.
- Permite **reutilizar trechos de código** em outras partes do programa ou em projetos diferentes.

Sommerville (2016) argumenta que a decomposição de programas em módulos bem estruturados resulta em melhorias significativas na legibilidade do código, facilita processos de manutenção e estimula a reutilização de componentes.

Conceito de Função

Uma **função em C** é um bloco de código com um nome, que pode receber dados (parâmetros), realizar operações e (opcionalmente) **retornar um resultado**.

Estrutura básica de uma função:

```
tipo_de_retorno nome_da_funcao(lista_de_parametros) {
   // Instruções
   return valor; // se houver retorno
}
```

Exemplo simples:

```
int somar(int a, int b) {
```

```
    return a + b;
}
```

Criando um Programa com Funções

Vamos ver um exemplo mais completo com três funções:

1. Uma função para calcular fatorial
2. Uma função para calcular a média de notas
3. Uma função que exibe o resultado com base na média

```
#include <stdio.h>

// Protótipos das funções
int calcular_fatorial(int n);
float calcular_media(int n1, int n2, int n3);
void exibir_resultado(float media);

int main() {
    int numero = 5;
    float media;

    // Chamando função que retorna fatorial
    int fat = calcular_fatorial(numero);
    printf("Fatorial de %d = %d\n", numero, fat);

    // Chamando função que retorna média
    media = calcular_media(80, 90, 70);

    // Chamando função void (sem retorno)
    exibir_resultado(media);
```

```
    return 0;
}

// Função para calcular fatorial (sem recursão)
int calcular_fatorial(int n) {
    int resultado = 1;
    for (int i = 1; i <= n; i++) {
        resultado *= i;
    }
    return resultado;
}

// Função para calcular a média de 3 números
float calcular_media(int n1, int n2, int n3) {
    return (n1 + n2 + n3) / 3.0;
}

// Função void para exibir resultado baseado na média
void exibir_resultado(float media) {
    printf("Média: %.2f\n", media);
    if (media >= 70) {
        printf("Status: Aprovado\n");
    } else {
        printf("Status: Reprovado\n");
    }
}
```

Passagem de Parâmetros por Valor

Quando você chama uma função passando **valores simples** (como int ou float), uma **cópia** é enviada para a função. Isso é chamado de **passagem por valor**.

```
#include <stdio.h>

void alterar_valor(int x) {
    x = 100;
    printf("Dentro da função: %d\n", x);
}

int main() {
    int numero = 50;
    printf("Antes da função: %d\n", numero);
    alterar_valor(numero);
    printf("Depois da função: %d\n", numero);
    return 0;
}
```

Note que o valor original **não muda**, pois foi feita uma cópia.

E se passarmos um array?

Em C, **arrays são passados por referência**, mesmo que pareçam por valor. Ou seja, a função **pode modificar os dados do array original**.

```
#include <stdio.h>

void alterar_array(int vet[], int tamanho) {
    vet[0] = 999;
}

int main() {
    int numeros[3] = {1, 2, 3};
    printf("Antes: %d\n", numeros[0]);
```

```
    alterar_array(numeros, 3);
    printf("Depois: %d\n", numeros[0]);
    return 0;
}
```

Neste exemplo, o valor foi realmente alterado.

IMPORTANTE Arrays como Parâmetros Embora a sintaxe sugira passagem por valor, arrays em C são sempre passados por referência. Isso significa que modificações no array dentro da função afetam o array original. Esta característica é fundamental para entender o comportamento de funções que manipulam vetores e matrizes, e será explorada mais profundamente quando estudarmos ponteiros.

Retorno de Valores em Funções

Em C, uma **função pode retornar um valor** para o ponto onde foi chamada. Esse valor pode ser um número, um caractere, uma estrutura, um ponteiro ou até mesmo void (quando a função **não retorna nada**).

Retornar valores é uma maneira fundamental de **comunicar o resultado de uma operação**, por exemplo, o resultado de uma soma, o estado de uma verificação ou o conteúdo processado de uma função.

Conceito: "Uma Função, Um Resultado"

Em C, **cada função pode retornar apenas um valor diretamente**. Por isso, quando precisamos retornar mais de um resultado, usamos outras estratégias, como ponteiros e structs (mostradas abaixo).

Exemplo 1 -- Retornando um valor simples

Vamos criar uma função que encontra o maior entre três números:

```
#include <stdio.h>


// Retorna o maior valor entre três inteiros
int encontrar_maior(int a, int b, int c) {
    int maior = a;


    if (b > maior) maior = b;
    if (c > maior) maior = c;
```

```
    return maior;
}

int main() {
    int x = 15, y = 23, z = 19;

    int resultado = encontrar_maior(x, y, z);

    printf("Maior valor: %d\n", resultado);

    return 0;
}
```

Importante: O return envia o valor calculado para quem chamou a função.

Exemplo 2 -- Retornando status (sucesso/erro)

Quando queremos saber **se uma operação foi bem-sucedida ou não**, podemos fazer a função retornar um **código de status** (geralmente 0 para erro, 1 para sucesso), e retornar o resultado por **referência** (usando ponteiro).

```
#include <stdio.h>

// Função que tenta dividir e retorna status
int dividir_seguro(float a, float b, float *resultado) {
    if (b == 0) {
        return 0; // Erro: divisão por zero
    }

    *resultado = a / b;
    return 1; // Sucesso
}
```

```
int main() {
    float resultado;

    if (dividir_seguro(10.0, 2.0, &resultado)) {
        printf("Resultado da divisão: %.2f\n", resultado);
    } else {
        printf("Erro: Divisão por zero!\n");
    }

    return 0;
}
```

Detalhes:

- A função retorna 1 se a divisão foi bem-sucedida e 0 se houve erro.
- O resultado da operação é armazenado no endereço apontado por resultado.

Exemplo 3 -- Retornando múltiplos valores com ponteiros

Como vimos, uma função em C **não consegue retornar dois valores ao mesmo tempo** diretamente. Mas é possível simular isso usando **parâmetros por referência**:

```
#include <stdio.h>

// Calcula várias operações e retorna pelos ponteiros
void operacoes(int a, int b, int *soma, int *produto, int *diferenca) {
    *soma = a + b;
    *produto = a * b;
    *diferenca = a - b;
}

int main() {
```

```
    int x = 10, y = 4;
    int soma, produto, diferenca;

    operacoes(x, y, &soma, &produto, &diferenca);

    printf("Soma: %d\n", soma);
    printf("Produto: %d\n", produto);
    printf("Diferença: %d\n", diferenca);

    return 0;
}
```

Entendendo Structs em C: Estruturas para Organizar Dados Complexos

O que são Structs?

Na linguagem C, uma struct (abreviação de *structure*, ou estrutura) é um tipo de dado definido pelo programador que permite **agrupar variáveis de diferentes tipos** sob um mesmo nome. Isso é extremamente útil quando queremos representar um **objeto real** ou uma **entidade lógica** com múltiplas características.

Pense em um struct como uma "caixa" que contém outras variáveis dentro — cada uma com um papel específico.

Por que usar Structs?

Imagine que você quer armazenar os dados de um livro em um programa. Um livro possui:

- um **código** (inteiro),
- um **título** (string),
- um **autor** (string),
- um **ano de publicação** (inteiro),
- e uma **informação de disponibilidade** (booleano: 0 ou 1).

Sem structs, você precisaria de várias variáveis separadas para cada livro — o que rapidamente se tornaria confuso e difícil de manter.

Com structs, você pode agrupar todos esses dados em **uma única unidade lógica**, como neste exemplo:

```
struct Livro {
   int codigo;
   char titulo[50];
   char autor[50];
   int ano_publicacao;
   int disponivel;
};
```

Como declarar e usar Structs

Declaração da estrutura

```
struct Pessoa {
   char nome[50];
   int idade;
   float altura;
};
```

Essa estrutura representa uma **pessoa** com nome, idade e altura.

Criação de variáveis do tipo struct

```
struct Pessoa p1;
```

Agora p1 é uma variável do **tipo Pessoa** e pode armazenar todos esses dados juntos.

Atribuição de valores

A função strcpy em C é usada para **copiar uma string** de uma origem (source) para um destino (destination). Ela está definida na biblioteca padrão <string.h>

```
strcpy(p1.nome, "Ana");
p1.idade = 28;
p1.altura = 1.65;
```

Exemplo completo

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

```
struct Pessoa {
    char nome[50];
    int idade;
    float altura;
};

int main() {
    struct Pessoa aluno;

    strcpy(aluno.nome, "Carlos");
    aluno.idade = 21;
    aluno.altura = 1.75;

    printf("Nome: %s\n", aluno.nome);
    printf("Idade: %d\n", aluno.idade);
    printf("Altura: %.2f\n", aluno.altura);

    return 0;
}
```

Structs e Arrays

Você pode criar um array de structs para armazenar várias pessoas, livros, produtos, etc.

```
struct Pessoa grupo[100]; // Vetor com 100 pessoas
```

Acessar os elementos:

```
grupo[0].idade = 30;
strcpy(grupo[1].nome, "Maria");
```

Structs com ponteiros (passagem por referência)

Quando passamos uma struct para uma função, por padrão ela é copiada (passagem por valor). Isso consome mais memória e impede que alterações persistam fora da função. A solução: **usar ponteiros**.

Exemplo com modificação por referência:

```
void aumentar_idade(struct Pessoa *p) {
   p->idade++;
}
```

Uso no main:

```
aumentar_idade(&aluno);
```

Note o uso de ->, que é equivalente a (*p).idade.

Structs dentro de Structs (aninhadas)

Você pode definir uma struct que contenha outra:

```
struct Endereco {
   char rua[50];
   int numero;
};

struct Pessoa {
   char nome[50];
   int idade;
   struct Endereco endereco;
};
```

Uso:

```
strcpy(p1.endereco.rua, "Av. Brasil");
p1.endereco.numero = 123;
```

Boas práticas ao usar Structs

- Use nomes descritivos para os campos
- Crie funções auxiliares para imprimir ou modificar dados da struct
- Prefira passar structs por ponteiro em funções

- Agrupe funções relacionadas a uma struct em um mesmo arquivo (modularização)

Aplicações práticas de Structs

- Representar **entidades reais**: alunos, clientes, produtos, veículos, funcionários;
- Modelar **registros em arquivos** (como bancos de dados simples);
- Implementar **estruturas de dados mais complexas** como listas encadeadas, árvores binárias, grafos etc.;
- Comunicação com hardware ou redes (ex: pacotes de dados).

As estruturas (structs) permitem agrupar dados de diferentes tipos em uma única entidade lógica. Este conceito é fundamental para modelar entidades do mundo real que possuem múltiplas características. Stroustrup (2013) identifica as estruturas como marco importante na evolução em direção aos paradigmas de programação orientada a dados.

```
#include <stdio.h>

#include <string.h>


// Definição da estrutura

struct Produto {

   int codigo;

   char nome[50];

   float preco;

   int estoque;

};


// Função que trabalha com struct

void exibir_produto(struct Produto p) {

   printf("Código: %d\n", p.codigo);

   printf("Nome: %s\n", p.nome);

   printf("Preço: R$ %.2f\n", p.preco);

   printf("Estoque: %d unidades\n", p.estoque);
```

```
    printf("--------------------\n");
}

// Função que modifica struct (passagem por referência)
void aplicar_desconto(struct Produto *p, float percentual) {
    p->preco *= (1 - percentual / 100);
}

int main() {
    // Inicialização de struct
    struct Produto produto1 = {1001, "Notebook", 2500.00, 15};

    // Outra forma de inicialização
    struct Produto produto2;
    produto2.codigo = 1002;
    strcpy(produto2.nome, "Mouse");
    produto2.preco = 45.90;
    produto2.estoque = 50;

    exibir_produto(produto1);
    exibir_produto(produto2);

    // Aplicando desconto
    aplicar_desconto(&produto1, 10); // 10% de desconto
    printf("Após desconto:\n");
    exibir_produto(produto1);

    return 0;
```

}

CASE COM O ESPECIALISTA Modelagem de Dados em Sistemas Reais Em um sistema de gestão hospitalar que desenvolvi, utilizamos structs complexas para representar pacientes, consultas e prontuários. A struct Paciente continha não apenas dados básicos como nome e idade, mas também ponteiros para listas de consultas e histórico médico. Esta abordagem permitiu um código mais legível e manutenível, fundamental em sistemas críticos da área de saúde.

Exemplo 4 -- Retornando uma struct (vários dados organizados)

```
#include <stdio.h>

struct Resultado {
    int soma;
    int produto;
    int diferenca;
};

// A função retorna uma struct
struct Resultado calcular(int a, int b) {
    struct Resultado r;
    r.soma = a + b;
    r.produto = a * b;
    r.diferenca = a - b;
    return r;
}

int main() {
    struct Resultado r = calcular(10, 5);

    printf("Soma: %d\n", r.soma);
```

```
    printf("Produto: %d\n", r.produto);
    printf("Diferença: %d\n", r.diferenca);

    return 0;
}
```

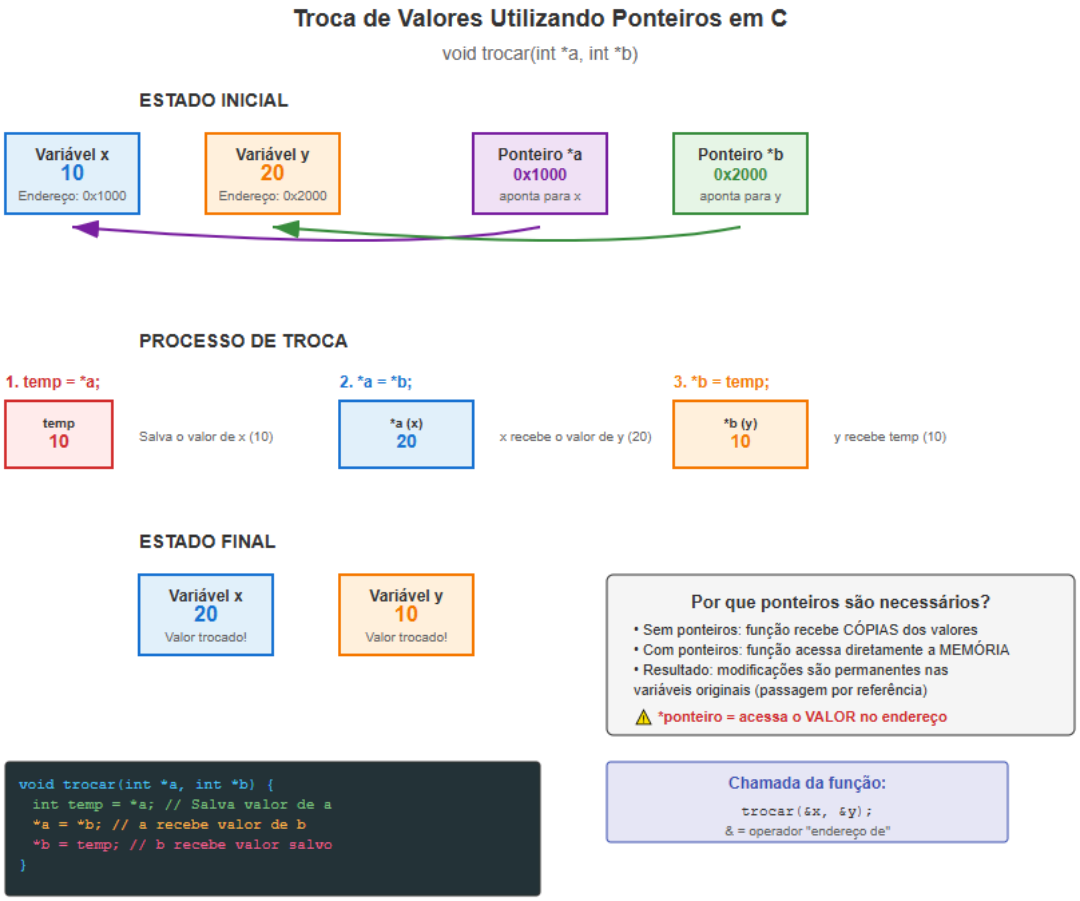
Esta abordagem é útil quando os resultados pertencem logicamente a um mesmo conjunto de dados.

PONTO DE REFLEXÃO Reutilização vs. Especialização Ao projetar funções, você enfrenta um dilema: criar funções muito genéricas (alta reutilização, mas possivelmente menos eficientes) ou funções especializadas (alta eficiência, mas menor reutilização). Como você equilibraria esses aspectos em um projeto real? Considere exemplos de sua área de interesse.

3.4 Ponteiros

Conceitos Fundamentais de Ponteiros

Ponteiros representam um dos conceitos mais poderosos e, simultaneamente, mais desafiadores da linguagem C. Kernighan e Ritchie (2011) destacam que os ponteiros oferecem ao desenvolvedor controle direto sobre o gerenciamento de memória, viabilizando otimizações substanciais e a construção de estruturas de dados sofisticadas.



Um ponteiro é uma variável que armazena o endereço de memória de outra variável. Esta capacidade permite acesso indireto a dados, passagem eficiente de parâmetros e implementação de estruturas dinâmicas.

```
#include <stdio.h>


int main() {
    int valor = 42;
    int *ponteiro = &valor;


    printf("Valor da variável: %d\n", valor);
    printf("Endereço da variável: %p\n", &valor);
    printf("Valor do ponteiro: %p\n", ponteiro);
    printf("Valor apontado pelo ponteiro: %d\n", *ponteiro);
```

```
    // Modificação através do ponteiro
    *ponteiro = 100;
    printf("Novo valor da variável: %d\n", valor);

    return 0;
}
```

Ponteiros e Arrays

A relação entre ponteiros e arrays em C é íntima e fundamental. O nome de um array é, essencialmente, um ponteiro constante para seu primeiro elemento. Esta relação permite formas elegantes e eficientes de manipular arrays.

Para aprofundar seus conhecimentos sobre ponteiros e arrays, incluindo aritmética de ponteiros, consulte: https://ocw.mit.edu/courses/6-087-practical-programming-in-c-january-iap-2010/

Passagem por Referência

A passagem por referência usando ponteiros permite que funções modifiquem variáveis do escopo chamador, superando a limitação da passagem por valor. Este mecanismo é essencial para implementar funções que precisam retornar múltiplos valores ou modificar estruturas complexas.

```
#include <stdio.h>

// Função que troca valores usando ponteiros
void trocar_valores(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Função que calcula múltiplos resultados
void calcular_operacoes(int a, int b, int *soma, int *produto, int *diferenca) {
    *soma = a + b;
```

```
    *produto = a * b;
    *diferenca = a - b;
}

int main() {
    // Exemplo de troca de valores
    int x = 10, y = 20;
    printf("Antes da troca: x = %d, y = %d\n", x, y);
    trocar_valores(&x, &y);
    printf("Depois da troca: x = %d, y = %d\n", x, y);

    // Exemplo de múltiplos retornos
    int num1 = 15, num2 = 3;
    int soma, produto, diferenca;
    calcular_operacoes(num1, num2, &soma, &produto, &diferenca);
    printf("Soma: %d, Produto: %d, Diferença: %d\n", soma, produto, diferenca);

    return 0;
}
```

PONTO DE REFLEXÃO Responsabilidade na Manipulação de Ponteiros Com grandes poderes vêm grandes responsabilidades. Os ponteiros oferecem controle direto sobre a memória, mas isso também significa que erros podem ter consequências graves. Como você garantiria a segurança ao usar ponteiros em um sistema crítico? Quais verificações implementaria?

Ponteiros para Funções

Os ponteiros para funções representam um mecanismo avançado que permite armazenar e chamar funções dinamicamente. Esta técnica é fundamental para implementar callbacks, sistemas de eventos e programação funcional em C.

Para explorar mais sobre ponteiros para funções e suas aplicações avançadas, incluindo implementação de sistemas de callback, visite o curso de Harvard CS50: https://cs50.harvard.edu/x/2025/

IMPORTANTE Aplicações Práticas de Ponteiros para Funções Em sistemas embarcados e aplicações de tempo real, ponteiros para funções são amplamente utilizados para implementar máquinas de estado, handlers de interrupção e sistemas de dispatch. Esta técnica permite código mais flexível e eficiente, especialmente em situações onde o comportamento do programa deve ser determinado dinamicamente durante a execução.

EXEMPLOS PRÁTICOS INTEGRADOS

Exemplo Prático: Sistema de Gestão de Biblioteca

Este projeto une todos os principais conceitos da linguagem C que você estudou até aqui:

- structs
- arrays
- funções
- ponteiros (implícitos em arrays e structs)
- manipulação de arquivos

Objetivo

Construiremos um **sistema simples** que permite:

- Cadastrar livros
- Listar todos os livros registrados
- Marcar livros como emprestados ou devolvidos
- Salvar todos os dados em arquivo para persistência

Etapa 1: Inclusão de bibliotecas e definição de limites

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

Essas bibliotecas nos permitem:

- stdio.h: funções de entrada e saída (como printf e scanf);
- stdlib.h: manipulação de arquivos e outras funções utilitárias;
- string.h: trabalhar com strings, como copiar ou comparar.

Agora definimos alguns limites:

```
#define MAX_LIVROS 100 // Quantidade máxima de livros no sistema
#define MAX_NOME 50 // Tamanho máximo do nome do autor ou título
```

Etapa 2: Criando a estrutura Livro

```
struct Livro {
   int codigo;
   char titulo[MAX_NOME];
   char autor[MAX_NOME];
   int ano_publicacao;
   int disponivel; // 1 = disponível, 0 = emprestado
};
```

A struct agrupa os **dados que compõem um livro**. Usamos tipos diferentes:

- int para o código, ano e status;
- char[] (strings) para título e autor.

Essa organização facilita a manipulação de muitos livros com as mesmas características.

Etapa 3: Criando o banco de dados da biblioteca

```
struct Livro biblioteca[MAX_LIVROS]; // Array que guarda até 100 livros
int total_livros = 0; // Quantos livros foram cadastrados
```

Etapa 4: Protótipos das funções

```
void carregar_biblioteca(void);
void salvar_biblioteca(void);
void adicionar_livro(void);
void listar_livros(void);
int buscar_livro(int codigo);
void emprestar_livro(void);
void devolver_livro(void);
void menu_principal(void);
```

Etapa 5: A função main

```
int main() {
```

```
    carregar_biblioteca(); // Lê os dados do arquivo, se existirem
    menu_principal(); // Mostra o menu interativo ao usuário
    salvar_biblioteca(); // Salva alterações feitas no programa
    return 0;
}
```

Aqui acontece a **orquestração** do programa. É o ponto de partida da execução.

Etapa 6: Carregando dados do arquivo

```
void carregar_biblioteca(void) {
    FILE *arquivo = fopen("biblioteca.txt", "r");

    if (arquivo == NULL) {
        printf("Arquivo de biblioteca não encontrado. Iniciando biblioteca vazia.\n");
        return;
    }

    while (fscanf(arquivo, "%d %s %s %d %d",
                &biblioteca[total_livros].codigo,
                biblioteca[total_livros].titulo,
                biblioteca[total_livros].autor,
                &biblioteca[total_livros].ano_publicacao,
                &biblioteca[total_livros].disponivel) == 5 &&
           total_livros < MAX_LIVROS) {
        total_livros++;
    }

    fclose(arquivo);
    printf("Biblioteca carregada com %d livro(s).\n", total_livros);
}
```

Esta função:

- **Abre** o arquivo em modo leitura ("r")
- **Lê os dados** usando fscanf e os armazena no array de structs
- Fecha o arquivo com fclose()

Este exemplo demonstra a integração prática de:

- **Structs** para modelar entidades complexas
- **Arrays** para armazenamento de múltiplos registros
- **Funções** para modularização e organização do código
- **Manipulação de arquivos** para persistência de dados
- **Ponteiros implícitos** através da passagem de arrays

Análise de Dados Meteorológicos

Outro exemplo prático envolvendo processamento de arquivos com estruturas multidimensionais para análise de dados meteorológicos. Este exemplo demonstra como utilizar arrays bidimensionais para representar dados temporais (meses e dias), structs para organizar informações complexas de cada dia, e funções para processamento e análise dos dados coletados.

Para explorar implementações completas destes exemplos e outros casos práticos, consulte o repositório de exemplos disponível em:
https://web.stanford.edu/class/cs107/

SÍNTESE E FECHAMENTO

Chegamos ao final de uma jornada fundamental em sua formação como desenvolvedor de sistemas. Nesta unidade, exploramos conceitos que constituem a base sólida para programação eficiente e estruturada em linguagem C. Vamos recapitular os principais pilares que você dominou.

Estruturas de Dados Homogêneas emergiram como ferramentas essenciais para organização eficiente de informações. Os arrays unidimensionais e multidimensionais não são apenas contêineres de dados, mas estruturas que permitem representar desde listas simples até matrizes complexas, fundamentais em algoritmos científicos e sistemas de processamento de dados. A compreensão profunda dessas estruturas prepara você para enfrentar desafios em processamento de imagens, simulações numéricas e análise de big data.

Manipulação de Arquivos revelou-se uma competência crucial para persistência e processamento de dados. Dominar as técnicas de leitura, escrita e processamento de

arquivos texto capacita você a desenvolver sistemas que mantêm estado, processam logs, geram relatórios e integram-se com outros sistemas. Esta habilidade é indispensável no mercado atual, onde dados são considerados o novo petróleo da economia digital.

Funções e Modularização representaram um salto qualitativo em sua capacidade de estruturar código. A arte de dividir problemas complexos em funções bem definidas não apenas melhora a legibilidade e manutenibilidade do código, mas também promove reutilização e facilita trabalho em equipe. As estruturas heterogêneas (structs) expandiram suas possibilidades de modelagem, permitindo representar entidades do mundo real com múltiplas características.

Ponteiros, talvez o conceito mais desafiador abordado, abriram as portas para programação de alto desempenho. O controle direto sobre memória, a implementação eficiente de passagem por referência e a flexibilidade dos ponteiros para funções são competências que distinguem programadores experientes. Estes conhecimentos são especialmente valiosos em desenvolvimento de sistemas embarcados, jogos e aplicações de tempo real.

A integração destes conceitos através dos exemplos práticos demonstrou como diferentes elementos da linguagem C trabalham sinergicamente para criar soluções robustas e eficientes. O sistema de gestão de biblioteca exemplificou como structs, arrays, funções e arquivos se combinam para resolver problemas reais do mundo corporativo.

Conexões Interdisciplinares emergem naturalmente destes conhecimentos. Os conceitos de estruturas de dados conectam-se diretamente com disciplinas de algoritmos e estruturas de dados avançadas, banco de dados e arquitetura de software. A manipulação de arquivos estabelece pontes com disciplinas de sistemas operacionais e redes de computadores. As técnicas de modularização preparam o terreno para programação orientada a objetos e padrões de projeto.

Preparação para Próximos Conteúdos foi cuidadosamente estabelecida. Os ponteiros que você dominou agora são prerequisitos essenciais para alocação dinâmica de memória, listas ligadas, árvores e grafos. As funções bem estruturadas que você aprendeu a criar facilitarão a compreensão de conceitos avançados como recursão, algoritmos de ordenação e programação funcional.

Você construiu uma base sólida que o capacita não apenas a resolver problemas algorítmicos complexos, mas também a projetar sistemas escaláveis e eficientes. Os conhecimentos adquiridos nesta unidade são fundamentais para qualquer carreira em desenvolvimento de software, seja em sistemas embarcados, aplicações web, inteligência artificial ou análise de dados.

Continue praticando, experimentando e desafiando-se. A programação é uma disciplina onde a teoria ganha vida através da prática constante. Os conceitos que você dominou nesta unidade são ferramentas poderosas que, combinadas com criatividade e perseverança, permitirão que você construa soluções inovadoras para os desafios tecnológicos do século XXI.

REFERÊNCIAS

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Algoritmos: teoria e prática**. 3. ed. Rio de Janeiro: Elsevier, 2014.

DEITEL, P.; DEITEL, H. **C: como programar**. 8. ed. São Paulo: Pearson Prentice Hall, 2016.

KERNIGHAN, B. W.; RITCHIE, D. M. **C: a linguagem de programação padrão ANSI**. 2. ed. Rio de Janeiro: Campus, 2011.

SCHILDT, H. **C completo e total**. 3. ed. São Paulo: Makron Books, 2015.

SOMMERVILLE, I. **Engenharia de software**. 10. ed. São Paulo: Pearson Education do Brasil, 2016.

STROUSTRUP, B. **A linguagem de programação C++**. 4. ed. Porto Alegre: Bookman, 2013.

TENENBAUM, A. M.; LANGSAM, Y.; AUGENSTEIN, M. J. **Estruturas de dados usando C**. São Paulo: Pearson Makron Books, 2013.

Recursos Web:

ASSOCIATION FOR COMPUTING MACHINERY. **ACM Digital Library**. Disponível em: https://dl.acm.org/. Acesso em: 02 jul. 2025.

GEEKSFORGEEKS. **C Programming Language Tutorial**. Disponível em: https://www.geeksforgeeks.org/c/c-programming-language/. Acesso em: 02 jul. 2025.

GNU PROJECT. **GNU C Reference Manual**. Disponível em: https://www.gnu.org/software/gnu-c-manual/. Acesso em: 02 jul. 2025.

HARVARD UNIVERSITY. **CS50: Introduction to Computer Science**. Disponível em: https://cs50.harvard.edu/x/2025/. Acesso em: 02 jul. 2025.

IEEE COMPUTER SOCIETY. **IEEE Xplore Digital Library**. Disponível em: https://ieeexplore.ieee.org/. Acesso em: 02 jul. 2025.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY. **MIT OpenCourseWare**. Disponível em: https://ocw.mit.edu/courses/6-087-practical-programming-in-c-january-iap-2010/. Acesso em: 02 jul. 2025.

MICROSOFT CORPORATION. **Microsoft C/C++ Documentation**. Disponível em: https://learn.microsoft.com/en-us/cpp/c-language/. Acesso em: 02 jul. 2025.

STANFORD UNIVERSITY. **CS107: Computer Organization & Systems**. Disponível em: https://web.stanford.edu/class/cs107/. Acesso em: 02 jul. 2025.