

UNIP

UNIVERSIDADE PAULISTA

Programação Estruturada em C

Autor: Prof. Fábio Ferreira de Assis

Colaboradores: Prof. Angel Antonio Gonzalez Martinez
Prof. Tiago Guglielmeti Correal

Professor conteudista: Fábio Ferreira de Assis

Engenheiro da Computação pela Universidade Paulista (UNIP), licenciado em Computação pelo Centro de Ensino Claretiano, especialista em Engenharia Clínica pelo Instituto Israelita de Ensino e Pesquisa (IIEP) Albert Einstein, mestre em Engenharia da Informação pela Universidade Federal do ABC (UFABC) e doutorando em Ciências da Educação pela Universidade Aberta de Asunción (UAA).

Dados Internacionais de Catalogação na Publicação (CIP)

A848p Assis, Fábio Ferreira de.

Programação Estruturada em C / Fábio Ferreira de Assis. – São Paulo: Editora Sol, 2025.

216 p., il.

Nota: este volume está publicado nos Cadernos de Estudos e Pesquisas da UNIP, Série Didática, ISSN 1517-9230.

1. Linguagem C. 2. Programação. 3. Funções. I. Título.

CDU 681.3.062

U522.53 – 25

Prof. João Carlos Di Genio
Fundador

Profa. Sandra Rejane Gomes Miessa
Reitora

Profa. Dra. Marília Ancona Lopez
Vice-Reitora de Graduação

Profa. Dra. Marina Ancona Lopez Soligo
Vice-Reitora de Pós-Graduação e Pesquisa

Profa. Dra. Claudia Meucci Andreatini
Vice-Reitora de Administração e Finanças

Profa. M. Marisa Regina Paixão
Vice-Reitora de Extensão

Prof. Fábio Romeu de Carvalho
Vice-Reitor de Planejamento

Prof. Marcus Vinícius Mathias
Vice-Reitor das Unidades Universitárias

Profa. Silvia Renata Gomes Miessa
Vice-Reitora de Recursos Humanos e de Pessoal

Profa. Laura Ancona Lee
Vice-Reitora de Relações Internacionais

Profa. Melânia Dalla Torre
Vice-Reitora de Assuntos da Comunidade Universitária

UNIP EaD

Profa. Elisabete Brihy
Profa. M. Isabel Cristina Satie Yoshida Tonetto

Material Didático

Comissão editorial:

Profa. Dra. Christiane Mazur Doi
Profa. Dra. Ronilda Ribeiro

Apoio:

Profa. Cláudia Regina Baptista
Profa. M. Deise Alcantara Carreiro
Profa. Ana Paula Tôrres de Novaes Menezes

Projeto gráfico:

Prof. Alexandre Ponzetto

Revisão:

Vitor Andrade
Vera Saad

Sumário

Programação Estruturada em C

APRESENTAÇÃO	7
INTRODUÇÃO	8

Unidade I

1 PROGRAMAÇÃO ESTRUTURADA EM C	11
1.1 Introdução à programação estruturada em C	11
1.1.1 Estrutura de um programa em C	11
1.1.2 Tipos de dados básicos em C	14
1.1.3 Comando printf()	21
1.1.4 Comando scanf()	30
1.1.5 Exemplos do uso de printf() e scanf()	34
1.1.6 Identificadores e variáveis	38
1.1.7 Boas práticas em programação C	44
2 OPERADORES BÁSICOS EM C	51
2.1 Operadores de atribuição, aritméticos, relacionais, lógicos e de incremento/decremento	51
2.1.1 Operadores de atribuição	51
2.1.2 Operadores aritméticos	52
2.1.3 Operadores relacionais	54
2.1.4 Operadores lógicos	56
2.1.5 Operadores de incremento/decremento	57
2.1.6 Aplicações e exemplos do uso de operadores na linguagem C	58

Unidade II

3 OPERAÇÕES E CONTROLE DE FLUXO	64
3.1 Estrutura condicional: if	64
3.2 Estruturas condicionais: if – else	67
3.3 Estrutura condicional: if, else if, else (encadeadas)	74
4 ESTRUTURAS DE REPETIÇÃO (LAÇOS)	79
4.1 Laço de repetição: while	80
4.2 Laço de repetição: do – while	98
4.3 Laço de repetição: for	105
4.4 Laços de repetição aninhados: for, while e do – while	117
4.5 Comparação entre laços de repetição: for, while e do – while	127
4.6 Comando break	128
4.7 Comando continue	131
4.8 Switch – case	133

Unidade III

5 VETORES.....	144
5.1 Introdução a vetores (arrays unidimensionais).....	144
5.1.1 Declaração e preenchimento dos valores do vetor.....	145
5.1.2 Impressão de vetor.....	149
5.2 Operações com vetores.....	150
6 MATRIZES	163
6.1 Matrizes (arrays bidimensionais).....	163
6.2 Manipulação de matrizes: acesso e operações comuns.....	165

Unidade IV

7 FUNÇÕES	184
7.1 Funções em C.....	184
7.1.1 Chamada de funções.....	186
7.2 Passagem de parâmetros	188
7.3 Escopo de variáveis.....	190
7.4 Aplicações de funções em C: exemplos	194
8 PONTEIROS E MANIPULAÇÃO DE ARQUIVOS	200
8.1 Definição e uso de ponteiros em C.....	200
8.2 Manipulação de arquivos.....	203
8.2.1 Leitura e escrita em arquivos.....	205

APRESENTAÇÃO

Esta disciplina é uma das bases fundamentais para estudantes de cursos da área de tecnologia em análise e desenvolvimento de sistemas, além de outras correlatas. Seu principal objetivo é capacitar os alunos a desenvolver soluções computacionais eficazes e eficientes pelo domínio da linguagem C e da programação estruturada.

Ao longo do curso, apresentaremos os conceitos essenciais da linguagem C, explorando desde os fundamentos da estrutura de um programa até as técnicas mais avançadas de manipulação de dados, o controle de fluxo e a modularização de código. A abordagem desta disciplina permitirá que os alunos desenvolvam o raciocínio lógico, a habilidade de construir algoritmos e a competência para criar programas que realizam tarefas de forma precisa e otimizada.

Assim como a matemática e a estatística desempenham papel crucial na interpretação e na modelagem de problemas cotidianos e tecnológicos, a programação em C é a chave para entender a lógica e o funcionamento interno dos sistemas computacionais. Por meio de uma série de atividades práticas e exemplos aplicados, a disciplina abordará tópicos como estruturação de programas, manipulação de vetores e matrizes, uso de funções e ponteiros e manipulação de arquivos, proporcionando uma visão abrangente das técnicas de programação.

Além de contribuir diretamente para o desenvolvimento de habilidades técnicas, o conteúdo aprendido será útil para diversas áreas da tecnologia, como a ciência de dados, a eletrônica aplicada ao hardware computacional, a gestão de sistemas de banco de dados e a engenharia de software. O domínio da linguagem C proporcionará aos alunos uma base sólida para compreender e aplicar conceitos em outras linguagens de programação e ferramentas de desenvolvimento.

Espera-se que, ao longo deste curso, os alunos não apenas adquiram o conhecimento técnico necessário, mas também se sintam motivados a explorar o mundo da programação e suas inúmeras aplicações no contexto cotidiano e no universo tecnológico. Desejamos que a experiência seja enriquecedora e que esta disciplina abra caminhos para um entendimento mais profundo do papel da programação na solução de problemas reais e na inovação tecnológica.

Boa leitura!

INTRODUÇÃO

Escolhemos trabalhar com a linguagem C nesta disciplina devido à sua combinação de desempenho elevado, estrutura sintática objetiva e possibilidade de atuar diretamente com recursos do sistema, como memória e processador, o que oferece um aprendizado sólido e próximo da lógica computacional de baixo nível. Assim como a matemática é a base de diversas ciências, a linguagem C serve de alicerce para o desenvolvimento de muitas outras linguagens modernas, como C++, C# e Java. Seu aprendizado é um passo estratégico para qualquer profissional que busque uma carreira sólida em tecnologia e programação, uma vez que a linguagem C é amplamente usada em áreas críticas, como no desenvolvimento de sistemas operacionais e de softwares embarcados e nas aplicações de alto desempenho.

A programação estruturada em C é uma abordagem que proporciona ao programador uma forma de desenvolver soluções de forma organizada, lógica e eficiente. A linguagem C é reconhecida como uma das mais importantes e versáteis, especialmente para quem está iniciando no mundo da programação, devido à sua simplicidade, ao seu poder e à sua ampla aplicação em diferentes áreas da computação.

Desenvolvida nos anos de 1970, a linguagem C tornou-se a base de muitas outras e é amplamente utilizada em sistemas operacionais, softwares embarcados, aplicativos de alto desempenho e desenvolvimento de jogos. Um dos principais motivos de sua popularidade é a forma como ela combina a simplicidade de uma linguagem de alto nível e a capacidade de manipular diretamente recursos de hardware, oferecendo um controle preciso sobre o funcionamento dos programas.

Como dissemos na apresentação, esta disciplina foi estruturada para que os alunos possam desenvolver não apenas habilidades técnicas, mas também o raciocínio lógico e a capacidade de solucionar problemas. Ao longo do curso, a programação em C será acentuada de forma prática e contextualizada, incentivando o estudante a aplicar os conceitos em exercícios e projetos reais, promovendo o entendimento sobre como a linguagem C é utilizada para resolver problemas do cotidiano e da área tecnológica.

Este livro-texto está organizado para oferecer um aprendizado progressivo, partindo dos conceitos mais básicos e avançando até tópicos mais complexos. O objetivo é garantir que o estudante compreenda profundamente como programar de maneira estruturada, aplicando boas práticas e técnicas eficientes. Ao final do curso, espera-se que os alunos reconheçam a importância da programação em C para o desenvolvimento de soluções eficazes e compreendam como seus conhecimentos podem ser aplicados em diversas áreas da tecnologia.

Ao estudar este livro-texto, esperamos que você reconheça a importância do conteúdo da disciplina tanto para sua vida cotidiana quanto para sua carreira profissional. Além disso, desejamos que você, como futuro(a) tecnólogo(a), seja capaz de aplicar de forma prática e eficaz as técnicas abordadas ao longo do curso.

A seguir, apresentamos a organização deste livro-texto:

- **Unidade I:** introduz os conceitos básicos da programação estruturada em C, desenvolvendo o raciocínio lógico e a compreensão da estrutura de um programa simples. Acentua operações de entrada e saída de dados utilizando `printf()` e `scanf()`.
- **Unidade II:** tem foco no controle de fluxo e na repetição de ações em um programa, abordando as estruturas condicionais e `else` em suas formas simples e complexas, bem como os laços de repetição `while()`, `do-while()` e `for()`.
- **Unidade III:** os alunos aprenderão a trabalhar com vetores (arrays unidimensionais) e matrizes (arrays bidimensionais). Serão ensinadas as técnicas de declaração, inicialização e manipulação desses conjuntos de dados, além de métodos para acessar e realizar operações que facilitam o processamento de dados em programas que exigem a manipulação de múltiplos elementos.
- **Unidade IV:** aborda conceitos avançados, como a criação e o uso de funções para organizar e modularizar o código. Os alunos também explorarão o emprego de ponteiros para manipular endereços de memória e aprenderão a manipular arquivos para a leitura e a escrita de dados, permitindo que seus programas interajam de forma mais completa com o sistema operacional.

Ao concluir as quatro unidades, o estudante estará familiarizado com a base da linguagem C e estará apto a construir desde programas simples até estruturas avançadas em linguagem de programação C, adotando boas práticas de programação. Além disso, o aluno será capaz de desenvolver programas que executam ações repetitivas, tornando suas soluções mais completas e funcionais, de manipular e de processar dados utilizando vetores e matrizes, tornando-se apto a aplicar esses conceitos em problemas complexos.

Esperamos que você compreenda a relevância da programação estruturada em C para a sua formação e reconheça como as técnicas aprendidas poderão ser aplicadas em diversos contextos profissionais e acadêmicos. Bom estudo!

Unidade I

1 PROGRAMAÇÃO ESTRUTURADA EM C

Nesta unidade, iniciaremos nossa jornada com os conceitos básicos de programação em C visando desenvolver o raciocínio lógico e compreender a estrutura de um programa simples.

1.1 Introdução à programação estruturada em C

A programação estruturada, foco deste livro-texto, baseia-se em princípios como a organização do código em funções, o uso de estruturas de controle de fluxo (como condicionais e laços de repetição) e a manipulação clara e eficiente de dados. Esses conceitos permitem ao programador construir programas que não apenas executam as tarefas desejadas, mas também são mais fáceis de entender, modificar e manter.

Ao longo deste curso, você aprenderá a aplicar esses conceitos na prática, começando pela criação de programas simples e avançando para tópicos mais complexos, como o uso de vetores, matrizes, ponteiros e manipulação de arquivos. O objetivo é que você desenvolva o raciocínio lógico sólido e a habilidade de transformar ideias em soluções computacionais estruturadas e eficientes.

Dominar a programação em C é um passo vital para quem deseja seguir carreira na área de tecnologia, pois muitos dos fundamentos apresentados aqui servirão de base para o aprendizado de outras linguagens e paradigmas de programação. Ao final do curso, você estará preparado para enfrentar desafios de programação com mais confiança e competência, construindo uma sólida fundação para o seu desenvolvimento profissional.

Apresentaremos a estrutura de um programa em C e os principais tipos de dados, acentuando as boas práticas que devem ser adotadas ao programar. Também aprenderemos a realizar operações de entrada e saída de dados, inicialmente, utilizando comandos básicos como `scanf` e `printf`.

1.1.1 Estrutura de um programa em C

A estrutura básica de um programa em C segue um padrão simples, mas organizado.

A primeira parte de um programa em C geralmente inclui diretivas com o uso de `#include`, que são instruções processadas antes da compilação pelo pré-processador. A diretiva `#include` adiciona o arquivo `.h` (sendo `h` a abreviação de header, que significa "cabeçalho" em inglês) associado a uma biblioteca. Um exemplo de uso dessa diretiva para incluir uma biblioteca é: `#include <stdio.h>`.

Todo programa em C precisa ter uma função especial chamada `main()`, pois é por ela que o computador começa a executar o código. Portanto, ela é a função principal do programa. É nela que o fluxo principal do programa é definido. Dentro da função `main()`, temos o conjunto de declarações e comandos (ou instruções) que determinam o comportamento e as operações que a função executa.

As funções devem ter um tipo de retorno de valores. Funções que não são do tipo retornam valores ao chamador. No caso da função, o retorno normalmente indica o status de execução do programa (importante em sistemas operacionais). Esses conceitos iniciais serão discutidos no decorrer deste livro-texto.

Como informação introdutória útil, vale destacar que, na linguagem C, é comum o uso de parênteses após algumas palavras. Por exemplo, suponha uma função chamada de cadastrar, sem nenhum argumento. Nos exemplos de código, ela vai ser escrita como `cadastro()`, com parênteses no fim da palavra. Assim, a sintaxe da linguagem C difere dos textos convencionais, e os leitores devem ter isso em mente ao ler os exemplos do livro.

A seguir temos um exemplo de código de programa em linguagem C com uma estrutura mínima. Nele, podemos ver diretiva `#include` de um dos arquivos de cabeçalho da biblioteca padrão, o (standard i/o, ou entrada/saída padrão). Além disso, vemos a função principal `main()` sendo precedida pela palavra `int`, que indica que esse programa retorna um valor inteiro.

```
#include <stdio.h>

int main() {
    printf("Hello World!");

    return 0;
}
```

Nesse programa, temos o comando de saída `printf()` com a mensagem `Hello Word!`, que será exibida no console do sistema operacional. Por fim, o comando `return 0` indica que o programa foi executado corretamente.

O que acontece ao executar o programa do exemplo anterior?

Antes de compilar, o pré-processador inclui o conteúdo do arquivo `stdio.h`, associado à biblioteca padrão de entrada e saída, que disponibiliza, por exemplo, a função `printf()`. Sem a inclusão desse arquivo, não é possível utilizar o comando `printf()`, entre outros de entrada e saída.

A execução do programa começa pela função principal chamada `main()`. Esse nome não deve ser alterado, pois é padrão da linguagem C.

O comando `printf()` é responsável por exibir a mensagem que está entre aspas `"Hello World!"` no console. Quando o programa é executado, essa frase aparece na tela do terminal, também conhecido como console ou prompt de comando.

O comando `return 0` indica que o programa foi executado com sucesso. O valor 0 é um padrão usado para informar ao sistema operacional que não houve erros durante a execução.



Observação

Na linguagem C, a maioria dos comandos termina com um ponto e vírgula (;). Isso é usado para indicar o fim de uma instrução. O uso do ponto e vírgula é essencial para a compilação correta do código em C. Esquecer de colocá-lo pode gerar erros, enquanto usá-lo indevidamente pode resultar em comportamentos inesperados.

A seguir, temos um segundo exemplo de código em linguagem C, também com uma estrutura mínima. Diferentemente do anterior, neste caso, o tipo de retorno da função principal é `void`, o que indica que a função não retorna nenhum valor. Por esse motivo, o programa não tem o comando `return 0`; no final da execução. Isso significa que o sistema operacional não recebe nenhuma informação sobre o status da execução do programa.

Os demais elementos do código seguem o mesmo padrão demonstrado anteriormente, incluindo o arquivo de cabeçalho e o uso do comando para exibir uma mensagem no console; para as saídas com `printf` é comum o uso dos termos: texto exibido no console, saída na tela, impressão na tela ou simplesmente impressão.

```
#include <stdio.h>

void main() {
    printf("Olá Aluno!");
}
```

O que acontece ao executar o programa do último exemplo?

Ele tem comportamento similar ao programa apresentado anteriormente, com a exceção de não retornar nenhum valor, pois é do tipo `void`. Portanto:

- O programa compila e executa sem erros.
- Exibe a mensagem "Olá Aluno!" na tela.

A seguir, temos um programa vazio em C, ou seja, é um código funcional que não realiza nenhuma operação ou saída visível. Vamos analisar cada elemento do programa:

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

```
void main() {
```

```
}
```

Como visto nos exemplos anteriores, inicialmente temos a inclusão do arquivo de cabeçalho `stdio.h`, que fornece funções para entrada e saída, como `printf()` e `scanf()`.

Aqui, novamente a função principal `main()` é declarada com o tipo de retorno, indicando que o programa não retorna nenhum valor. Como já acentuado, isso significa que o sistema operacional não recebe nenhuma informação sobre o status da execução do programa. A função principal é obrigatória para que o programa seja executável, mas ela está vazia nesse caso, sem nenhum código ou instrução dentro das chaves `{ }`.

O bloco `{ }` dentro da função `main()` está vazio, indicando que o programa não realiza nenhuma operação.

O que acontece ao executar o programa do exemplo anterior?

Ele compila e roda corretamente, pois não há erros de sintaxe. Nenhuma saída será exibida no console, pois não há comandos como `printf` ou outras operações dentro da função. O programa simplesmente inicia e termina sem fazer nada perceptível ao usuário.

Por que usar um programa vazio? Ele pode ser útil para:

- **Testes básicos:** verificar se o ambiente de desenvolvimento (IDE ou compilador) está configurado corretamente.
- **Ponto de partida:** criar um esqueleto inicial para adicionar instruções e funcionalidades posteriormente.

O uso dos comandos básicos de saída na tela `printf` e de entrada de dados `scanf` será discutido depois.

1.1.2 Tipos de dados básicos em C

Trata-se de elementos essenciais na construção de qualquer programa, pois definem o tipo de valor que uma variável pode armazenar e determinam a quantidade de memória que será alocada para essa armazenagem. Uma variável pode ser comparada a uma "caixa" projetada para conter um tipo específico de conteúdo, e o tipo de dado define as características dessa caixa.

Compreender e selecionar os tipos de dados adequados é crucial para garantir que o programa funcione de forma correta e eficiente, evitando desperdício de recursos e possíveis erros.

A linguagem C oferece uma ampla gama de tipos de dados, organizados em três categorias principais:

- **Tipos primitivos:** representam os dados básicos, como números inteiros, números com ponto flutuante e caracteres.
- **Modificadores de tipos:** ajustam as propriedades dos tipos básicos, como tamanho e faixa de valores.
- **Tipos compostos:** permitem a criação de estruturas mais complexas, como arrays, structs e ponteiros.

A escolha apropriada desses tipos de dados não apenas garante a correta funcionalidade do programa, mas também contribui para sua eficiência e clareza.

Os quatro principais tipos de dados em C (tipos de dados primitivos) são:

- **char** (tipo de dado caractere): 'a', '1', '#'.
- **int** (tipo de dado inteiro): -1, 0, 1, 2, ..., 10.
- **float** (tipo de dado ponto flutuante (decimal)): -1.5, 1.0, 2.2.
- **double** (tipo de dado ponto flutuante de dupla precisão).

Os principais tipos de dados em C são conhecidos como tipos primitivos, conforme acentuado na tabela 1.

Tabela 1 – Tipos de dados primitivos em C

Tipo	Descrição	Exemplo	Tamanho (padrão)
char	Armazena um único caractere	char letra = 'A';	1 byte (8 bits)
int	Armazena números inteiros	int idade = 25;	4 bytes (32 bits)
float	Armazena números decimais de precisão simples	float pi = 3.14;	4 bytes (32 bits)
double	Armazena números decimais de precisão dupla	double pi = 3.14159;	8 bytes (64 bits)

O tamanho de alguns tipos pode variar de plataforma para plataforma e até mesmo de acordo com o compilador, como é o caso do tipo, que pode ocupar 2 bytes em algumas plataformas (alguns microcontroladores, por exemplo). Além disso, enfatizamos inicialmente que o separador decimal, no caso da linguagem C, é o ponto (.), e não a vírgula, como estamos acostumados no Brasil.

Vamos detalhar a seguir cada um dos quatro tipos básicos em C.

char (caractere)

Armazena um único caractere, como letras, dígitos ou símbolos. Um char ocupa 1 byte de memória.

Podemos declarar e atribuir um caractere a uma variável char do seguinte modo:

```
char letra = 'A';  
printf("O caractere armazenado é: %c", letra);
```

A função printf() exibe a mensagem e o caractere armazenado na variável letra na tela. No local onde temos %c será exibido A.

Podemos usar scanf("%c", &variavel); para capturar um caractere do teclado. Exemplo de entrada e saída de um char:

```
char letra;  
  
printf("Digite um caractere: ");  
scanf("%c", &letra);  
  
printf("Voce digitou: %c\n", letra);
```

O %c é utilizado no scanf() para receber um caractere (char) do usuário. Nesse exemplo, a variável letra armazena um único caractere digitado pelo usuário. No printf(), %c é usado para exibir caracteres individuais na tela.

O tipo char pode ser usado como um vetor para representar strings. Em C, uma string nada mais é do que um conjunto de caracteres armazenados em um array, sendo finalizada com o caractere nulo \0, que indica o fim da cadeia, por exemplo:

```
//String em C (finalizada automaticamente com '\0')  
char nome[] = "Programação";  
  
printf("Nome armazenado: %s\n", nome);
```

A frase em cinza após o // é um comentário. O %s é usado no printf() para exibir strings.

Em C, existem dois tipos principais de comentários, usados para documentar e explicar o código. Eles não são executados pelo compilador.

- **Comentário de uma linha**

- Usa // antes do texto. Tudo após // na mesma linha será ignorado pelo compilador.

- **Comentário de múltiplas linhas (ou em bloco)**

– Usa `/*` para iniciar e `*/` para terminar. Pode ocupar várias linhas.

Dica:

- Use comentários com moderação para explicar por que algo está sendo feito (e não apenas o que está sendo feito).
- Comentários são úteis para melhorar a legibilidade e a manutenção do código.

Lendo uma string (vetor de char)

Para armazenar e ler uma string, podemos usar: `scanf("%s", nome);`. Aqui, `nome` é um vetor de `char`, declarado, por exemplo, como:

```
char nome[20]; //suporta até 19 caracteres + '\0'

printf("Digite seu nome: ");
scanf("%19s", nome); // evita estouro de buffer

printf("Seu nome é: %s\n", nome);
```

O `scanf("%s", nome);` não lê espaços. Para ler espaços com `scanf`, utilize `scanf("%[^\\n]", nome);`. Exemplo:

```
char nome[20]; // Suporta até 19 caracteres + '\0'

printf("Digite seu nome: ");
//Lê até 19 caracteres, incluindo espaços
scanf("%19[^\n]", nome);

printf("Seu nome é: %s\n", nome);
```

O uso de `printf` e `scanf` será discutido posteriormente.

int (inteiro)

Utilizado para armazenar números inteiros, positivos ou negativos, **sem casas decimais**. A capacidade de armazenamento pode variar de acordo com o compilador e o sistema operacional, mas geralmente ocupa 4 bytes na maioria dos computadores modernos. Em sistemas mais antigos ou específicos, pode ocupar 2 bytes. Exemplo:

```
int num1 = 5, num2 = 10; // Declaração e inicialização das variáveis
```

Podemos ler variáveis do tipo inteiro e imprimir na tela, por exemplo:

```
int num1, num2; // Declaração das variáveis

// Solicitando entrada do usuário
printf("Digite dois números inteiros: ");
scanf("%d %d", &num1, &num2); // Lendo os valores

// Exibindo os valores lidos
printf("O primeiro número digitado foi: %d\n", num1);
printf("O segundo número digitado foi: %d\n", num2);
```

Explicação:

- **int num1,num2:** declaração das variáveis inteiras.
- **scanf("%d %d", &num1,&num2):** lê dois números inteiros separados por espaço. Podemos ler vários números dentro de um scanf.

float (ponto flutuante (decimal))

Usado para armazenar números com casas decimais, ou seja, valores fracionários. Um float normalmente ocupa 4 bytes de memória e tem precisão limitada (aproximadamente seis dígitos significativos), ou seja, o tipo é ideal para armazenar números reais com precisão de até seis casas decimais. Conforme já explicado, o separador decimal, no caso da linguagem C, é o ponto (.), e não a vírgula. Exemplo:

```
// Declaração e inicialização das variáveis
float notaAluno = 0;
float valor1 = 5.0, valor2 = 8.8;
float cotacaoDolar = 5.7591;

// Recebendo do usuário a nota do aluno
printf("Digite a nota do aluno: ");
scanf("%f", &notaAluno);

// Imprimindo na tela a cotação do dólar com uma casa decimal
printf("Valor da cotação do dólar: %.1f\n", cotacaoDolar);
```

No printf podemos adicionar um ponto e um valor entre o % e o f como %. 1f para definir o número de casas decimais. Caso não seja definido, serão impressas na tela todas as casas decimais do tipo da variável.

double (ponto flutuante de dupla precisão)

Semelhante ao float, mas com maior capacidade e precisão (cerca de 15 dígitos significativos). O double geralmente ocupa 8 bytes de memória e é ideal para cálculos que requerem maior precisão.

```
double raio, pi = 3.141592653589793, circunferencia;

printf("Digite o raio do círculo: ");
scanf("%lf", &raio);

circunferencia = 2 * pi * raio; // Fórmula da circunferência

printf("A circunferência do círculo é: %.5f\n", circunferencia);
```

No `scanf()` usamos `%lf` para ler variáveis do tipo `double`.

No uso do `printf`, usamos `%f` para indicar a impressão de uma variável do tipo `float` ou `double`.

O valor de π (pi) pode ser declarado como uma constante antes da função `main()`, utilizando a diretiva `#define PI 3.141592653589793`. O número do π é frequentemente usado em cálculos matemáticos, e o tipo `double` é ideal para armazená-lo devido à sua alta precisão. No entanto, armazenamos uma aproximação do número (pi), já que ele é um número irracional e não é possível armazenar suas infinitas casas na memória de um computador digital convencional.

O arredondamento de valores de ponto flutuante (`float` ou `double`) quando formatamos a saída com `printf()` segue a regra do arredondamento padrão (também chamada de arredondamento meio para cima ou "round half-up").

Se quisermos controlar manualmente o arredondamento, podemos usar a função `round()` do arquivo de cabeçalho `math.h`.

Modificadores de tipo

Eles alteram a capacidade de armazenamento ou a faixa de valores que um tipo de dado pode representar. Eles podem ser combinados com os tipos primitivos para ajustar o uso de memória e a precisão dos dados. Os principais modificadores são:

- **signed**: pode armazenar valores positivos e negativos (padrão para a maioria dos tipos inteiros).
- **unsigned**: armazena apenas valores positivos, o que permite dobrar a faixa de valores positivos que um tipo de dado pode representar.
- **short**: reduz a quantidade de memória usada, mas diminui a faixa de valores (geralmente aplicado ao tipo `int`).
- **long**: aumenta a capacidade de armazenamento de um tipo de dado, permitindo que ele represente uma faixa maior de valores.

Por exemplo, `unsigned int` pode armazenar apenas valores positivos, enquanto `long int` permite armazenar inteiros maiores que o tipo `int` padrão.

A tabela 2 elenca os tipos de dados definidos no padrão ANSI C.

Tabela 2 – Tipos de dados primitivos em C

Tipo	Formatador em C	Tamanho (em bytes)	Escala/Precisão
char	%c caractere ou %s (string)	1	-128 a 127 (signed) ou 0 a 255 (unsigned)
unsigned char	%c	1	0 a 255
signed char	%c	1	-128 a 127
int	%d ou %i	2 ou 4	-32.768 a 32.767 (16 bits) ou -2^{31} a $2^{31}-1$ (32 bits)
unsigned int	%u	2 ou 4	0 a 65.535 (16 bits) ou 0 a $2^{32}-1$ (32 bits)
signed int	%d ou %i	2 ou 4	Igual a int
short int	%hd	2	-32.768 a 32.767
unsigned short int	%hu	2	0 a 65.535
signed short int	%hd	2	Igual a short int
long int	%ld	4	-2.147.483.648 a 2.147.483.647
unsigned long int	%lu	4	0 a 4.294.967.295
signed long int	%ld	4	Igual a long int
float	%f	4	~6-7 dígitos de precisão
double	%lf	8	~15-16 dígitos de precisão
long double	%Lf	10 ou 16	~18-19 dígitos de precisão
void	-	0	Sem valores armazenados

Há um último tipo na tabela 2 que não se enquadra na definição de tipo primitivo: o void, que representa a ausência de tipo. Ele é utilizado em situações específicas, como:

- Funções que não retornam valor.
- Ponteiros genéricos (void *).
- Parâmetros vazios em funções.

Exemplo: void saudacao() indica que a função saudacao() não retorna nada.

Observe a seguir o resumo dos tipos:

- **Inteiros (int, short, long):** o formador varia com base no tipo e no tamanho, como %d para int e %ld para long int. float ocupa 4 bytes na memória; double é fixado em 8 bytes; depende do compilador: pode ser 10 bytes (x86) ou 16 bytes (x64).
- **Reais (float, double, long double):** usam, %f, %lf e %Lf para diferenciar as precisões.

- **Unsigned:** utilizam formadores específicos (%u, %lu) para evitar interpretações como valores negativos. Em sistemas de 16 bits, int pode ter 2 bytes. Em sistemas de 32 ou 64 bits, geralmente têm 4 bytes. long int. e unsigned long int podem variar entre 4 e 8 bytes dependendo do compilador e do sistema operacional.

1.1.3 Comando printf()

É uma função de saída utilizada na linguagem C para exibir informações na tela. Ele permite a formatação de texto e a apresentação de variáveis de diferentes tipos (inteiros (int), reais (float), caracteres (char) etc.) ao usuário. Por exemplo, ao usar printf("Olá Aluno!");, podemos exibir a mensagem dentro das aspas duplas na tela ou qualquer outra mensagem. O comando printf é essencial para exibir os resultados de um programa e facilitar a comunicação entre o programa e o usuário, tornando a interação mais clara e eficaz.

Resumindo: o comando printf em C é usado para exibir informações na saída padrão (geralmente o console). Ele faz parte do arquivo de cabeçalho stdio.h e permite imprimir texto, valores de variáveis e resultados formatados.

Sintaxe básica do comando `printf("texto formatado",argumentos);`:

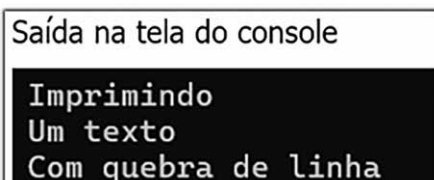
- **Texto formatado:** é a string que será exibida no console. Pode incluir texto comum, caracteres de escape (como \n para quebra de linha) e especificadores de formato (como %d para inteiros).
- **Argumentos:** são os valores ou variáveis que serão inseridos nos lugares indicados pelos especificadores de formato na string.

O exemplo a seguir ilustra um código que, após ser compilado e executado, mostra a impressão de uma mensagem com quebra de linha em cada parte separada pelo caractere \n "barra invertida n". A sintaxe do código segue o já explicado.

```
#include <stdio.h>

int main() {
    printf("Imprimindo\nUm texto\nCom quebra de linha");

    return 0;
}
```

A imagem mostra uma janela de terminal com o título "Saída na tela do console". O conteúdo exibido no terminal é: "Imprimindo", "Um texto" e "Com quebra de linha", cada uma em uma linha separada por uma quebra de linha (\n).

Saída na tela do console

```
Imprimindo
Um texto
Com quebra de linha
```

Figura 1 – Exemplo com quebra de linha

O uso de vários comandos `printf` não implica, por si só, quebras de linha. Para que a saída seja exibida em linhas separadas, é necessário incluir o caractere `\n` dentro das aspas da string.

O código apresentado anteriormente destaca como a ausência de quebras de linha `\n` e espaços no final das strings pode influenciar a saída do programa. Assim, usando a mesma estrutura do programa anterior, agora três comandos são aplicados para imprimir textos consecutivos. Nenhum dos textos tem uma quebra de linha `\n` no final. Observe que, propositalmente, não foi colocado espaço ao final das strings, causando a junção do texto.

```
#include <stdio.h>

int main() {
    printf("Imprimindo");

    printf("Um texto");

    printf("Sem quebra de linha");

    return 0;
}
```

Saída na tela

ImprimindoUm textoSem quebra de linha

Figura 2

A saída do programa pode não ser a que se esperava, pois, no mínimo, deveria ter os espaços no final dos dois primeiros comandos `printf`. Além disso, fica claro que o caractere `\n` é essencial para adicionar uma quebra de linha no final de cada texto. Sem ele, o cursor permanece na mesma linha após cada impressão, resultando na saída contínua. Para ajustar o texto, basta adicionar um espaço ao final do texto dentro dos dois primeiros `printf`.

```
printf("Imprimindo ");

printf("Um texto ");

printf("Sem quebra de linha");
```

Essas três chamadas ao `printf` exibem o texto na mesma linha, porque não há `\n` (quebra de linha) em nenhuma delas, porém foi adicionado um espaço no final dos dois primeiros comandos para separar as palavras. O resultado será:

Imprimindo Um texto Sem quebra de linha

Figura 3

E esse mesmo resultado pode ser alcançado com um único comando:

```
printf("Imprimindo Um texto Sem quebra de linha");
```

Em C, a quebra de linha só ocorre se for explicitamente inserida com `\n`. A separação de comandos `printf` no código-fonte não afeta a formatação na tela, a menos que se use `\n`.

O código apresentado ilustra a importância de atenção aos detalhes na formatação de strings em programas em C. Demonstra como pequenos ajustes nos textos podem afetar a formatação da saída. Ou seja, indica como usar caracteres de escape para formatar a saída em C. O uso de quebras de linha `\n` juntamente com tabulação `\t` é comum para organizar o texto exibido no console, seja para melhorar a legibilidade, seja para criar estruturas visuais específicas.

- **Caractere de escape `\n`:** provoca uma quebra de linha, movendo o cursor para a próxima linha no console.
- **Caractere de escape `\t`:** insere uma tabulação horizontal, criando um espaço maior entre palavras.

```
#include <stdio.h>

int main() {
    printf("Um \n\ttexto \n\t\tSimples");

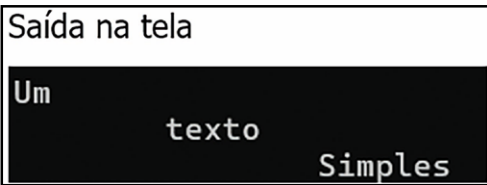
    return 0;
}
```

O que acontece ao executar o programa do exemplo anterior, com quebra de linha e uso de tabulação?

Ele compila e roda corretamente, pois não há erros de sintaxe. Analisando a saída do código, temos:

- **Primeira linha:** a palavra "Um" é impressa. O `\n` move o cursor para a próxima linha.
- **Segunda linha:** o `\t` insere uma tabulação antes da palavra "texto". Após "texto", outro `\n` move o cursor para a próxima linha.
- **Terceira linha:** dois `\t` inserem duas tabulações antes da palavra "Simples".

Como resultado, temos a saída a seguir:



```
Saída na tela
Um
    texto
        Simples
```

Figura 4

Especificadores de formato com o comando

Eles indicam o tipo de dado que será exibido. A tabela 3 ilustra os mais comuns na linguagem C para saída com printf:

Tabela 3 – Especificadores de formato no printf

Especificador	Tipo de dado	Descrição	Exemplo de uso	Saída
%d ou %i	Inteiro (int)	Exibe um número inteiro com sinal (base 10)	<code>printf("%d", 42);</code>	42
%f	Número real (float ou double)	Exibe um número real em formato decimal	<code>printf("%f", 3.14);</code>	3.140000
%c	Caractere (char)	Exibe um único caractere	<code>printf("%c", 'A');</code>	A
%s	string (char[])	Exibe uma cadeia de caracteres (string)	<code>printf("%s", "Aula");</code>	Aula
%o	Inteiro (int)	Exibe o valor em formato octal (base 8)	<code>printf("%o", 64);</code>	100
%x	Inteiro (int)	Exibe o valor em formato hexadecimal (minúsculas)	<code>printf("%x", 255);</code>	ff
%X	Inteiro (int)	Exibe o valor em formato hexadecimal (maiúsculas)	<code>printf("%X", 255);</code>	FF
%p	Variável ou ponteiro	Exibe o endereço de memória de uma variável ou um ponteiro	<code>printf("%p", &a);</code>	0x7fff..
%%	Literal	Exibe o caractere %	<code>printf("%%");</code>	%

O operador & retorna o endereço da variável x. O %p imprime o endereço em formato hexadecimal. Tanto variáveis normais quanto ponteiros têm endereços que podem ser exibidos.

No uso do printf, usamos %f para indicar a impressão de uma variável do tipo float ou double, pois ambos são tratados como double quando passados para funções com número variável de argumentos, como o printf.

Os especificadores de formato tornam o comando printf altamente versátil para diferentes tipos de dados e formatações em C. Há outros especificadores além dos mais comuns mencionados; para uma lista completa e atualizada, consulte a documentação oficial da linguagem C ou um manual confiável.

Com o uso dos especificadores de formato, é possível imprimir variáveis na saída padrão. No exemplo apresentado a seguir, o programa imprime o valor do tipo inteiro armazenado na variável num1. Aqui, utilizamos o especificador %d para indicar que a variável é do tipo inteiro.


```
#include <stdio.h>

int main() {
    int num1 = 10;

    printf("%d", num1);

    return 0;
}
```

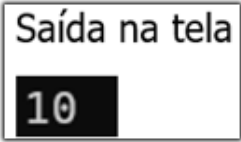
A screenshot of a terminal window with a white background and a black border. The title bar at the top says "Saída na tela". Inside the window, the number "10" is displayed in white text on a black rectangular background.

Figura 5

O comando `printf` é uma função poderosa da linguagem C usada para exibir informações na tela. Ele permite a formatação de texto e a apresentação de variáveis de diferentes tipos, como (inteiros (`int`), reais (`float`), caracteres (`char`), entre outros. Por exemplo, ao utilizar `printf("A soma de %d e %d é %d", a, b, a+b);`, sendo `a` e `b` variáveis do tipo inteiro com valores previamente definidos, o programa exibirá o resultado da soma de forma clara e organizada. A operação pode ser armazenada em uma variável auxiliar antes de ser impressa, garantindo que o resultado seja reutilizado sem a necessidade de refazer o cálculo. Exemplo:

```
int a = 5, b = 10, soma = 0; // Declaração e inicialização das variáveis
soma = a + b; // Soma de a e b armazenada na variável soma
printf("A soma de %d e %d é %d\n", a, b, soma); // Exibição do resultado
```

A variável é inicializada com 5, `b` com 10 e `soma` com 0. Após a operação da segunda linha, `soma = a + b`, a variável `soma` é modificada e passa valer 15, e este valor será impresso no local da variável `soma`, no `printf` indicado pelo `%d` correspondente à variável `soma`. O `printf(-)` exibe o resultado, substituindo `%d` pelos valores das variáveis. Saída esperada no console: a soma de 5 e 10 é 15.

O comando `printf` é fundamental para comunicar os resultados de um programa e facilitar a interação com o usuário.

O código apresentado a seguir mostra como formatar a saída de um número com quantidade mínima de caracteres. O uso de largura mínima com especificadores (`%nd`), onde `n` indica a quantidade de caracteres, é útil para alinhar saídas em tabelas ou valores.

Se o número tiver mais dígitos do que o especificado (por exemplo, 12345 com %3d), ele será impresso normalmente sem cortes ou truncamentos.

```
#include <stdio.h>

int main() {
    int num1 = 10;
    printf("%d\n", num1);
    printf("%3d\n", num1);
    printf("%5d", num1);

    return 0;
}
```

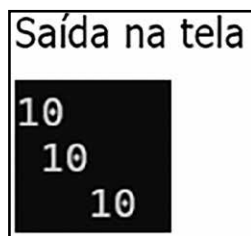
A imagem mostra a saída de um programa em C no terminal. O título "Saída na tela" está no topo. Abaixo dele, há um fundo preto com o número "10" branco impresso três vezes, cada uma em uma linha separada por uma quebra de linha. Isso demonstra o efeito dos especificadores de formato padrão, %3d e %5d, que adicionam espaços em branco à esquerda do número para atingir uma largura mínima de 3 e 5 caracteres, respectivamente.

Figura 6

Analisando a saída do código, temos:

- **%d\n**: o especificador %d é usado para imprimir um número inteiro na sua forma padrão. O \n provoca uma quebra de linha após a impressão. Saída: 10 seguido de uma nova linha.
- **%3d\n**: o especificador %3d define que o número inteiro deve ser exibido com, no mínimo, três caracteres de largura. Se o número tiver menos dígitos, espaços em branco serão adicionados à esquerda para completar a largura. Neste caso, o número 10 ocupa apenas dois caracteres, então um espaço em branco será adicionado antes dele.
- **%5d**: o especificador %5d funciona da mesma maneira que %3d, mas define uma largura mínima de cinco caracteres. Como o número 10 ocupa apenas dois caracteres, três espaços em branco serão adicionados antes dele.

Caractere de barra invertida

Em C, certos caracteres especiais não podem ser inseridos diretamente pelo teclado, como o retorno de carro (CR) ou o alerta sonoro. Para representá-los e aumentar a portabilidade, a linguagem C utiliza sequências de escape, que são constantes de barra invertida (\) seguidas por um código.

Essas constantes de barra invertida são amplamente usadas para:

- Representar caracteres invisíveis, como `\n` (nova linha) e `\t` (tabulação).
- Inserir caracteres que têm significado especial, como aspas duplas (`\"`) ou aspas simples (`\'`), em strings.
- Trabalhar com códigos octais e hexadecimais para representar valores na tabela ASCII.

Quadro 1 – Caractere de barra invertida

Código	Significado	Descrição
<code>\a</code>	Alerta (beep)	Emita um som de alerta no terminal (se suportado pelo dispositivo)
<code>\b</code>	Retrocesso (BS)	Move o cursor uma posição para trás
<code>\f</code>	Alimentação de formulário (FF)	Passa para a próxima página em dispositivos de impressão
<code>\n</code>	Nova linha (LF)	Move o cursor para o início da próxima linha
<code>\r</code>	Retorno de carro (CR)	Move o cursor para o início da linha atual, sem avançar para a próxima
<code>\t</code>	Tabulação horizontal (HT)	Insere um espaço de tabulação horizontal
<code>\"</code>	Aspas duplas	Permite usar aspas duplas dentro de strings
<code>\'</code>	Aspas simples	Permite usar aspas simples em strings ou caracteres
<code>\0</code>	Nulo	Marca o final de uma string
<code>\\</code>	Barra invertida	Insere uma barra invertida literal (<code>\</code>)
<code>\v</code>	Tabulação vertical	Move o cursor para a próxima linha vertical alinhada
<code>\N</code>	Constante octal	Representa um caractere pelo seu valor octal na tabela ASCII (exemplo: <code>\101</code>)
<code>\xN</code>	Constante hexadecimal	Representa um caractere pelo seu valor hexadecimal (exemplo: <code>\x41</code>)

Vamos analisar mais alguns exemplos para fixar melhor o conteúdo. No exemplo a seguir, temos alguns novos elementos, como a inclusão do arquivo de cabeçalho `locale.h`, para permitir a configuração de localização para que o compilador suporte os caracteres com acentuação.

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

int main() {
    setlocale(LC_ALL, "Portuguese");

    printf("Número\tQuadrado\n");
    printf("=====\t=====\n");

    printf("%4d\t%8d\n", 1, 1 * 1);
    printf("%4d\t%8d\n", 2, 2 * 2);
    printf("%4d\t%8d\n", 3, 3 * 3);
    printf("%4d\t%8d\n", 4, 4 * 4);
    printf("%4d\t%8d\n", 5, 5 * 5);

    return 0;
}
```

Analisando o código, temos inicialmente a inclusão dos arquivos de cabeçalho:

- **#include <stdio.h>**: permite usar funções como `printf()` para entrada e saída de dados.
- **#include <locale.h>**: necessária para definir configurações regionais, como o suporte a caracteres acentuados (á, é, ç, etc.) e símbolos localizados.

Podemos ver no código um exemplo de programa em C que exibe uma tabela de números e seus quadrados. Cada número e seu respectivo quadrado são exibidos em linhas separadas. Como resultado, temos uma tabela bem definida e organizada.

Saída na tela	
Número	Quadrado
====	=====
1	1
2	4
3	9
4	16
5	25

Figura 7

No exemplo a seguir, mostramos o uso de mais alguns caracteres de barra invertida. Esse código indica como usar sequências de escape para manipular a exibição de texto e caracteres especiais no console, um recurso essencial para formatação e controle de saída em C.

```
#include <stdio.h>

int main() {
    printf("Linha 1\n");           // Nova linha
    printf("\tLinha com tabulação\n"); // Tabulação horizontal
    printf("Texto com \"aspas\"\n"); // Uso de aspas duplas
    printf("Caractere: %c\n", '\n'); // Representação do caractere \n
    printf("Alerta sonoro: \a\n"); // Emite um beep, se suportado

    return 0;
}
```

Saída na tela

```
Linha 1
      Linha com tabulaçãOo
Texto com "aspas"
Caractere:
Alerta sonoro:
```

Figura 8

Analisando o código, temos:

- **Linha 1\n**: exibe o texto "Linha 1" seguido por uma nova linha (\n), movendo o cursor para a próxima linha.
- **\tLinha com tabulação\n**: adiciona uma tabulação horizontal (\t) antes de "Linha com tabulação" e, em seguida, insere uma nova linha (\n).
- **Texto com \"aspas\"\n**: exibe o texto com aspas duplas (") inclusas, seguido por uma nova linha.
- **Caractere: %c\n**: exibe o texto "Caractere:" e o caractere especial \n, que representa uma nova linha.
- **Alerta sonoro: \a\n**: exibe "Alerta sonoro:" e emite um beep (\a), se suportado, seguido por uma nova linha.

No final, o programa exibe o texto formatado no console, incluindo caracteres especiais como nova linha (\n), tabulação horizontal (\t), aspas duplas (") e alerta sonoro (\a), que pode emitir um som audível, se suportado pelo sistema.

O erro na palavra "tabulação", que aparece corrompida na saída ("tabulaçãOo"), ocorre devido a problemas de codificação de caracteres (como UTF-8 vs. ANSI) ou à ausência de configuração regional

adequada. Em alguns casos, isso pode ser corrigido incluindo o cabeçalho `<locale.h>` e configurando a localidade com o comando `setlocale(LC_ALL, "");`, conforme ilustrado anteriormente.

O `printf` é uma função de saída usada para exibir informações na tela. Ele utiliza especificadores de formato (como `%d`, `%f`, `%s`) para indicar o tipo de dado que será exibido. Sequências de escape (como `\n`, `\t`, `\\`) permitem inserir caracteres especiais ou invisíveis, como quebras de linha ou tabulações.

O uso correto de strings e sequências de escape torna o código mais limpo, legível e portátil, funcionando adequadamente em diferentes sistemas.

Em C, não existe um tipo de dado nativo chamado string, como em outras linguagens (Python, Java). No entanto, o termo "string" é amplamente usado para se referir a uma sequência de caracteres terminada por `'\0'` (caractere nulo).

1.1.4 Comando `scanf()`

É uma função definida no arquivo de cabeçalho padrão `<stdio.h>` usada para ler dados de entrada, geralmente fornecidos pelo usuário através do teclado. Ou seja, permite que o programa leia informações digitadas pelo usuário. Esses dados são interpretados e atribuídos a variáveis específicas, permitindo a interação entre o usuário e o sistema.

Por meio de `scanf()`, é possível capturar dados de diferentes tipos, como inteiros, números de ponto flutuante, caracteres e strings que serão armazenados em variáveis durante a execução do programa.

Neste livro-texto e em diversas outras obras da literatura sobre a linguagem C, é possível encontrar a escrita de comandos sem o uso de parênteses ou outros símbolos junto ao nome da função. Isso ocorre porque, nesses casos, trata-se apenas de uma explicação textual. No entanto, dentro do **código-fonte** de um programa, a forma correta de utilizar `scanf` é como função, ou seja, sempre deve ser escrita com parênteses: `scanf()`. Sem eles, o compilador gerará um erro, pois não reconhecerá a chamada como uma função. O mesmo se aplica ao `printf` e a qualquer outra função em C.

Sintaxe da função

- `scanf("formato", &variavel)`
- **Formato:** é uma string (chamada de formatador) que define o tipo de dado a ser lido pela função `scanf`. Veja alguns exemplos comuns:
 - `%d`: para leitura de inteiros (int).
 - `%f`: para leitura de números de ponto flutuante do tipo (float).
 - `%lf`: para leitura de números de ponto flutuante do tipo (double).

- **%c**: para leitura de um caractere (char).
- **%s**: para a leitura de uma string, ou seja, um vetor (array) de caracteres do tipo char.
- O identificador de nome variavel representa o nome da variável previamente declarada no programa.



Observação

O acento na palavra foi omitido intencionalmente, pois nomes de variáveis em C não podem conter acentos nem caracteres especiais.

No scanf, os especificadores %f e %lf não são equivalentes. No scanf() usamos %f para ler variáveis do tipo float e %lf para ler variáveis do tipo double.

- **&variavel**: o operador & é usado para passar o endereço de memória da variável à função scanf. Isso permite que a função armazene o valor lido diretamente nessa variável.
- **Leitura de caracteres (%c)**: sempre adicione um **espaço** antes de %c no formato para evitar capturar o caractere de nova linha (\n) deixado no buffer por comandos anteriores. Exemplo: `scanf(" %c", &letra);`

Por que usamos o operador & no scanf ?

O operador & é necessário para a maioria das variáveis, pois o valor lido é armazenado diretamente no endereço de memória. Ou seja, usamos o operador & para fornecer o endereço de memória da variável ao scanf. Isso é necessário porque o scanf precisa modificar o valor da variável diretamente, o que só é possível acessando seu endereço.

- **Strings (vetores de char)**: no caso da leitura de uma string com o especificador %s, não usamos o operador &. Isso acontece porque o nome do vetor já representa o seu endereço de memória. Trata-se de uma exceção.
- **Uso de colchetes [] no scanf() em C**: está relacionado a um recurso chamado scanset (conjunto de caracteres). Ele permite ler uma sequência de caracteres até que um caractere não especificado apareça.

Podemos definir um tamanho máximo para a leitura de uma string com scanf, garantindo que não ultrapasse o limite do array, evitando o estouro de buffer. A sintaxe é `scanf("%x[^\n]", nome);`, onde x é o número máximo de caracteres a serem lidos, e `[^\n]` é um scanset que instrui o scanf a ler até a quebra de linha, sem incluí-la. Exemplo:

```
char nome[50]; // Suporta até 49 caracteres + '\0'

printf("Digite seu nome completo: ");
scanf("%49[^\n]", nome);
```

Explicação

O número 49 em `%49[^\n]` define que serão lidos no máximo 49 caracteres, reservando um espaço para o caractere nulo `–('\0')`.

O `scanf` lerá todos os caracteres digitados, exceto `'\n'`, ou seja, até a quebra de linha.

O espaço antes de `%` (em `" %49[^\n]"`) serve para ignorar espaços em branco ou quebras de linha anteriores, evitando problemas com o Enter deixado por leituras anteriores.

O uso de `scanf` exige cuidado com os tipos de dados e o controle do buffer, temas que serão estudados depois.

O uso do `scanf()` é crucial para criar programas interativos que respondem aos dados fornecidos pelos usuários.

Temos também a possibilidade de ler textos com espaços entre as palavras com outras funções como: `fgets()`.

Exemplo de `fgets()` para leitura de textos (strings) com espaços:

```
char frase[50];

printf("Digite uma frase: ");
fgets(frase, 50, stdin); // Lê uma linha inteira

printf("Frase digitada: %s", frase);
```

O `fgets()` permite ler espaços e vários caracteres. Além disso, é mais seguro, pois garante que a string não ultrapasse o tamanho do vetor.

Resumo:

- Para ler strings com espaços, use `%[^\n]` ou `fgets()`.
- Nunca use `scanf("%s", nome)` sem um limite, pois pode causar falhas de segurança! `scanf("%49[^\n]", nome)` usa um limite de tamanho seguro na entrada.

- A função `fgets(nome,sizeof(nome),stdin)`; é uma alternativa mais segura para evitar estouro de buffer. O operador `sizeof(nome)` retorna o tamanho total da variável, que, neste caso, é o array `nome`. Quando usado na função `fgets()`, ele define automaticamente o tamanho máximo de caracteres que podem ser lidos, garantindo que a entrada não ultrapasse o limite do array e evitando problemas de segurança.
- Sempre use o operador `&` antes do nome da variável, exceto para strings (arrays de caracteres), pois o nome do array já representa o endereço de memória.
- O `scanf()` pode ser problemático se a entrada do usuário não corresponder ao formatador especificado. Por isso, em programas robustos, é comum usar funções como `fgets()` para leitura de strings e depois converter para o tipo desejado.

Erro comum:

- **Esquecer o operador `&`:** um dos erros mais comuns ao usar a função `scanf` em C é esquecer o operador `&`, o que faz com que o programa não saiba onde armazenar o valor lido. Isso ocorre porque `scanf` espera um ponteiro para a região de memória onde o valor deve ser gravado.
 - **Exceção:** no caso da leitura de strings (vetores de char), não é necessário usar o operador, pois o nome do vetor já representa o endereço de memória do primeiro elemento.

Observemos a seguir a explicação técnica:

- Em programas simples, que utilizam variáveis básicas como `int`, `float`, ou `double`, geralmente declaradas no mesmo escopo, é necessário usar o `&`.
- Já em programas mais complexos, comuns em contextos profissionais, é comum trabalhar com alocação dinâmica de memória e com variáveis declaradas como ponteiros. Nesses casos, o uso do `&` pode ser:
 - Desnecessário, se o ponteiro já aponta para o local correto.
 - Útil para outras finalidades, como ter acesso a membros de estruturas ou manipulação de endereços.
- **Escrever `scanf` sem parênteses:** `scanf` sem os parênteses resultará em erro de compilação, pois o compilador não reconhecerá a chamada como uma função.
- **Problemas com buffer:** ao usar `scanf` após leitura de strings ou números, o caractere de nova linha (`\n`) deixado no buffer pode interferir em comandos subsequentes.

1.1.5 Exemplos do uso de printf() e scanf()

No exemplo a seguir, temos um programa em linguagem C que faz a leitura de entrada de dados do usuário utilizando a função `scanf()`, permitindo que o usuário forneça três tipos diferentes de dados: um número inteiro (`int`), um número de ponto flutuante (`float`) e um caractere (`char`). A saída no console exibe as informações fornecidas pelo usuário de forma organizada.

```
#include <stdio.h>

int main() {
    int idade;
    float altura;
    char inicial;

    printf("Digite sua idade: ");
    scanf("%d", &idade); // Lê um número inteiro

    printf("Digite sua altura: ");
    scanf("%f", &altura); // Lê um número de ponto flutuante

    printf("Digite a inicial do seu nome: ");
    scanf(" %c", &inicial); // Lê um caractere (observe o espaço antes do %c)

    printf("\nIdade: %d, Altura: %.2f, Inicial: %c\n", idade, altura, inicial);
    return 0;
}
```

Entrada de dados
Digite sua idade: 30
Digite sua altura: 1.75
Digite a inicial do seu nome: F
Saída na tela
Idade: 30, Altura: 1.75, Inicial: F

Figura 9

Analisando a estrutura do código, temos os itens acentuados a seguir.

Início do programa

<stdio.h>. Inicialmente, o arquivo de cabeçalho `stdio.h` é incluído para permitir o uso das funções de entrada e saída padrão, como `printf()` e `scanf()`.

int main() {}. O programa começa sua execução pela função `main()`, que é obrigatória em programas que geram executáveis, o abre-chaves `{` indica o início do bloco principal do programa, este deverá ser fechado no final do código com um fecha-chaves `}`.

Três variáveis são declaradas:

- **idade**: do tipo `int`, para armazenar um número inteiro.
- **altura**: do tipo `float`, para armazenar um número com casas decimais.
- **inicial**: do tipo `char`, para armazenar um único caractere.

Entrada de dados

Haverá a leitura da idade (número inteiro).

```
int idade; //declaração da variável idade

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

O `printf()` exibe a mensagem "Digite sua idade: ".

Depois, o `scanf("%d", &idade);` recebe um número inteiro digitado pelo usuário e o armazena na variável `idade`.

Por sua vez, o operador `&` é necessário para passar o endereço de memória da variável `idade`, permitindo que a função `scanf()` armazene o valor corretamente.

Leitura da altura (número real)

```
float altura;

printf("Digite sua altura: ");
scanf("%f", &altura); // Lê um número de ponto flutuante
```

O usuário insere um valor decimal (`float`), armazenado na variável `altura`. O formato `%f` indica que a entrada será interpretada como um número de ponto flutuante.

Leitura da inicial do nome (caractere)

```
char inicial;

printf("Digite a inicial do seu nome: ");
scanf(" %c", &inicial);
```

`%c` é utilizado para capturar um único caractere. O espaço antes de `%c` ("`%c`") é necessário para ignorar o caractere de nova linha (`\n`) deixado no buffer pelo `scanf()` anterior.

Saída dos dados na tela

Haverá a exibição dos dados.

```
printf("\nIdade: %d", idade);  
printf("\nAltura: %.2f", altura);  
printf("\nInicial: %c", inicial);
```

O `printf()` exibe os valores fornecidos pelo usuário: o `\n` no início do comando indica uma quebra de linha para depois exibir os dados.

`%d` exibe a idade como número inteiro.

`%.2f` exibe a altura com duas casas decimais. O `.2` (ponto dois) indica a quantidade de casas decimais que devem ser impressas. O uso de apenas `%f`, sem a indicação de quantas casas decimais devem ser impressas, resulta na impressão de todas as seis casas decimais do tipo `float`.

`%c` exibe a inicial como um caractere.

Resumo do programa:

- O programa solicita e recebe três tipos de dados do usuário: inteiro (`int`), ponto flutuante (`float`) e caractere (`char`).
- O programa utiliza `scanf()` para capturar os valores e `printf()` para exibi-los no console, ou, simplificando, exibe os dados na tela.

No exemplo a seguir, temos um programa que armazena informações de um livro, como o título, o número de páginas e o ano de publicação.

Explicação resumida do código:

- `char titulo[100];` declara uma string para armazenar o título do livro.
- `scanf("%99[^\n]", titulo);` lê uma linha de texto (título) até encontrar o caractere de nova linha (`\n`). O especificador `%99[^\n]` garante que não haja estouro de buffer, limitando a leitura a 99 caracteres. Esse limite é importante porque deixa espaço para o caractere nulo (`\0`), que é adicionado automaticamente ao final da string para indicar o término dela.
- `scanf("%d", &paginas);` lê um número inteiro (número de páginas).
- `scanf("%d", &ano);` lê outro número inteiro (ano de publicação).
- Limitar o número de caracteres lidos é uma prática essencial para evitar vulnerabilidades, como estouro de buffer.

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

int main() {
    setlocale(LC_ALL, "Portuguese");

    char titulo[100];
    int paginas;
    int ano;

    printf("Digite o título do livro: ");
    scanf("%99[^\n]", titulo); // Lê uma string até encontrar uma nova linha

    printf("Digite o número de páginas: ");
    scanf("%d", &paginas); // Lê um número inteiro

    printf("Digite o ano de publicação: ");
    scanf("%d", &ano); // Lê outro número inteiro

    printf("\nLivro cadastrado:\n");
    printf("Título: %s\n", titulo);
    printf("Páginas: %d\n", paginas);
    printf("Ano: %d\n", ano);

    return 0;
}
```

Entrada de dados
Digite o título do livro: Aprendendo C Digite o número de páginas: 200 Digite o ano de publicação: 2025
Saída na tela
Livro cadastrado: Título: Aprendendo C Páginas: 200 Ano: 2025

Figura 10

A saída do código apresentado é o resultado de um programa que solicita ao usuário informações sobre um livro e, em seguida, exibe essas informações. Vamos analisar passo a passo:

- **Entrada do usuário**

- O programa solicita que o usuário insira o título do livro, o número de páginas e o ano de publicação.
- O usuário fornece as seguintes informações:
 - **Título:** Aprendendo C.

- Número de páginas: 200.
- Ano de publicação: 2025.

- **Processamento**

- O programa armazena essas informações em variáveis apropriadas (por exemplo, uma string para o título e inteiros para o número de páginas e o ano).

- **Saída**

- O programa exibe as informações fornecidas pelo usuário:
 - **Título:** Aprendendo C.
 - Páginas: 200.
 - Ano: 2025.

1.1.6 Identificadores e variáveis

Em C, os nomes atribuídos a variáveis, funções e outros elementos criados pelo programador são chamados de identificadores. Para que sejam válidos, esses nomes devem começar obrigatoriamente por uma letra ou pelo caractere de sublinhado (`_`), e os demais caracteres podem incluir letras, números ou sublinhados.

1.1.6.1 Nomes de identificadores

Na linguagem C, as letras maiúsculas e minúsculas são tratadas como caracteres distintos, o que significa que os identificadores `count`, `Count` e `COUNT` são diferentes entre si.

Por fim, um identificador não pode ter o mesmo nome que uma palavra-chave reservada da linguagem C, como `int`, `return`, `if`, entre outras, nem deve coincidir com os nomes das funções presentes na biblioteca padrão ou que você mesmo definiu no código.

Esses identificadores seguem algumas regras importantes para sua definição:

- O primeiro caractere deve ser uma letra (`a-z` ou `A-Z`) ou um sublinhado (`_`).
- Os caracteres subsequentes podem ser letras, números (`0-9`) ou sublinhados.
- Um identificador não pode conter espaços, caracteres especiais (como `@`, `#`, `!`) ou começar com números.
- Identificadores diferenciam letras maiúsculas e minúsculas (case-sensitive).

Exemplos de identificadores válidos:

```
int contador;      // Começa com uma letra
float _media;      // Começa com um sublinhado
char nomeCliente;  // Usa letras e maiúsculas
int teste123;      // Contém números após o início
```

Exemplos de identificadores inválidos:

```
int 1numero;       // Inválido: começa com número
float valor$;       // Inválido: contém caractere especial '$'
char nome cliente; // Inválido: contém espaço
```

1.1.6.2 Variáveis

Uma variável é uma posição nomeada de memória, que é usada para guardar um valor que pode ser modificado pelo programa. Em C, todas as variáveis precisam ser declaradas antes de serem utilizadas. De forma simplificada, podemos compará-las a uma "caixa" destinada a conter um tipo específico de dado, sendo que o tipo de dado define as características e o conteúdo que essa caixa pode armazenar.

A estrutura básica de uma declaração em C segue o formato `tipo lista_de_variáveis ;`

Nesse modelo, o tipo representa qualquer tipo de dado válido da linguagem, podendo incluir modificadores como `unsigned` ou `long`. Já `lista_de_variáveis` corresponde a um ou mais identificadores, separados por vírgulas, que indicarão os nomes das variáveis a serem criadas. Veja alguns exemplos a seguir:

```
int i, j, k, l;
short int si;
double salario, precoProduto;
```

Onde as variáveis são declaradas?

As variáveis estão associadas a um escopo, podendo ser declaradas dentro desse escopo e tendo sua visibilidade limitada a ele.

Por padrão, elas podem ser declaradas em três lugares básicos:

- **Dentro de funções:** estas são variáveis locais.
- **Na definição dos parâmetros das funções:** parâmetros formais.
- **Fora de todas as funções:** variáveis globais, respectivamente.

Delimitadores de escopo em C

Na linguagem C, os delimitadores de escopo são usados para indicar onde um bloco de código começa e termina. Eles definem o contexto em que variáveis, funções e instruções são visíveis e executadas.

As chaves {} são os principais delimitadores de escopo em C. Tudo que está entre { e } pertence a um mesmo bloco de código.

Importância do escopo:

- Variáveis locais só existem dentro do escopo em que foram declaradas.
- Funções e estruturas de controle (if, while, for, etc.) usam blocos {} para delimitar ações.
- Evita conflitos entre nomes de variáveis.
- Permite estruturação e organização clara do código.



Observação

Declarar uma variável dentro de {} a torna inacessível fora desse bloco.
É possível criar escopos aninhados (blocos dentro de blocos).

Inicialização de variáveis

Em C, é possível atribuir um valor às variáveis no momento em que elas são declaradas. Para isso, basta utilizar o operador de atribuição = seguido de uma constante após o nome da variável. A forma geral para inicializar uma variável é:

```
tipo nome_da_variável = constante(valor inicial);
```

Observe os exemplos de inicialização de variáveis:

```
char ch = 'a';           // Variável do tipo caractere
int first = 0;           // Variável inteira inicializada com zero
float balance = 123.23; // Variável de ponto flutuante com valor decimal
```

Variáveis locais e globais em C

Na linguagem C, as variáveis podem ser de dois tipos, locais ou globais, dependendo de onde são declaradas no código. A principal diferença entre elas está no escopo (onde podem ser acessadas) e na duração (quanto tempo permanecem na memória).

As **variáveis locais** são aquelas declaradas dentro de funções ou blocos de código, delimitados por chaves {}. Seu escopo é restrito ao bloco onde foram definidas, o que significa que só podem ser acessadas dentro desse mesmo trecho de código.

Essas variáveis são criadas assim que o bloco começa a ser executado e destruídas ao final do bloco, deixando de ocupar espaço na memória. Portanto, elas existem apenas durante a execução do bloco no qual estão inseridas.

Internamente, variáveis locais são armazenadas em uma região da memória chamada pilha (stack). Essa área é usada para armazenar dados temporários durante a execução de funções.

Observe a seguir suas características:

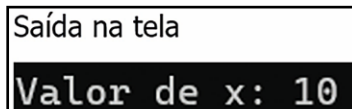
- **Escopo:** só podem ser acessadas dentro da função ou bloco onde foram declaradas.
- **Duração:** são criadas quando a função/bloco é executada e destruída quando a execução termina.
- **Memória:** usam a pilha (stack), garantindo que cada função tenha suas próprias variáveis locais.

No exemplo a seguir, a variável `x` só existe dentro da função `exemplo()` e não pode ser acessada fora dela. O programa define uma função chamada `exemplo()` que utiliza uma variável local chamada `x`. Essa variável é inicializada com o valor 10 e exibida no console usando `printf`. Para tal, é necessário fazer a chamada da função `exemplo()` que ocorre dentro da função `main()`; após sua execução, o programa termina.

```
#include <stdio.h>

void exemplo() {
    int x = 10; // Variável local
    printf("Valor de x: %d\n", x);
}

int main() {
    exemplo();
    // printf("%d", x); // Erro: 'x' não é acessível aqui
    return 0;
}
```



```
Saída na tela
Valor de x: 10
```

Figura 11

```
#include <stdio.h>

void exemplo() {
    int x = 10; // Variável local
    printf("Valor de x: %d\n", x);
}

int main() {
    exemplo();
    printf("Valor de x dentro do main %d", x);
    return 0;
}
```

No exemplo anterior, ao executar o programa ocorrerá um erro de compilação, pois a variável `x`, utilizada no `printf` dentro da função `main()`, não foi declarada nesse escopo. A variável `x` foi declarada apenas dentro da função `exemplo()`, por isso não está acessível em `main`, já que sua visibilidade está restrita ao bloco em que foi criada.

As **variáveis globais** são aquelas declaradas fora de qualquer função, geralmente no início do programa, antes da função `main()`.

Observe a seguir as características das variáveis globais:

- **Escopo:** podem ser acessadas e modificadas por qualquer função dentro do programa.
- **Duração:** existem durante toda a execução do programa, sendo criadas no início e destruídas ao final.
- **Memória:** usam a área global ou área de dados na memória.

Exemplo:

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

int global = 100; // Variável global

void exemplo() {
    printf("Valor da variável global: %d\n", global);
}

int main() {
    setlocale(LC_ALL, "Portuguese");

    exemplo();
    global = 200; // Modificando o valor da variável global
    printf("Novo valor da variável global: %d\n", global);
    return 0;
}
```

```
Saída na tela
Valor da variável global: 100
Novo valor da variável global: 200
```

Figura 12

No exemplo, a variável global pode ser acessada e modificada tanto pela função `exemplo()` quanto pelo `main()`. Ou seja, uma variável global pode ser acessada por todas as funções do programa. Analisando o código do programa acentuado, temos: a declaração de uma variável global chamada `global` com o valor inicial 100, bem como a definição de uma função chamada `exemplo()` que exibe o valor atual da variável global. No `main`, configuramos a localidade para Português com a função `setlocale()` e:

- Chama a função `exemplo()` para mostrar o valor inicial da variável global.
- Modifica o valor da variável global para 200.
- Exibe o novo valor da variável global.

Quadro 2 – Comparação entre variáveis locais e variáveis globais em C

Aspecto	Variáveis locais	Variáveis globais
Localização da declaração	Declaradas dentro de uma função ou bloco de código ({}).	Declaradas fora de qualquer função, no início do programa.
Escopo	Somente acessíveis dentro da função ou bloco onde foram declaradas.	Acessíveis por todas as funções do programa.
Tempo de vida	Criadas quando a função/bloco é chamada e destruída ao final da execução.	Existentes durante toda a execução do programa.
Memória	Armazenadas na pilha (stack).	Armazenadas na área global (data segment).
Compartilhamento de dados	Não podem ser compartilhadas entre funções.	Compartilhadas por todas as funções do programa.
Conflito de nomes	Duas ou mais funções diferentes podem ter variáveis com os mesmos nomes.	Cuidado ao utilizar variável global com o mesmo nome de uma variável local no mesmo programa.
Inicialização	Não são automaticamente inicializadas (devem ser atribuídas antes do uso).	São automaticamente inicializadas (exemplo: 0 para números).
Performance	Mais rápidas, pois usam a pilha, que é gerenciada eficientemente.	Podem ser mais lentas, pois ficam em memória global.
Boas práticas	Preferidas para evitar efeitos colaterais e facilitar o rastreamento.	Devem ser usadas com cautela, pois podem causar conflitos.



Observação

O mesmo nome até pode ser utilizado, mas o valor vai ser o declarado no escopo "mais interno". Ou seja, no caso de um conflito entre uma variável local e uma global com o mesmo nome, a linguagem usa o escopo local. Mesmo que isso não seja uma boa prática de programação, pois pode confundir a leitura e a interpretação do programa, um programa que contenha uma variável global e outra local com o mesmo nome não é considerado, e ele compila e executa "normalmente".

1.1.7 Boas práticas em programação C

Adotar boas práticas de programação é essencial para escrever códigos claros, organizados e fáceis de manter. Em C, algumas boas práticas incluem:

- **Comentário do código:** adicione comentários que expliquem partes importantes do código para torná-lo compreensível para outras pessoas (ou para você mesmo no futuro).
- **Nomes de variáveis significativos:** use nomes de variáveis que descrevam seu propósito, em vez de nomes genéricos como x ou y.
- **Indentação e formatação consistente:** mantenha o código bem indentado e formatado para facilitar a leitura e a manutenção.
- **Uso de constantes e macros:** defina valores constantes usando #define ou const para facilitar futuras alterações e tornar o código mais legível.
- **Evitar o uso excessivo de variáveis globais:** prefira o uso de variáveis locais para evitar conflitos e manter o escopo do código mais claro. Essas práticas ajudam a evitar erros, simplificam a identificação de problemas e tornam o programa mais compreensível e profissional.



Saiba mais

Para se aprofundar e estudar mais, consulte os livros da bibliografia e compreenda a estrutura padrão dos programas C.

Pratique no compilador online para criar seus próprios exemplos com printf() e scanf().

Disponível em: <https://bit.ly/4kFNYbp>. Acesso em: 25 jun. 2025.

Explore a documentação oficial da biblioteca <stdio.h> para ver todos os recursos de entrada e saída disponíveis.

Disponível em: <https://shre.ink/xKdb>. Acesso em: 25 jun. 2025.

Boas práticas com identificadores

Use nomes significativos para identificar o propósito da variável ou função:

```
int idade;           // Fácil de entender
float salarioBase; // Nome autoexplicativo
```

Evite nomes muito longos ou confusos, mas priorize clareza:

```
int x; //nome vago, o ideal é usar nome significativo
int nomeAlunosDaTurmaManhaColegio; // nome exagerado
```

Prefira usar o esquema a seguir:

```
int totalAlunos; // Claro e objetivo
float mediaAluno;
```

Use convenções de nomenclatura consistentes, como:

- **Camel Case:** é uma convenção de nomenclatura que consiste em escrever nomes de variáveis e funções unindo as palavras em uma única palavra, e a primeira letra de cada palavra (exceto a primeira) é maiúscula.

```
float mediaSalarial, totalVendas;
```

- **Snake Case:** neste padrão, as palavras em uma string são separadas por underscores e todas as letras são minúsculas. Em C, a convenção de nomenclatura Snake Case é frequentemente usada para variáveis e funções.

```
float media_salarial, total_vendas;
```

Esses exemplos ilustram como definir identificadores corretamente, respeitando as regras da linguagem C e mantendo o código mais legível e organizado.

Boas práticas no uso de variáveis

Prefira variáveis locais:

- Elas são mais seguras e evitam alterações acidentais de valores por funções não relacionadas.
- Tornam o programa mais modular e fácil de depurar.

Use variáveis globais com cuidado:

- Só utilize globais se realmente precisar compartilhar dados entre várias funções.
- Identifique-as claramente e limite seu uso para evitar conflitos de valor.

Use nomes descritivos:

- Para globais, prefira nomes que indiquem claramente sua função no programa.

Acentuaremos a seguir 10 dicas de boas práticas gerais em C.

Nomes significativos para variáveis e funções

- Use nomes descritivos que indiquem a finalidade da variável ou função.
- Evite nomes genéricos como x, y, n, ou temp.

Exemplo:

```
int numero_de_paginas; // Bom
int n;                 // Ruim
```

Comentários claros e úteis:

- Comente o código para explicar a lógica complexa ou decisões importantes.
- Evite comentários óbvios ou redundantes.

Exemplo:

```
float area = PI * raio * raio; //Calcula a área do círculo ( $\pi * r^2$ )
```

Evite código repetido:

- Use funções para encapsular trechos de código que são repetidos.
- Isso facilita a manutenção e reduz a chance de erros.

Exemplo:

```
void imprimirDadosLivro(char *titulo, int paginas, int ano) {
    printf("Título: %s\n", titulo);
    printf("Páginas: %d\n", paginas);
    printf("Ano: %d\n", ano);
}
```

Validação de entradas

- Sempre valide entradas do usuário para evitar comportamentos inesperados.

Exemplo:

```
if (scanf("%d",&paginas) != 1) {  
    printf("Entrada inválida para o número de páginas!\n");  
    return 1; // Encerra o programa com erro  
}
```

Evite o uso de variáveis globais

- Variáveis globais podem levar a código difícil de depurar e manter.
- Prefira passar variáveis como parâmetros para funções.

Exemplo:

```
void processarDados(int dados) {  
    // Função usa parâmetros em vez de variáveis globais  
}
```

Use constantes para valores fixos

- Evite usar valores "hardcoded" diretamente no código.
 - Em programação, o termo hardcoded (ou codificado de forma fixa) se refere a valores definidos diretamente no código-fonte.
- Defina constantes com #define ou const.

Exemplo:

```
#define MAX_PAGINAS 1000  
const float PI = 3.14159;
```

Indentação e formatação

- Use indentação consistente para melhorar a legibilidade.
- Escolha um estilo de formatação (como K&R (Kernighan & Ritchie) ou Allman) e siga-o em todo o código.
 - **K&R**: ideal para projetos que priorizam compactação do código e são baseados em convenções já estabelecidas (como o kernel, do Linux).

Principais características são:

- **Chaves ({}) na mesma linha**

- A chave de abertura { fica na mesma linha da declaração da função, estrutura de controle (if, for, while, etc.) ou bloco de código.
- A chave de fechamento } fica em uma nova linha, alinhada com o início da declaração.

Exemplo:

```
if (condicao) {  
    // código  
}
```

- **Indentação**

- O código dentro de um bloco é indentado (geralmente com quatro espaços ou um tab).

- **Espaçamento**

- Espaços são usados para separar operadores e após vírgulas.

Exemplo:

```
if (a == b && c > d) .
```

- **Vantagens**

- Ocupa menos espaço vertical, tornando o código mais compacto.
- Amplamente adotado na comunidade C.

- **Desvantagens**

- Alguns desenvolvedores consideram menos legível, especialmente em blocos de código longos.
- **Allman**: recomendado para projetos que valorizam clareza e legibilidade, especialmente em equipes grandes ou com código complexo.

Principais características são:

- **Chaves ({}) em linhas separadas**

- A chave de abertura { e a chave de fechamento } ficam em linhas separadas, alinhadas com a declaração do bloco, tornando os blocos de código mais destacados.

Exemplo:

```
if (condicao)
{
    // código
}
```

- **Indentação**

- O código dentro de um bloco é indentado (geralmente com quatro espaços ou um tab).

- **Espaçamento**

- Similar ao K&R, espaços são usados para separar operadores e após vírgulas.

Exemplo:

```
if (a == b && c > d)
```

- **Vantagens**

- Facilita a leitura e a identificação de blocos de código.
 - Ideal para códigos longos ou complexos.

- **Desvantagens**

- Ocupa mais espaço vertical, o que pode aumentar o tamanho do código.

Evite funções muito longas

- Divida funções grandes em funções menores e mais especializadas.
- Isso facilita a leitura, o teste e a manutenção.

Exemplo:

```
void processarEntrada() {
    lerDados();
    validarDados();
    salvarDados();
}
```

Testes e depuração

- Teste o código em diferentes cenários para garantir que ele funcione corretamente.
- Use ferramentas de depuração, como GNU Debugger (GDB), para identificar e corrigir problemas. O GDB é uma das ferramentas de depuração mais populares para linguagens como C e C++. Ele é gratuito, open-source e suporta uma variedade de plataformas e arquiteturas.
- Ferramentas de depuração são programas que ajudam desenvolvedores a encontrar e corrigir erros (bugs) em seu código. Elas permitem:
 - Executar o código linha por linha.
 - Inspecionar o valor de variáveis em tempo de execução.
 - Analisar a pilha de chamadas (call stack) para entender o fluxo do programa.
 - Detectar vazamentos de memória, falhas de segmentação (segmentation faults) e outros problemas comuns.

Padrões de codificação e documentação

- Siga um padrão de codificação consistente, como o GNU Coding Standards ou o Google C++ Style Guide (adaptado para C).
- Documente o código, especialmente funções públicas e APIs.
- Use ferramentas para gerar documentação automaticamente.



Lembrete

Todo programa em C precisa ter uma função principal chamada `main()`. O uso correto de `printf()` e `scanf()` é essencial para a entrada e saída de dados. Cada tipo de dado em C tem seu próprio especificador de formato no `printf` e `scanf`. Variáveis locais só existem dentro do bloco onde foram declaradas, enquanto variáveis globais existem durante toda a execução do programa.

Pontuação é crucial: sempre finalize comandos com `;`.

2 OPERADORES BÁSICOS EM C

Agora vamos estudar os operadores básicos utilizados na linguagem C, elementos essenciais para a construção de expressões e execução de operações fundamentais em qualquer programa.

O objetivo é desenvolver o raciocínio lógico e compreender como os operadores são usados para manipular dados, realizar comparações e construir instruções básicas que formarão a base de estruturas mais complexas.

Os operadores básicos em C são essenciais para a manipulação de dados e o controle de fluxos dentro do programa. Eles incluem operadores aritméticos, relacionais, lógicos, de atribuição, incremento/decremento. Compreender esses operadores é fundamental para escrever código eficiente e otimizado em C.

2.1 Operadores de atribuição, aritméticos, relacionais, lógicos e de incremento/decremento

Na linguagem C, os operadores são símbolos que realizam operações sobre variáveis e valores. Eles são vitais para manipulação de dados e controle do fluxo do programa.

2.1.1 Operadores de atribuição

São usados para atribuir valores a variáveis. O operador básico é o `=`, que atribui o valor à direita à variável à esquerda. Além disso, existem operadores combinados, como `+=`, `-=`, `*=`, `!/=`, e `%=`, que realizam uma operação aritmética antes de atribuir o valor. Por exemplo, `a += 3` é equivalente a `a = a + 3`. Esses operadores são úteis para simplificar expressões e reduzir a repetição de código.

A tabela 4 apresenta os operadores de atribuição em C, destacando suas funcionalidades e aplicações práticas:

Tabela 4 – Operadores de atribuição

Operador	Equivalente a	Exemplo	Resultado
<code>=</code>	<code>x = y</code>	<code>x = y</code>	Atribui y a x
<code>+=</code>	<code>x = x + y</code>	<code>x += y</code>	Soma y a x e armazena em x
<code>-=</code>	<code>x = x - y</code>	<code>x -= y</code>	Subtrai y de x e armazena em x
<code>*=</code>	<code>x = x * y</code>	<code>x *= y</code>	Multiplika x por y e armazena em x
<code>/=</code>	<code>x = x / y</code>	<code>x /= y</code>	Divide x por y e armazena em x
<code>%=</code>	<code>x = x % y</code>	<code>x %= y</code>	Calcula o resto da divisão de x por y e armazena em x

Exemplo:

```
int a = 5; // a recebe o valor 5
a += 3; // Equivalente a a = a + 3; (a agora vale 8)

int num = 10;
num *= 5; // Equivale a num = num * 5 (num agora vale 50)
```

2.1.2 Operadores aritméticos

São utilizados para realizar operações matemáticas básicas, como soma (+), subtração (-), multiplicação (*), divisão (/) e o operador de módulo (%) que retorna o resto de uma divisão entre dois inteiros. Esses operadores são essenciais para qualquer tipo de cálculo ou manipulação de valores dentro de um programa.

A seguir, temos um programa que demonstra o uso desses operadores.

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

int main() {
    setlocale(LC_ALL, "Portuguese");

    int a = 10, b = 3;

    printf("Soma: %d\n", a + b);
    printf("Subtração: %d\n", a - b);
    printf("Multiplicação: %d\n", a * b);
    printf("Divisão: %d\n", a / b);
    printf("Resto: %d\n", a % b);

    return 0;
}
```

Explicação resumida do código:

- **Duas variáveis inteiras são declaradas e inicializadas:**
 - a receber o valor 10
 - b recebe o valor 3
- **Operações realizadas:**
 - Soma (a + b)

- $10 + 3 = 13$
- Saída: Soma: 13
- **Subtração ($a - b$):**
 - $10 - 3 = 7$
 - Saída: Subtração: 7
- **Multiplicação ($a * b$):**
 - $10 * 3 = 30$
 - Saída: Multiplicação: 30
- **Divisão (a / b):**
 - $10 / 3 = 3$ (divisão inteira, pois a e b são inteiros)
 - Saída: Divisão: 3
- **Resto da divisão ($a \% b$):**
 - $10 \% 3 = 1$ (resto da divisão de 10 por 3)
 - Saída: Resto: 1

Saída na tela

```
Soma: 13
Subtração: 7
Multiplicação: 30
Divisão: 3
Resto: 1
```

Figura 13

Na divisão inteira, como a e b são inteiros, a divisão a / b resulta em um valor inteiro (3), descartando a parte decimal.

Já no resto da divisão, o operador $\%$ retorna o resto da divisão inteira. No caso, $10 \% 3$ resulta em 1, porque 10 dividido por 3 é 3 com resto 1.

Exemplos dessa operação:

- $10 \% 3$ retorna 1, porque 10 dividido por 3 é 3 com resto 1
- $10 \% 2$ retorna 0, porque 10 dividido por 2 é 5 com resto 0
- $9 \% 3$ retorna 0, porque 9 dividido por 3 é 3 com resto 0
- $9 \% 2$ retorna 1, porque 9 dividido por 2 é 4 com resto 1

Para encontrar o resto, fazemos a operação de divisão, conforme exemplo: $10 \div 3 = 3$ com resto 1.

O resultado aqui é apresentado como um quociente inteiro e um resto

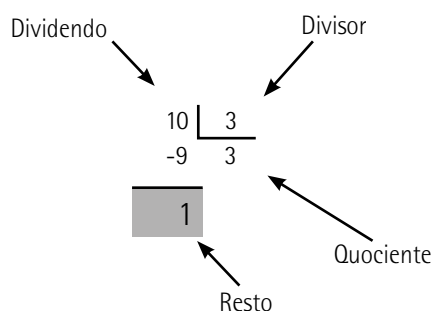


Figura 14

Cuidado: verifique se o **divisor é diferente de zero** para evitar erros de divisão por zero.



Lembrete

O **quociente** é o valor inteiro obtido na divisão. Se a divisão for exata, o **quociente** e o **resultado** são a mesma coisa. O resto é a parte que sobra após a divisão.

2.1.3 Operadores relacionais

Eles são usados na linguagem C para comparar dois valores, sejam eles variáveis ou constantes. O resultado dessas comparações é interpretado como um valor lógico: 1 quando a expressão é verdadeira e 0 quando é falsa.

Exemplo de operadores relacionais: ($=$, $>$, $<$, $>=$, $<=$, $!=$).

A tabela 5 traz operadores relacionais em linguagem C. Para tal, mostra exemplos que retornam true (1) e false (0), traduzindo (verdadeiro ou falso) cobrindo todas as possibilidades.

Tabela 5 – Operadores de atribuição

Operador	Descrição	Exemplo	Resultado
==	Igual a	5 == 5	1 (true)
!=	Diferente de	5 != 3	1 (true)
>	Maior que	10 > 5	1 (true)
<	Menor que	5 < 10	1 (true)
>=	Maior ou igual a	10 >= 10	1 (true)
<=	Menor ou igual a	5 <= 10	1 (true)
==	Igual a	5 == 3	0 (false)
!=	Diferente de	5 != 5	0 (false)
>	Maior que	5 > 10	0 (false)
<	Menor que	10 < 5	0 (false)
>=	Maior ou igual a	5 >= 10	0 (false)
<=	Menor ou igual a	10 <= 5	0 (false)

Agora vamos explicar as linhas da tabela.

Exemplos que retornam verdadeiro (1 – true):

- 5 == 5 retorna 1 (true), pois 5 é igual a 5, então a condição é verdadeira.
- 5 != 3 retorna 1 (true), pois 5 é diferente de 3, então a condição é verdadeira.
- 10 > 5 retorna 1 (true), pois 10 é maior que 5, então a condição é verdadeira.
- 5 < 10 retorna 1 (true), pois 5 é menor que 10, então a condição é verdadeira.
- 10 >= 10 retorna 1 (true), pois 10 é maior ou igual a 10, então a condição é verdadeira.
- 5 <= 10 retorna 1 (true), pois 5 é menor ou igual a 10, então a condição é verdadeira.

Exemplos que retornam falso (0 – false):

- 5 == 3 retorna 0 (false), pois 5 não é igual a 3, então a condição é falsa.
- 5 != 5 retorna 0 (false), pois 5 não é diferente de 5, então a condição é falsa.
- 5 > 10 retorna 0 (false), pois 5 não é maior que 10, então a condição é falsa.
- 10 < 5 retorna 0 (false), pois 10 não é menor que 5, então a condição é falsa.
- 5 >= 10 retorna 0 (false), pois 5 não é maior ou igual a 10, então a condição é falsa.
- 10 <= 5 retorna 0 (false), pois 10 não é menor ou igual a 5, então a condição é falsa.



Lembrete

Se a comparação for verdadeira, retorna 1 (true). Se a comparação for falsa, retorna 0 (false).

Os operadores relacionais são usados em estruturas condicionais, como if, while e for.

2.1.4 Operadores lógicos

Permitem a construção de expressões lógicas mais complexas. Eles combinam múltiplas condições para criar verificações mais detalhadas, sendo úteis quando é necessário testar várias condições ao mesmo tempo dentro de uma instrução condicional.

Em C, existem três operadores lógicos principais, conforme exposto no quadro 3.

Quadro 3 – Operadores de lógicos em C

Operador	Descrição
AND (&&)	Retorna verdadeiro (1) apenas se ambas as condições forem verdadeiras
OR ()	Retorna verdadeiro (1) se pelo menos uma das condições for verdadeira
NOT (!)	Inverte o valor lógico de uma condição, transformando verdadeiro em falso e vice-versa

Ordem de precedência dos operadores lógicos:

! (NOT) - Mais alto
&& (AND)
|| (OR) - Mais baixo

Se houver múltiplos operadores em uma expressão sem parênteses, o operador de maior precedência será avaliado primeiro.

Exemplo:

```
int resultado = !0 || 0 && 1;
```

A avaliação ocorre assim:

```
!0 → 1 (NOT tem a maior precedência)  
0 && 1 → 0 (AND tem precedência maior que OR)  
1 || 0 → 1 (OR é o último a ser avaliado)
```


Logo, a variável resultado passa a ter o valor 1.

Pergunta: o que acontece, dentro de uma estrutura condicional if, quando a condição avaliada é verdadeira?

Resposta: o bloco de código dentro do if é executado!

Os operadores lógicos em C são essenciais para a criação de condições em estruturas de controle. O uso correto deles permite escrever código mais eficiente e claro. O entendimento da precedência também ajuda a evitar resultados inesperados em expressões booleanas.

A seguir, exploraremos o uso desses operadores e suas aplicações.

2.1.5 Operadores de incremento/decremento

São usados para aumentar ou diminuir o valor de uma variável.

Podemos utilizar para somar ou subtrair 1 de uma variável.

`++` Incrementa em 1 (`x++` ou `++x`)

`--` Decrementa em 1 (`x--` ou `--x`)

A tabela 6 traz exemplos de operadores de incremento/decremento em linguagem C.

Tabela 6

Operador	Descrição	Exemplo de uso	Resultado
<code>++x</code>	Pré-incremento	<code>int x = 5;</code> <code>++x</code>	<code>(++x)</code> x passa a ser 6 antes da linha atual ser avaliada
<code>x++</code>	Pós-incremento	<code>int x = 5;</code> <code>x++;</code>	<code>(x++)</code> x ainda é 5 na linha atual, só depois vira 6
<code>--x</code>	Pré-decremento	<code>int x = 5;</code> <code>--x</code>	<code>(--x)</code> x passa a ser 4 antes da linha atual ser avaliada
<code>x--</code>	Pós-decremento	<code>int x = 5;</code> <code>x--</code>	<code>(x--)</code> x ainda é 5 na linha atual, só depois vira 4

Exemplo:

```
int contador = 0;
contador++;
printf("Contador: %d\n", contador); // Imprime Contador: 1
```

Se tivesse sido escrito `printf("Contador: %d\n", contador++);`, o valor exibido seria 0 (valor antigo), e o contador só seria incrementado depois do `printf`.

```
int contador = 0;
printf("Contador: %d\n", contador++); // Imprime Contador: 0
// na sequência a variável contador passa a valer 1
```

Exemplo:

```
int a = 5, b = 5;
printf("Pré-incremento: %d", ++a); // Imprime 6
printf("Pós-incremento: %d", b++); // Imprime 5
printf("Valor de b agora: %d", b); // Imprime 6

int c = 5, d = 5;
printf("Pré-decremento: %d", --c); // Imprime 4
printf("Pós-decremento: %d", d--); // Imprime 5
printf("Valor de d agora: %d", d); // Imprime 4
```

Os operadores são elementos vitais em qualquer linguagem de programação. Com eles, realizamos cálculos, comparações, construímos lógicas de decisão e controlamos o fluxo dos programas. Entender bem seu funcionamento é essencial para evoluir na programação e dominar estruturas mais avançadas.



Saiba mais

Aprofunde seus conhecimentos lendo a obra a seguir:

MELO, A. C. V.; SILVA, F. S. C. *Princípios de linguagens de programação*. São Paulo: Edgard Blucher, 2010.

2.1.6 Aplicações e exemplos do uso de operadores na linguagem C

A seguir, apresentamos aplicações comuns dos operadores em C e exemplos práticos que ilustram como os operadores básicos em C são aplicados no dia a dia da programação.

Cálculos matemáticos

São feitos usando operadores aritméticos (+, -, *, /, %) para somas, médias, porcentagens, descontos, área de um triângulo, o IMC, juros compostos etc.

Exemplo: calcular a área de um triângulo e imprimir com duas casas decimais.

```
float base = 5, altura = 10;
float area = (base * altura) / 2.0;
printf("Área: %.2f", area);
```

Operadores utilizados: multiplicação*, divisão / e atribuição =

Exemplo: calcular média de três notas e imprimir com duas casas decimais.

```
float n1 = 7.5, n2 = 8.0, n3 = 9.0;  
float media = (n1 + n2 + n3) / 3;  
printf("Média: %.2f", media);
```

Operadores utilizados: adição +, divisão / e atribuição =

Comparações e decisões

Em C, usamos operadores relacionais e lógicos (==, !=, >, <, &&, ||, !) para construir expressões que controlam o fluxo de execução do programa, como em instruções if, while, entre outras.

Atualização de variáveis

Os operadores de atribuição composta (+=, -=, *=, /=, %=) são usados para atualizar variáveis de forma prática e eficiente, combinando uma operação aritmética com a atribuição.

Exemplo: somar pontos em um jogo, calcular saldo de conta e acumular somas.

pontos += 50; // aumenta 50 pontos na variável pontos



Lembrete

Os operadores são símbolos que executam operações aritméticas, lógicas, relacionais ou de atribuição. A divisão entre inteiros em C descarta as casas decimais. O operador % (módulo) retorna o resto da divisão entre dois inteiros. Operadores relacionais sempre retornam 0 (falso) ou 1 (verdadeiro). Use ++ ou -- para incrementar ou decrementar valores de forma prática.



Resumo

Nesta unidade, aprendemos os primeiros passos da programação em C. Vimos como a linguagem C é importante por ser rápida, próxima do hardware e base para outras linguagens modernas.

Conhecemos a estrutura básica de um programa em C, incluindo a função `main()`, o uso de bibliotecas com `#include` e comandos como `printf()` para mostrar mensagens na tela e `scanf()` para ler dados do usuário.

Estudamos os tipos de dados, como `int`, `float`, `char` e `double` e aprendemos como declarar variáveis para armazenar informações. Também vimos a diferença entre variáveis locais e globais e a importância do escopo no programa.

Exploramos os operadores básicos, como os operadores matemáticos (+, -, *, /, %), os relacionais (==, !=, > etc.), os lógicos (&&, ||, !) e os de incremento e decremento (++ , --).

Por fim, discutimos boas práticas de programação, como usar nomes claros nas variáveis, manter o código bem organizado e evitar repetir o código desnecessariamente.

Esses conhecimentos são a base para continuar avançando na linguagem C e desenvolver programas cada vez mais completos.



Exercícios

Questão 1. Avalie as afirmativas a seguir sobre o tipo de dado "char" na linguagem C.

I – O tipo char é utilizado tipicamente para armazenar números flutuantes, ou seja, números reais.

II – O tamanho padrão de armazenamento de uma variável declarada com o tipo "char" é 4 bytes, o que equivale a 16 bits.

III – Tipicamente, utilizamos variáveis declaradas com o tipo "char" para armazenar um único caractere, como a letra "a". Se quisermos armazenar uma palavra ou uma sentença completa, podemos usar um array (ou vetor) de "char", no qual cada elemento do vetor corresponde a um caractere.

É correto apenas o que se afirma em:

A) I.

B) II.

C) III.

D) I e II.

E) I e III.

Resposta correta: alternativa C.

Análise das afirmativas

I – Afirmativa incorreta.

Justificativa: o tipo "char" é, na realidade, um tipo inteiro utilizado para armazenar apenas números inteiros pequenos. Cada valor ocupa 1 byte ou 8 bits, o que limita fortemente o tamanho dos valores que podem ser armazenados em uma variável do tipo "char".

II – Afirmativa incorreta.

Justificativa: há um erro com relação à conversão entre bits e bytes: 4 bytes correspondem a 32 bits, e não a 16 bits. Observe que $4 \times 8 = 32$.

III – Afirmativa correta.

Justificativa: a afirmativa ilustra usos típicos de dado "char": armazenar caracteres simples ou, quando utilizado como um array (vetor), armazenar palavras inteiras, frases etc.

Questão 2. Avalie as afirmativas a seguir sobre os usos das funções "fgets" e "scanf" na linguagem C.

I – A função "fgets" pode ser utilizada para a leitura de cadeias longas de texto, como frases completas, e possibilita o uso de espaços em branco. A função também recebe o tamanho máximo de caracteres que podem ser lidos, o que é uma garantia para evitar problemas como a invasão ou o transbordo de memória.

II – A função "fgets" interrompe a leitura dos dados de entrada assim que encontra um espaço em branco.

III – A função "scanf" é totalmente segura para a entrada de dados, independentemente da forma como ela seja utilizada.

É correto apenas o que se afirma em:

A) I.

B) II.

C) III.

D) I e II.

E) I e III.

Resposta correta: alternativa A.

Análise das afirmativas

I – Afirmativa correta.

Justificativa: a função "fgets" é uma forma mais segura para a entrada de dados textuais complexos na linguagem C, especialmente caso não sejam utilizadas bibliotecas adicionais para a entrada de dados, que podem ter funções mais sofisticadas para essa finalidade.

II – Afirmativa incorreta.

Justificativa: a função "fgets" lê a entrada para as seguintes condições:

- Até encontrar uma quebra de linha (representada, na linguagem C, pelo caractere "\n").
- Até o final do arquivo (indicado com o EOF).
- Até o limite máximo de caracteres, que é um dos seus argumentos.

Essa função não interrompe a leitura quando encontra um caractere em branco.

É importante ter muito cuidado ao utilizar a função "scanf" para a entrada de dados: usar essa função sem um limite de leitura pode ocasionar invasões (ou transbordos) de memória, o que pode gerar erros na execução do programa.

III – Afirmativa incorreta.

Justificativa: mesma explicação da afirmativa anterior.

This image shows a single sheet of white paper with horizontal ruling lines. The lines are evenly spaced and run across the width of the page. There are no margins, text, or other markings on the paper.