UNIDADE II

ESTRUTURAS DE DECISÃO, CONTROLE E REPETIÇÃO EM LINGUAGEM C

INTRODUÇÃO DA UNIDADE

Chegou a hora de deixar seus programas mais inteligentes!

Bem-vindo à segunda unidade! Aqui, seus códigos começam a ganhar mais autonomia. Imagine um semáforo que sabe quando mudar de cor, um caixa eletrônico que confere sua senha rapidinho ou um sistema que conta estoque sem você precisar se preocupar - tudo isso acontece graças às estruturas que você encontrará nesta unidade.

Essas estruturas são como o cérebro dos programas: elas permitem que seu código **tome decisões**, **escolha caminhos diferentes** e **execute tarefas repetidas automaticamente**.

Conforme observa Gaddis (2021), programas úteis e inteligentes dependem de saber mudar o fluxo das coisas conforme a situação. .

Aqui você terá à disposição conteúdos e exercícios para se familiarizar com as estruturas condicionais, desde as mais simples até as mais complexas, e com os laços que automatizam repetições - cada um pensado para resolver problemas específicos.

Conforme observam Deitel e Deitel (2024), saber usar essas estruturas é chave para pensar como um programador de verdade e criar soluções que funcionam de verdade.

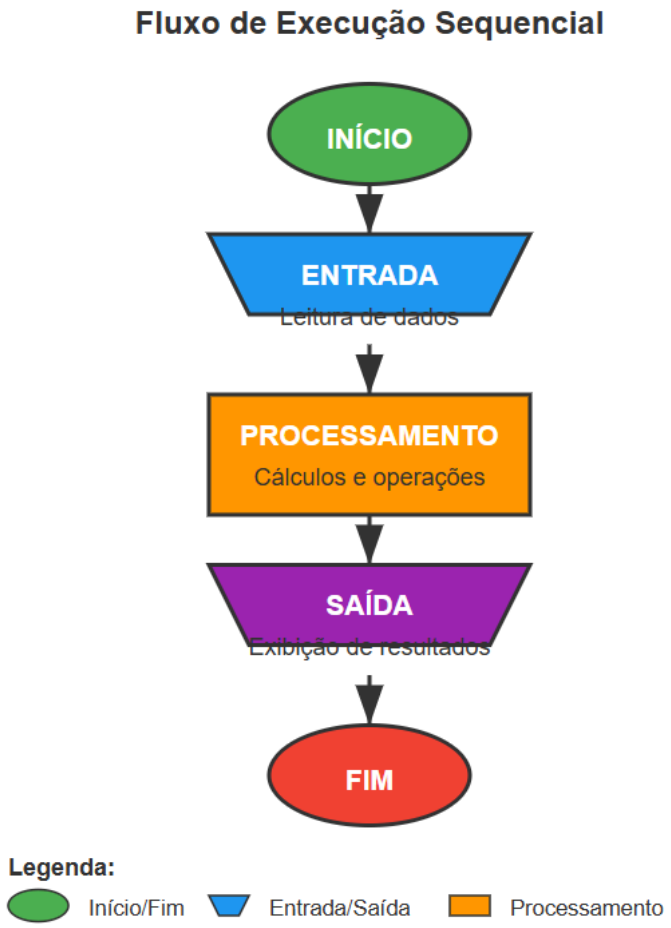
E o melhor: tudo isso usando a linguagem C, que é como aprender a base da programação no nível hard - perto do hardware, com uma sintaxe clara que abre portas para outras linguagens.

Pronto para explorar como fazer seu código pensar por você? Vamos nessa!

DESENVOLVIMENTO DO CONTEÚDO

Compreendendo o Fluxo de Controle

Quando você escreve um programa, normalmente ele executa as instruções em ordem, uma depois da outra, de cima para baixo. Isso é chamado de **execução sequencial**.



No entanto, programas do mundo real precisam ser mais flexíveis: tomar decisões com base em condições e repetir ações até que uma meta seja alcançada. Para isso, usamos o **fluxo de controle**, que permite ao programa "decidir" o que fazer e quando fazer, quebrando a sequência linear e tornando o código mais inteligente e adaptável.

Schildt (2013) explica que as estruturas de controle permitem especificar a ordem de execução das instruções no programa, tornando possível criar algoritmos complexos e eficientes.

Em outras palavras, o fluxo de controle permite instruções como:

- **"Se acontecer isso, faça aquilo; senão, faça outra coisa."**
- **"Repita essa tarefa até que uma condição seja satisfeita."**

Exemplo prático: sistema de login

Imagine um sistema de login. O programa verifica se a senha digitada está correta:

- Se estiver, permite o acesso.
- Se não estiver, solicita nova tentativa - repetindo isso até o usuário acertar ou desistir.

Sem o fluxo de controle, o programa apenas seguiria linha por linha, sem capacidade de decisão ou repetição. O fluxo de controle representa o mecanismo fundamental que permite aos programas **tomar decisões** através de **estruturas de decisão** (condicionais) e **executar tarefas repetitivas** através de **laços de repetição** (estruturas de repetição). As **estruturas de decisão**, como if-else e switch, permitem que o programa escolha diferentes caminhos de execução baseados em condições específicas, enquanto os **laços de repetição**, como for, while e do-while, possibilitam a execução repetida de blocos de código até que determinadas condições sejam satisfeitas. Estes dois pilares do controle de fluxo transformam programas lineares em soluções dinâmicas capazes de resolver problemas complexos e adaptar-se a diferentes situações durante a execução.

Estruturas de Decisão: Como Tomar Decisões em um Programa

As **estruturas de decisão** (ou **estruturas condicionais**) permitem ao programa escolher entre caminhos diferentes com base em uma condição. Isso torna o software adaptável a diferentes situações.

Entendendo a Estrutura if/else em C

A estrutura condicional mais comum é o if. Ela avalia uma **expressão booleana** - ou seja, uma condição que pode ser verdadeira (true) ou falsa (false).

- Se a condição for **verdadeira**, um bloco de código é executado.
- Se for **falsa** e houver um else, outro bloco é executado.

Estrutura geral:

```
if (condição) {
   // Código executado se a condição for verdadeira
} else {
   // Código executado se a condição for falsa
}
```

Exemplo: Verificação de idade

Leitura da idade:

- O programa exibe a mensagem "Digite sua idade: " e espera o usuário digitar um número inteiro.
- Esse número é lido com scanf("%d", &idade); e armazenado na variável idade.

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

```
int main() {
   int idade;

   printf("Digite sua idade: ");
   scanf("%d", &idade);

   if (idade >= 18) {
      printf("Acesso liberado: você é maior de idade.\n");
   } else {
      printf("Acesso negado: você é menor de idade.\n");
   }

   return 0;
}
```

O que acontece nesse programa?

1. **Leitura da idade**: o programa pede ao usuário para digitar a idade.
2. **Avaliação da condição**: idade >= 18 é verdadeiro ou falso?
3. **Execução condicional**: a. Se **verdadeiro**, imprime: "Acesso liberado...". b. Se **falso**, imprime: "Acesso negado...".

Uso do \n:

- O \n ao final das mensagens serve para **quebrar a linha**, ou seja, move o cursor para a próxima linha no terminal após a mensagem ser exibida.
- Isso melhora a formatação da saída, deixando o texto mais organizado.

Condicionais Aninhadas: Decisões Mais Detalhadas

Às vezes, é preciso verificar mais de uma condição. Para isso, podemos usar um if dentro de outro if - o que chamamos de **condicional aninhada**.

Exemplo:

```
if (idade >= 18) {
```

```
    if (idade < 65) {
        printf("Adulto.\n");
    } else {
        printf("Idoso.\n");
    }
} else {
    printf("Menor de idade.\n");
}
```

Esse tipo de estrutura permite decisões hierárquicas, mas deve ser usada com cuidado para não dificultar a leitura do código.

Operadores Lógicos: Combinando Condições

Às vezes, uma única condição não é suficiente. É possível **combinar várias condições** usando operadores lógicos:

Operador	Significado	Exemplo
&&	**E lógico**	idade >= 18 && renda > 1000
||	**OU lógico**	idade < 18 || renda < 500
!	**NÃO lógico (negação)**	!(idade >= 18)

Esses operadores retornam verdadeiro ou falso com base na combinação das condições envolvidas.

Exemplo com operadores lógicos:

```
if (idade >= 18 && idade < 65) {
    printf("Você está na idade adulta.\n");
} else {
    printf("Você é jovem ou idoso.\n");
}
```

Condicionais Compostas

Em muitos casos, enfrentamos situações que exigem múltiplas verificações combinadas. Essas situações frequentemente requerem o uso de **estruturas condicionais aninhadas** (também conhecidas como **if aninhado**), onde uma condição é testada dentro de outra condição, criando níveis hierárquicos de decisão.. Veja o

exemplo abaixo, que usa if, else if e else para simular um sistema de aprovação de cartão de crédito:

```
if (idade >= 18 && renda >= 2000) {
   printf("Aprovado para o cartão de crédito premium.\n");
} else if (idade >= 18 && renda >= 1000) {
   printf("Aprovado para o cartão de crédito básico.\n");
} else {
   printf("Não aprovado para cartão de crédito.\n");
}
```

Neste exemplo:

- O programa analisa idade e renda para decidir qual cartão oferecer (ou se deve negar).
- As verificações ocorrem **em ordem**, da mais restrita à mais geral.

DICA: Operadores de Comparação em C

Os principais operadores de comparação são:

Operador	**Significado**
==	Igual a
!=	Diferente de
<	Menor que
>	Maior que
<=	Menor ou igual a
>=	Maior ou igual a

Atenção: Um erro comum é usar = (atribuição) no lugar de == (comparação).

Para mais detalhes, consulte:
https://en.cppreference.com/w/c/language/operator_precedence

Resumo

- **Fluxo de controle** permite ao programa tomar decisões e repetir ações.
- **if/else** permite executar diferentes blocos com base em condições.

- **Condicionais aninhadas** e **compostas** permitem lidar com situações mais complexas.
- **Operadores lógicos** ajudam a combinar múltiplas condições.

Com essas ferramentas, você começa a criar programas verdadeiramente inteligentes e úteis!

A Estrutura switch/case: Múltiplas Alternativas

Quando precisamos comparar uma variável com múltiplos valores possíveis, a estrutura **switch/case** oferece uma alternativa mais elegante e eficiente do que várias estruturas if/else if.

Feofiloff (2014) observa que a estrutura switch é particularmente útil quando temos muitas alternativas baseadas no valor de uma única variável.

Como funciona o switch

O **comando switch** é uma estrutura de controle usada para **escolher entre várias opções com base no valor de uma variável**. Ele é mais organizado e legível do que usar vários if-else encadeados, especialmente quando estamos testando **diferentes valores fixos** de uma variável.

- O switch avalia o valor de uma variável (normalmente int ou char).
- Em seguida, compara esse valor com os **valores definidos em cada case**.
- Quando encontra uma correspondência, executa o código dentro daquele case.
- O comando break é usado para **encerrar o switch após executar o bloco correspondente**.
- Se nenhum case for correspondente, e houver um default, o código dentro do default é executado.

Exemplo de uso: Menu de opções

```
#include <stdio.h>


int main() {
   int opcao;


   printf("Menu:\n");
   printf("1 - Novo jogo\n");
```

```
    printf("2 - Carregar jogo\n");
    printf("3 - Sair\n");
    printf("Escolha uma opção: ");
    scanf("%d", &opcao);

    switch(opcao) {
        case 1:
            printf("Você escolheu iniciar um novo jogo.\n");
            break;
        case 2:
            printf("Você escolheu carregar um jogo salvo.\n");
            break;
        case 3:
            printf("Saindo do jogo. Até logo!\n");
            break;
        default:
            printf("Opção inválida.\n");
            break;
    }

    return 0;
}
```

Detalhes importantes:

- **case 1:** - Se a variável opcao for igual a 1, executa o bloco que imprime "Você escolheu iniciar um novo jogo.".
- **break;** - Evita que os blocos dos próximos "case" também sejam executados (isso se chama "fall-through").
- **default:** - É executado se nenhuma das opções (1, 2, 3) for escolhida.

Dica: Sem o break, o programa continuará executando os próximos case, mesmo que eles não tenham sido escolhidos. Isso pode causar comportamentos indesejados, exceto quando **intencionalmente usado**.

Estruturas de Repetição: Automatizando Tarefas

As **estruturas de repetição**, também chamadas de **laços** (loops), permitem executar repetidamente um bloco de código enquanto uma condição for verdadeira. Elas são fundamentais para automatizar tarefas repetitivas e processar grandes volumes de dados com eficiência.

Sebesta (2018) destaca que as estruturas de repetição são essenciais para criar programas eficientes, permitindo que operações sejam realizadas em conjuntos de dados sem a necessidade de replicar código manualmente.

O Laço while: Repetição Controlada por Condição

O while (em português, enquanto) é a estrutura de repetição mais básica e versátil. Ele executa um bloco enquanto a condição for verdadeira. A condição é avaliada antes de cada iteração, então o bloco pode **não ser executado nenhuma vez** se a condição inicial for falsa.

```
#include <stdio.h>


int main() {
   int contador = 1;
   int soma = 0;


   printf("Calculando a soma dos números de 1 a 10:\n");


   // Enquanto contador for menor ou igual a 10, executa o bloco
   while (contador <= 10) {
      printf("Adicionando %d à soma\n", contador);
      soma += contador; // acumula o valor do contador na soma
      contador++; // incrementa o contador para avançar o loop
   }
```

```
    printf("Soma total: %d\n", soma);
    return 0;
}
```

Um loop infinito ocorre quando a condição de parada de uma estrutura de repetição nunca se torna falsa, fazendo com que o programa execute indefinidamente o mesmo bloco de código. Esta situação representa um dos erros de lógica mais comuns em programação e pode causar sérios problemas: consumo excessivo de recursos do sistema (CPU e memória), travamento da aplicação, lentidão ou travamento do computador, e em casos extremos, pode exigir o encerramento forçado do programa ou reinicialização do sistema.

Importante: A condição de parada deve ser alterada dentro do laço para evitar loops infinitos. Certifique-se sempre de que existe um caminho lógico que torne a condição falsa em algum momento.

O Laço do-while: Garantindo pelo menos uma execução

O do-while (em português, faça enquanto) é uma variação do laço de repetição while que garante que o bloco seja executado **pelo menos uma vez**, pois a condição é avaliada **após** a execução do bloco. Isso é útil quando a ação deve ocorrer antes de verificar a condição.

```
#include <stdio.h>

int main() {
    int numero, tentativas = 0;
    const int numero_secreto = 42;

    printf("Jogo de Adivinhação!\n");
    printf("Tente adivinhar o número entre 1 e 100:\n");

    do {
        printf("Digite um número: ");
```

```
        scanf("%d", &numero);
        tentativas++;

        if (numero < numero_secreto) {
            printf("Muito baixo! Tente novamente.\n");
        } else if (numero > numero_secreto) {
            printf("Muito alto! Tente novamente.\n");
        } else {
            printf("Parabéns! Você acertou em %d tentativas!\n", tentativas);
        }
    } while (numero != numero_secreto);

    return 0;
}
```

IMPORTANTE Diferença entre while e do-while

- while : verifica a condição **antes** de executar o bloco (pode executar zero vezes).
- do-while: verifica a condição **depois** de executar o bloco (executa pelo menos uma vez).

Escolha a estrutura baseada na lógica do problema: use do-while quando precisar que a ação ocorra pelo menos uma vez.

O Laço for: Estrutura Completa de Controle

O laço **for** (em português, para) é ideal quando sabemos exatamente quantas vezes queremos repetir uma operação. Ele reúne em uma linha a **inicialização**, a **condição de parada** e o **incremento**, tornando o código organizado e fácil de ler.

```
for (inicialização; condição; incremento/decremento) {
// bloco de código a ser repetido }
```

Como funciona o for?

- **Inicialização:** define uma variável de controle (int i = 0) executada uma vez no início.
- **Condição:** verifica se a repetição deve continuar (i < tamanho).

- **Incremento:** atualiza o contador após cada iteração (i++).

Exemplo 1: Imprimir os primeiros 10 números naturais

```
#include <stdio.h>

int main() {
   printf("Números de 1 a 10:\n");

   for (int i = 1; i <= 10; i++) {
      printf("%d ", i);
   }

   printf("\n");
   return 0;
}
```

Exemplo 2: Somar os números de 1 a 100

```
#include <stdio.h>

int main() {
   int soma = 0;

   for (int i = 1; i <= 100; i++) {
      soma += i; // soma = soma + i
   }

   printf("A soma dos números de 1 a 100 é: %d\n", soma);
   return 0;
}
```

Exemplo 3: Processar um array para calcular a média

```
#include <stdio.h>


int main() {
   int numeros[] = {5, 10, 15, 20, 25};
   int tamanho = sizeof(numeros) / sizeof(numeros[0]);
   int soma = 0;


   for (int i = 0; i < tamanho; i++) {
      soma += numeros[i];
   }


   float media = (float)soma / tamanho;
   printf("A média dos números é: %.2f\n", media);


   return 0;
}
```

PONTO DE REFLEXÃO: Quando usar o for?

- Use o for quando souber quantas vezes o laço deve rodar.
- Use while se o número de repetições depender de condições dinâmicas.
- Use do-while se quiser executar o laço ao menos uma vez antes de verificar a condição.

Controle de Fluxo Avançado

As estruturas de repetição podem ser controladas por comandos especiais que ajudam a controlar a execução de forma mais refinada: **break**, **continue** e **return**.

O Comando break: Saída imediata do laço

O break interrompe o laço completamente, mesmo que a condição de parada ainda não tenha sido atingida.

Exemplo: Encontrar um número em um array e parar ao encontrar

```c
#include <stdio.h>

int main() {
    int numeros[] = {2, 4, 6, 8, 10, 12};
    int tamanho = sizeof(numeros) / sizeof(numeros[0]);
    int busca = 8;
    int encontrado = 0;

    for (int i = 0; i < tamanho; i++) {
        if (numeros[i] == busca) {
            encontrado = 1;
            printf("Número %d encontrado na posição %d.\n", busca, i);
            break; // Sai do for assim que encontrar
        }
    }

    if (!encontrado) {
        printf("Número %d não encontrado.\n", busca);
    }

    return 0;
}
```

O Comando continue: Pula para a próxima iteração

O continue ignora o restante do código dentro do laço para a iteração atual e pula para a próxima.

Exemplo: Imprimir apenas números pares de 1 a 20

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

```
int main() {
    printf("Números pares de 1 a 20:\n");

    for (int i = 1; i <= 20; i++) {
        if (i % 2 != 0) {
            continue; // pula o resto do loop para números ímpares
        }
        printf("%d ", i);
    }

    printf("\n");
    return 0;
}
```

O Comando return: Finaliza a função imediatamente

return encerra a execução de uma função e pode retornar um valor. Na função main, return encerra o programa.

Exemplo: Validação simples com return precoce

```
#include <stdio.h>

int validarNumero(int num) {
    if (num < 0) {
        printf("Número inválido: negativo.\n");
        return 0; // Indica falha
    }
    return 1; // Indica sucesso
}

int main() {
```

```
    int numero;

    printf("Digite um número positivo: ");
    scanf("%d", &numero);

    if (!validarNumero(numero)) {
        return 1; // Encerra o programa por erro
    }

    printf("Número válido: %d\n", numero);
    return 0; // Programa encerra normalmente
}
```

SAIBA MAIS Para uma compreensão mais profunda sobre controle de fluxo em C, consulte a documentação oficial: https://en.cppreference.com/w/c/language/statements

Aplicações Práticas e Casos de Estudo

Para consolidar o aprendizado, vamos analisar alguns casos práticos que demonstram a aplicação das estruturas de controle em situações reais de desenvolvimento.

Caso 1: Sistema de Menu Interativo

Um sistema de menu é uma aplicação clássica que combina estruturas de decisão (switch/case) e repetição (do-while):

```
#include <stdio.h>

void exibirMenu() {
    printf("\n=== SISTEMA BANCÁRIO ===\n");
    printf("1. Consultar Saldo\n");
    printf("2. Depositar\n");
    printf("3. Sacar\n");
    printf("4. Sair\n");
```

```
    printf("Escolha uma opção: ");
}

int main() {
    int opcao;
    float saldo = 1000.0;
    float valor;

    do {
        exibirMenu();
        scanf("%d", &opcao);

        switch (opcao) {
            case 1:
                printf("Seu saldo atual é: R$ %.2f\n", saldo);
                break;
            case 2:
                printf("Digite o valor para depósito: R$ ");
                scanf("%f", &valor);
                if (valor > 0) {
                    saldo += valor;
                    printf("Depósito realizado! Novo saldo: R$ %.2f\n", saldo);
                } else {
                    printf("Valor inválido!\n");
                }
                break;
            case 3:
                printf("Digite o valor para saque: R$ ");
```

```
            scanf("%f", &valor);
            if (valor > 0 && valor <= saldo) {
                saldo -= valor;
                printf("Saque realizado! Novo saldo: R$ %.2f\n", saldo);
            } else {
                printf("Valor inválido ou saldo insuficiente!\n");
            }
            break;
        case 4:
            printf("Obrigado por usar nosso sistema!\n");
            break;
        default:
            printf("Opção inválida! Tente novamente.\n");
    }
  } while (opcao != 4);

  return 0;
}
```

Comentários:

- A estrutura do-while mantém o menu ativo até que o usuário escolha sair (opcao == 4).
- O switch permite tratar facilmente as diferentes opções do menu.

Caso 2: Validação de Dados com Múltiplas Tentativas

Sistemas reais exigem validação rigorosa, permitindo que o usuário corrija entradas erradas:

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

int main() {
```

```
    char senha[20];
    const char senha_correta[] = "admin123";
    int tentativas = 0;
    const int max_tentativas = 3;

    printf("Sistema de Login\n");

    while (tentativas < max_tentativas) {
        printf("Digite a senha (%d/%d tentativas): ", tentativas + 1, max_tentativas);
        scanf("%s", senha);

        if (strcmp(senha, senha_correta) == 0) {
            printf("Login realizado com sucesso!\n");
            printf("Bem-vindo ao sistema!\n");
            return 0;
        } else {
            tentativas++;
            if (tentativas < max_tentativas) {
                printf("Senha incorreta! Restam %d tentativas.\n", max_tentativas -
tentativas);
            }
        }
    }

    printf("Número máximo de tentativas excedido. Acesso bloqueado!\n");
    return 1;
}
```

Otimização e Boas Práticas

O domínio das estruturas de controle vai além de conhecer sua sintaxe - envolve também compreender quando e como usá-las de forma eficiente. Cormen et al. (2012) ressaltam que a escolha adequada de estruturas de controle pode impactar significativamente na eficiência e legibilidade do código.

Otimizando condicionais

- Ordene as condições do if/else if de acordo com a probabilidade de ocorrência (mais comuns primeiro).
- Avalie condições menos custosas antes das mais complexas, principalmente em operadores lógicos.

Para aprofundar conhecimentos sobre otimização em C, recomenda-se: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

SAIBA MAIS Complexidade Computacional Compreender a complexidade das estruturas de controle é fundamental. Um loop simples tem complexidade O(n), loops aninhados podem ter $O(n^2)$ ou maior. Esta compreensão é crucial para escrever código eficiente em aplicações que processam grandes volumes de dados.

CASE COM O ESPECIALISTA: Por Que Testar Estruturas de Controle Importa

Ao aprender C, é natural focar apenas em fazer o código "funcionar". Mas programadores experientes sabem que **código bom não é apenas aquele que funciona - é aquele que se comporta bem em qualquer situação**. E isso começa com o domínio das **estruturas de controle**.

Para garantir que if, switch, for, while e outras estruturas se comportem corretamente, programadores experientes aplicam estratégias de teste, mesmo em programas simples. E esse cuidado **faz toda a diferença no aprendizado desde o início**.

Boas práticas de quem domina C:

- **Testar valores extremos:** como 0, valores negativos, e o maior valor possível de uma variável (INT_MAX), para garantir que a lógica não falhe em situações-limite.
- **Simular entradas inválidas:** como letras onde se esperam números ou números fora do intervalo esperado.
- **Verificar condições de borda:** como o primeiro e o último elemento de um array, ou a última repetição de um loop.

Esses testes ajudam você a evitar erros comuns - como laços infinitos, decisões erradas, ou falhas silenciosas - **antes que eles virem dor de cabeça**.

Ferramentas para te ajudar: GDB

Mesmo quem está começando pode (e deve!) aprender a usar ferramentas como o **GDB (GNU Debugger)**. Com ela, você consegue:

- Executar seu código **passo a passo**.
- Ver em tempo real **o valor de cada variável**.
- Parar o programa em pontos específicos (breakpoints).
- **Entender o que o código realmente está fazendo** - não apenas o que você *acha* que ele está fazendo.

Veja mais em: https://www.gnu.org/software/gdb/documentation/

Preparação para Temas Avançados

Dominar as estruturas de controle básicas prepara você para conceitos mais avançados como recursão, ponteiros e estruturas de dados complexas. A lógica que você desenvolve ao trabalhar com loops e condicionais será fundamental quando começar a trabalhar com algoritmos de busca, ordenação e estruturas como listas ligadas e árvores.

As estruturas que estudamos nesta unidade formam a base de padrões de design mais complexos. Por exemplo, o padrão Strategy frequentemente usa estruturas switch/case, enquanto o padrão Iterator é fundamentalmente baseado em loops controlados.

SÍNTESE E FECHAMENTO

Chegamos ao fim desta unidade sobre **estruturas de decisão, controle e repetição** em C - e, mais do que aprender comandos, você desenvolveu uma mentalidade lógica que será essencial para resolver problemas reais.

O que você conquistou até aqui:

- Aprendeu a usar if, else, switch, for, while e do-while com clareza.
- Praticou o controle de fluxo com comandos como break, continue e return.

E mais importante: entendeu que...

- Esses conceitos **vão com você** para qualquer linguagem que aprender depois.
- O domínio do controle de fluxo é **universal** na programação.
- Tucker e Noonan (2017) destacam que a compreensão profunda das estruturas de controle é transferível entre linguagens e constitui uma competência fundamental do desenvolvedor.

REFERÊNCIAS

BACKES, A. **Linguagem C: completa e descomplicada**. 2. ed. Rio de Janeiro: LTC, 2018.

BÖHM, C.; JACOPINI, G. Flow diagrams, Turing machines and languages with only two formation rules. **Communications of the ACM**, v. 9, n. 5, p. 366-371, 1966.

CORMEN, T. H. et al. **Algoritmos: teoria e prática**. 3. ed. Rio de Janeiro: Elsevier, 2012.

DEITEL, P.; DEITEL, H. C How to Program. 9th ed. Global Edition. Boston: Pearson, 2024.

DIJKSTRA, E. W. Go to statement considered harmful. **Communications of the ACM**, v. 11, n. 3, p. 147-148, 1968.

FEOFILOFF, P. **Algoritmos em linguagem C**. Rio de Janeiro: Elsevier, 2014.

GADDIS, Tony. **Starting Out with Programming Logic and Design**. 6. ed. Boston: Pearson, 2021.

SCHILDT, H. **C completo e total**. 3. ed. São Paulo: Makron Books, 2013.

SEBESTA, R. W. **Conceitos de linguagens de programação**. 11. ed. Porto Alegre: Bookman, 2018.

TUCKER, A.; NOONAN, R. **Linguagens de programação: princípios e paradigmas**. 2. ed. São Paulo: McGraw-Hill, 2017.

WIRTH, N. **Algoritmos e estruturas de dados**. Rio de Janeiro: LTC, 2019.

ZIVIANI, N. **Projeto de algoritmos com implementações em Pascal e C**. 4. ed. São Paulo: Cengage Learning, 2021.

Recursos Online:

C REFERENCE. **Control Flow Statements**. Disponível em: https://en.cppreference.com/w/c/language/statements. Acesso em: 02 jul. 2025.

EMBEDDED. **Programming Embedded Systems**. Disponível em: https://www.embedded.com/programming-embedded-systems/. Acesso em: 02 jul. 2025.

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

GNU. **GDB Documentation**. Disponível em: https://www.gnu.org/software/gdb/documentation/. Acesso em: 02 jul. 2025.

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

IEEE COMPUTER SOCIETY. **Computer Society**. Disponível em: https://www.computer.org/. Acesso em: 02 jul. 2025.

ISO. **ISO/IEC 9899:2024 - Information technology — Programming languages — C**. Disponível em: https://www.iso.org/standard/82075.html. Acesso em: 02 jul. 2025.

OPENMP ARCHITECTURE REVIEW BOARD. **OpenMP Application Programming Interface**. Disponível em: https://www.openmp.org/specifications/. Acesso em: 02 jul. 2025.