

Unidade IV

7 FUNÇÕES

7.1 Funções em C

Em C, função é um bloco de código que executa uma tarefa específica e pode ser reutilizado várias vezes no programa. O uso de funções facilita a organização, a legibilidade e a manutenção do código.

Estrutura de uma função:

```
tipo_retorno nome_funcao(lista de parâmetros) {  
    ...  
}
```

Exemplo:

```
int calcula_quadrado(int num) {  
    return num * num;  
}
```

Usado para retornar
o valor da função

Em C, apenas um valor
pode ser retornado

Figura 48

Observe seus componentes a seguir:

- **Tipo de retorno:** define o tipo de dado que a função devolve (int, float, void etc.).
- **Nome da função:** identificador usado para chamá-la.
- **Parâmetros:** são os dados recebidos (opcional).
- **Corpo da função:** instruções que serão executadas.
- **return:** interrompe a função e retorna o resultado (exceto em funções do tipo void, pois esse tipo não tem retorno).

As funções são utilizadas para dividir um programa em módulos menores e reutilizáveis, facilitando a organização e a manutenção do código. Ao definir uma função, as variáveis criadas dentro dela são chamadas de variáveis locais, ou seja, só podem ser usadas dentro daquele bloco de função.

Uso comum de funções:

- Para modularizar o código.
- Uma função executa uma operação e retorna um valor.

Procedimento:

- Um procedimento é uma função que não retorna um valor.
- Usamos void para indicar isso.

```
void mostra_quadrado(int num) {  
    printf("%d\n", num * num);  
}
```

A função main em C deve ser declarada como `int main ()`, conforme o padrão ISO C (normas ANSI/ISO).

```
int main() {  
    ...  
  
    return 0;  
}
```

- main é uma função também. Ela é chamada automaticamente quando o programa é iniciado.
- A função main retorna um inteiro:
 - Retorna 0 quando o programa termina como esperado.
 - Retorna outro valor em caso de erro!
- Por convenção, a função `main()` retorna um valor inteiro ao sistema operacional ao final da execução.

7.1.1 Chamada de funções

Para que uma função seja executada, é necessário chamá-la em algum ponto do programa. A chamada é feita usando o nome da função seguido dos argumentos entre parênteses, caso existam. Esses argumentos fornecem os dados que a função precisa para realizar sua tarefa.

No primeiro exemplo, temos um programa que calcula o quadrado de um número usando função antes da função `main()`.

```
#include<stdio.h>

int calcula_quadrado(int num) {
    return num * num;
}

int main() {

    int n1;
    scanf("%d", &n1);

    int quad = calcula_quadrado(n1);

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

    return 0;
}
```




Figura 49 – Programa que calcula o quadrado de um número usando função

Esse programa lê um número inteiro digitado pelo usuário, calcula o seu quadrado por meio de uma função chamada `calcula_quadrado()` e depois imprime o resultado na tela.

- `int quad = calcula_quadrado(n1);` chamada da função desse exemplo. Aqui, o número digitado pelo usuário é passado como argumento para a função `calcula_quadrado`.
- O valor retornado pela função é armazenado na variável `quad`.

No exemplo acentuado, o código da função `calcula_quadrado()` vem antes da função `main()` corretamente. Caso a função esteja após a `main`, será necessário declarar o protótipo antes do `main`. Ou seja, podemos declarar as funções em qualquer ordem se declararmos seus protótipos no início.



Observação

A função `main()` também é uma função como qualquer outra em C, mas tem execução obrigatória e inicia o programa. Deve, preferencialmente, retornar `int`.

O programa a seguir calcula o quadrado de um número dentro de uma função e retorna o resultado.

Protótipo ou assinatura da função

```
#include<stdio.h>

int calcula_quadrado(int num);

int main() {
    int n1;
    scanf("%d", &n1);
    int quad = calcula_quadrado(n1);
    printf("%d\n", quad);
    return 0;
}

int calcula_quadrado(int num) {
    return num * num;
}
```

Figura 50 – Exemplo que calcula o quadrado de um número usando função

Observe a seguir a explicação:

- A declaração `int calcula_quadrado(int num);` diz ao compilador que existe uma função que retorna `int` e recebe um `int` como parâmetro.
- `calcula_quadrado(n1)`: chamada da função dentro do `main()`.
- A definição completa da função aparece depois do `main()` porque já foi previamente declarada.

7.2 Passagem de parâmetros

Quando uma função é chamada, os valores (ou referências) são passados para os parâmetros da função. Em C, a passagem de parâmetros padrão é por valor, o que significa que uma cópia do valor do argumento é passada para o parâmetro da função. Alterações no parâmetro dentro da função não afetam a variável original fora da função (a menos que sejam usados ponteiros para passar por referência).

Normalmente, uma função recebe valores através de parâmetros, uma forma de comunicação entre funções. Esses parâmetros são tratados como variáveis locais durante a execução da função.

Em C, todo parâmetro de função é passado por valor. Ou seja, a passagem padrão de argumentos para funções é por valor. Isso significa que, quando você chama uma função, uma cópia do valor do argumento é criada e passada para o parâmetro correspondente na função. Qualquer modificação feita no parâmetro dentro da função não afeta a variável original que foi passada como argumento no código de chamada.

Para passar um argumento por referência, precisamos passar o valor do endereço de memória (ponteiro). Portanto, você precisa explicitamente passar o endereço de memória da variável original para a função. Isso é feito utilizando o operador de endereço & antes do nome da variável no momento da chamada da função.

O parâmetro da função, por sua vez, deve ser declarado como um ponteiro para o tipo da variável original (usando o operador *). Dentro da função, você pode usar o operador * para acessar e modificar o valor contido no endereço de memória apontado pelo ponteiro, efetivamente alterando a variável original.

O código a seguir traz um exemplo para demonstrar a passagem por valor para função:

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

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

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

    int valor = 10;

    printf("Antes da chamada da função: valor = %d\n", valor);
    modificarValor(valor);
    printf("Depois da chamada da função: valor = %d\n", valor);
    return 0;
}
```

Podemos ver que a modificação da variável `x` dentro da função `modificarValor()` não afetou o valor da variável `valor` na função `main()`.

Conforme definição em linguagem C, todos os parâmetros de função são, por padrão, passados por valor. Isso significa que a função recebe uma cópia da variável original, e qualquer alteração feita dentro da função não afeta o valor da variável fora da função.

Saída na tela:

```
Antes da chamada da função: valor = 10
Dentro da função, antes da modificação: x = 10
Dentro da função, depois da modificação: x = 100
Depois da chamada da função: valor = 10
```

Figura 51

Na passagem por valor, a função recebe uma cópia e as alterações não afetam o original.

O exemplo a seguir ilustra a passagem por referência para função (usando ponteiros).

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

void modifReferencia(int *x) {
    printf("Dentro da função, antes da modificação: *ptr = %d\n", *x);
    *x = 100;
    printf("Dentro da função, depois da modificação: *ptr = %d\n", *x);
}

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

    int valor = 10;

    printf("Antes da chamada da função: valor = %d\n", valor);

    //Passa o endereço de memória de valor
    modifReferencia(&valor);

    printf("Depois da chamada da função: valor = %d\n", valor);

    return 0;
}
```

Nesse exemplo, ao passar o endereço de memória de valor para a função `modifReferencia`, a função pôde alterar o valor original da variável `valor` na função `main` através do ponteiro `*x`.

Saída na tela:

```
Antes da chamada da função: valor = 10
Dentro da função, antes da modificação: *ptr = 10
Dentro da função, depois da modificação: *ptr = 100
Depois da chamada da função: valor = 100
```

Figura 52

Na passagem por referência, a função recebe o endereço da variável e as alterações afetam o original.

7.3 Escopo de variáveis

Em C, o escopo de uma variável define a região do código onde essa variável é visível e pode ser acessada. O escopo de uma variável é determinado pelo local onde ela é declarada dentro do programa.

Existem escopos em C que determinam a visibilidade das variáveis, os principais incluem:

- **Variáveis locais (escopo de bloco ou de função):** acessíveis apenas dentro do bloco de código ou da função onde são declaradas.
- **Variáveis globais:** acessíveis a partir de qualquer parte do código.

Escopo local

As variáveis declaradas dentro de uma função têm escopo local, o que significa que elas só são acessíveis dentro dessa função. As variáveis declaradas dentro de um bloco de código (delimitado por chaves `{}`), como em estruturas de decisão `if`, ou de repetição como `for`, têm um escopo ainda mais restrito, existindo apenas dentro desse bloco. Isso evita conflitos de nomes com variáveis em outras partes do programa. Os parâmetros da função também são considerados variáveis locais dentro da função.

Escopo global

A variável global pode ser usada diretamente em qualquer função, sem passar como parâmetro. Embora conveniente, o uso excessivo de variáveis globais pode dificultar o rastreamento de onde os dados são modificados, tornando o código mais propenso a erros e menos modular.



Observação

Como a variável global é compartilhada, qualquer alteração em seu valor por uma função afetará o comportamento de todas as outras partes do código que dependem desse valor. Isso pode tornar a depuração mais difícil, pois a origem de uma modificação inesperada pode estar em qualquer lugar do programa.

No exemplo a seguir, o programa ilustra o uso de variáveis (locais e globais) com escopos diferentes em funções.

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

```
double resultado;
```

```
void calcula_quadrado(double num) {  
    resultado = num * num;  
}
```

```
double calcula_soma(double n1, double n2) {  
    double r;  
    r = n1 + n2;  
    return r;  
}
```

```
int main() {  
    int a = 2, b = 3;  
    resultado = calcula_soma(a, b);  
    printf("%.2lf\n", resultado);  
    calcula_quadrado(resultado);  
    printf("%.2lf\n", resultado);  
    calcula_quadrado(resultado);  
    printf("%.2lf\n", resultado);  
  
    return 0;  
}
```

Variável global

A variável resultado pode ser acessada a partir de qualquer função

Variável local da função calcula_soma

Variáveis locais da função main

Variáveis locais são acessíveis apenas da função onde foram declaradas

Figura 53 – Exemplo de uso de variável local e global

Em vermelho, temos uma variável global resultado, que pode ser acessada por qualquer parte do programa. Já em verde, estão destacadas as variáveis a e b, que são locais da função main(), ou seja, só podem ser usadas dentro dela. Há outra de cor verde, a variável r, usada dentro da função calcula_soma(), que também é local e visível apenas nesse escopo.

Além disso, os parâmetros das funções são considerados variáveis locais. Por exemplo, o parâmetro `num` da função `calcula_quadrado()` e os parâmetros `n1` e `n2` da função `calcula_soma()` são acessíveis apenas dentro de suas respectivas funções.

O próximo exemplo traz a modificação de variável global.

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

int par(int n) {
    return (n % 2 == 0);
}

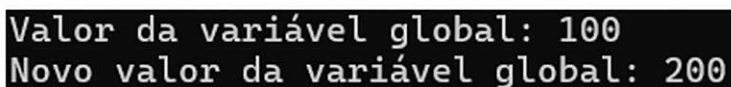
int main() {
    setlocale(LC_ALL, "Portuguese");

    int num = 10;

    if(par(num))
        printf("O número %d é par.\n", num);
    else
        printf("O número %d é ímpar.\n", num);

    return 0;
}
```

Saída na tela



```
Valor da variável global: 100
Novo valor da variável global: 200
```

Figura 54 – Exemplo de uso de variável global

No código anterior, 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 anterior, temos a declaração de uma variável global chamada `global` com o valor inicial 100. O código traz a definição de uma função chamada `exemplo()`, que exibe o valor atual da variável global. Lembre-se de que no `main()` configuramos a localidade para Português com a função `setlocale()`.

Observe o passo a passo a seguir:

- Chamamos a função `exemplo()` para mostrar o valor inicial da variável global.
- Depois, modificamos o valor da variável global para 200.
- O programa exibe no `printf()` o novo valor da variável global.

As vantagens das variáveis locais são:

- Melhor controle sobre o escopo e o uso da variável.
- Evitam conflitos de nomes com outras partes do programa.
- Facilitam o rastreamento de alterações e reduzem efeitos colaterais.

Como desvantagem, as variáveis locais não podem compartilhar informações diretamente entre funções.

Por sua vez, as vantagens das variáveis globais são:

- Permitem compartilhar dados entre várias funções.
- Permanecem na memória durante toda a execução do programa.

Como desvantagens, acentuam-se:

- Difícil rastrear alterações em programas grandes.
- Podem causar conflitos de nomes ou erros devido a alterações não intencionais.
- Consomem mais memória, pois permanecem alocadas durante toda a execução.

As variáveis globais são declaradas fora de qualquer função, normalmente no início do arquivo. São visíveis em qualquer parte do código, inclusive em outras funções. Devem ser usadas com cuidado para evitar conflitos e efeitos colaterais.

Já as variáveis estáticas de arquivo são declaradas fora das funções, mas com a palavra-chave `static`. Isso restringe o uso da variável ao arquivo onde foi declarada, ou seja, outras unidades de código não conseguem acessá-la (mesmo que incluam esse arquivo).

Por sua vez, os protótipos de função são declarações usadas para informar ao compilador a assinatura da função antes de sua definição.

Os nomes dos parâmetros declarados no protótipo não têm escopo real, ou seja, são ignorados, apenas os tipos importam.

Cada tipo de escopo define a visibilidade e o tempo de vida de uma variável ou função, o que impacta diretamente o comportamento do programa.

Para trabalhar com as boas práticas, deve-se observar os itens a seguir:

- Embora variáveis globais sejam convenientes, seu uso deve ser limitado.
- Prefira variáveis locais sempre que possível, para evitar efeitos colaterais e manter o código modular.
- Nomeie variáveis globais de forma clara e única para evitar confusões.
- Considere passar variáveis como parâmetros entre funções em vez de usar globais.



Lembrete

Toda função em C precisa ter um tipo de retorno (como int, float, void), um nome (identificador), parênteses () com ou sem parâmetros e um corpo com comandos entre {}.

7.4 Aplicações de funções em C: exemplos

No exemplo a seguir, o programa verifica se o número armazenado na variável num é par. Para tal, utiliza uma função chamada par() para fazer essa verificação.

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

int par(int n) {
    return (n % 2 == 0);
}

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

    int num = 10;

    if(par(num))
        printf("O número %d é par.\n", num);
    else
        printf("O número %d é ímpar.\n", num);

    return 0;
}
```

Explicação:

- **int par(int n):** essa função recebe um número inteiro n.
- Faz a verificação `if(n % 2 == 0)`, que retorna:
 - 1 (verdadeiro) se o número for par.
 - 0 (falso) se o número for ímpar.
- **if(par(num)):** chama a função `par()` com o valor de `num` como argumento.
 - Se a função retornar 1, imprime: "O número 10 é par. " .
 - Se retornar 0, imprime: "O número 10 é ímpar. " .

Saída na tela:

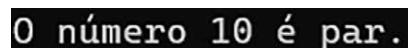
A screenshot of a terminal window with a black background and white text. The text displayed is "O número 10 é par.".

Figura 55

No exemplo a seguir, o programa calcula a área de um retângulo utilizando uma função que recebe a base e a altura como parâmetros e retorna o valor da área. Os dados são informados pelo usuário e o resultado é exibido na tela.

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

float calcular_area(float base, float altura) {
    return base * altura;
}

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

    float b, h;

    printf("Digite a base: ");
    scanf("%f", &b);

    printf("Digite a altura: ");
    scanf("%f", &h);

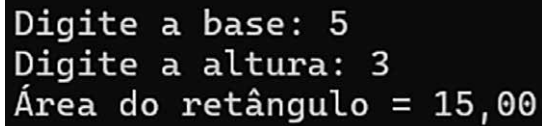
    float area = calcular_area(b, h);

    printf("Área do retângulo = %.2f\n", area);

    return 0;
}
```

A função `calcular_area()` recebe dois valores de ponto flutuante (`float`) (`base` e `altura`) e retorna o valor da multiplicação. No `main()`, lê os dados e passa as variáveis `b` e `h` como argumentos para a função `calcular_area()`. Após o retorno da função, exibe o resultado.

Saída na tela: simulação com valores de base 5 e altura 3.



```
Digite a base: 5
Digite a altura: 3
Área do retângulo = 15,00
```

Figura 56

O programa a seguir é usado para verificar maioridade. Ele determina se uma pessoa é maior ou menor de idade com base em uma função que recebe a idade e retorna 1 se for 18 anos ou mais; caso contrário, retorna 0.

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

int maior_de_idade(int idade);

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

    int idade;

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

    if (maior_de_idade(idade))
        printf("É maior de idade.\n");
    else
        printf("É menor de idade.\n");

    return 0;
}

int maior_de_idade(int idade) {
    return idade >= 18;
}
```

Nesse código, a função `maior_de_idade()` está declarada (protótipo) antes do `main()` e definida depois do `main()`, boa prática.

A função `maior_de_idade()` retorna 1 (verdadeiro), ou seja, se a idade for maior ou igual a 18, e 0 (falso) se for menor.

Na função `main()`, a idade informada pelo usuário é lida. Chama a função `maior_de_idade(idade)` e verifica se retornou verdadeiro (1).

- Se sim, imprime "É maior de idade. " .
- Caso contrário, imprime "É menor de idade. " .

O programa acentuado a seguir calcula o dobro de um número (void). Então, exibe o dobro de um número informado pelo usuário utilizando uma função void, que recebe o número como parâmetro e imprime o resultado diretamente.

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

void mostrar_dobro(int x);

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

    int num;
    printf("Digite um número: ");
    scanf("%d", &num);

    mostrar_dobro(num);

    return 0;
}

void mostrar_dobro(int x) {
    printf("O dobro de %d é %d\n", x, x * 2);
}
```

Na função principal `main()`, temos:

- **Declaração da variável num, depois:** solicita e lê do usuário um número inteiro e armazena em `num`.
- **Chama a função `mostrar_dobro`:** passando como argumento o número lido armazenado em `num`.

A função `mostrar_dobro()` não retorna nada (`void`), mas imprime diretamente o resultado da operação dentro dela. Aqui a função recebe um valor inteiro `x`, calcula seu dobro ($x * 2$) e **exibe o resultado** com `printf`.

No exemplo a seguir, o programa conta caracteres de um nome. Ele exibe quantos caracteres tem um nome digitado pelo usuário utilizando uma função que percorre a string manualmente (sem usar `strlen`).

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

int contar_caracteres(char nome[]);

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

    char nome[50];
    printf("Digite seu nome: ");
    scanf("%[^\\n]", nome); // lê nome com espaços

    int total = contar_caracteres(nome);
    printf("O nome possui %d caracteres.\\n", total);

    return 0;
}

int contar_caracteres(char nome[]) {
    int i = 0;
    //conta junto os espaços entre nome e sobrenome
    while(nome[i] != '\\0'){
        i++;
    }
    return i;
}
```

A função percorre o vetor de `char` até encontrar o caractere `'\\0'`, que marca o final da string, e conta quantos caracteres foram lidos.

- **`int contar_caracteres(char nome[]);`**: protótipo da função que receberá uma string e retornará (a quantidade de caracteres).
- **`char nome[50];`**: declara um vetor de caracteres (string) com capacidade para até 49 caracteres mais o `\\0`.
- **`scanf("%[^\\n]", nome);`**: usa `"%[^\\n]"` para ler nomes com espaços.
- **`int total = contar_caracteres(nome);`**: chama a função passando o nome digitado. O retorno da função armazena em `total`.
- **`printf()`**: exibe a quantidade total de caracteres digitados.

O programa a seguir conta caracteres de um nome. Ele exibe quantos caracteres possui um nome digitado pelo usuário utilizando a função `strlen` do arquivo de cabeçalho `<string.h>` para calcular o comprimento da string.

```
#include <stdio.h>
#include <string.h> //Necessária para usar strlen
#include <locale.h>

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

    char nome[100];
    int tamanho;

    printf("Digite seu nome: ");
    //Lê até a quebra de linha (com espaços)
    scanf(" %[^\\n]", nome);

    tamanho = strlen(nome);

    printf("Seu nome tem %d caracteres.\\n", tamanho);

    return 0;
}
```

Com o uso da função `strlen`, o programa fica mais simples. A função percorre o vetor `char` até encontrar o caractere especial `'\\0'`, que indica o final da string, e retorna a quantidade de caracteres armazenados até esse ponto. Observe o esquema a seguir:

- **Inclusão do arquivo de cabeçalho:** `string.h`: onde está a função `strlen`.
- **Declaração de variáveis:**
 - `char nome[100];`: vetor de caracteres para armazenar o nome.
 - `int tamanho;`: armazena o número de caracteres.
- O comando `scanf(" %[^\\n]", nome);` lê uma linha inteira com espaços.
- **Uso de `strlen`:** a função `strlen(nome)` retorna o total de caracteres digitados sem contar o caractere nulo `\\0`.
- **Saída:** exibe o número de caracteres do nome digitado pelo usuário.

8 PONTEIROS E MANIPULAÇÃO DE ARQUIVOS

8.1 Definição e uso de ponteiros em C

Ponteiro é uma variável que armazena um endereço de memória de outra variável. Ou seja, o ponteiro guarda o endereço (local na memória) de uma variável em vez de valores, serve para modificar valores diretamente.

Para declarar um ponteiro, basta incluir um asterisco antes do nome da variável:

```
int *p;
```

Nesse exemplo, p é um ponteiro que pode armazenar o endereço de uma variável do tipo int.

Operadores com ponteiros:

- **& (e comercial)**: retorna o endereço de uma variável.
- *** (Asterisco)**: usado para acessar o valor que está sendo apontado.

Um dos recursos mais poderosos da linguagem C, os ponteiros em C permitem manipular diretamente a memória e são essenciais para muitas operações no desenvolvimento de programas eficientes. Observe a seguir seus principais usos:

- **Acesso direto à memória**: ponteiros armazenam endereços de variáveis, o que permite acessar ou modificar diretamente os dados armazenados na memória.
- **Passagem por referência em funções (structs)**: com ponteiros, podemos passar o endereço de uma variável para uma função, permitindo que a função modifique o valor original.
- **Manipulação de arrays**: arrays são, na prática, ponteiros para a primeira posição de uma sequência de elementos.
- **Manipulação de strings**: strings em C são arrays de caracteres e seu uso eficiente depende de ponteiros.
- **Alocação dinâmica de memória**: ponteiros são usados para criar estruturas com tamanho definido em tempo de execução.
- **Trabalhar com estruturas (structs)**: ponteiros facilitam a manipulação de estruturas grandes, economizando memória e tempo de processamento.

Destaca-se a seguir um exemplo simples com uso básico de ponteiros em C.

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

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

    int x = 10;

    int *p;

    p = &x; // p recebe o endereço de x

    printf("Valor de x = %d\n", x);
    printf("Endereço de x = %p\n", &x);
    printf("Endereço armazenado em p = %p\n", p);
    printf("Valor apontado por p = %d\n", *p);

    return 0;
}
```

Observe a explicação a seguir:

- x é uma variável comum.
- p é um ponteiro que recebe o endereço de x com `p = &x;`
- `*p` acessa o valor de x.
- `&x` mostra o endereço de memória da variável x.

Como fica a saída na tela?

Os valores do endereço de x mostrados na tela podem variar em cada execução do programa, porque o endereço na memória é determinado pelo sistema operacional no momento da execução do programa.

```
Valor de x = 10
Endereço de x = 0061FF18
Endereço armazenado em p = 0061FF18
Valor apontado por p = 10
```

Figura 57

Agora temos um programa que mostra como acessar e alterar o valor de uma variável por meio de um ponteiro.

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

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

    int x = 10;
    int *p = &x;

    printf("Valor de x: %d\n", x);
    printf("Endereço de x: %p\n", &x);
    printf("Valor apontado por p: %d\n", *p);

    *p = 200;
    printf("Novo valor de x: %d\n", x);

    return 0;
}
```

Observe a seguir a explicação simplificada:

- p aponta para o endereço de x.
- Ao fazer *p = 20, alteramos o valor de x diretamente.
- Mostramos o endereço de x e como o ponteiro acessa o mesmo valor.

Saída na tela:

```
Valor de x: 10
Endereço de x: 0061FF18
Valor apontado por p: 10
Novo valor de x: 200
```

Figura 58

O programa acentuado a seguir troca os valores de duas variáveis utilizando ponteiros em uma função.

```
#include <stdio.h>

void trocar(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int x = 5, y = 10;
    printf("Valor original: x = %d, y = %d\n", x, y);

    trocar(&x, &y);
    printf("Valor trocado: x = %d, y = %d\n", x, y);

    return 0;
}
```

As variáveis originais são modificadas porque foram passadas por referência.

- Os ponteiros a e b recebem os endereços de x e y.
- A troca de valores ocorre dentro da função trocar().

Saída na tela:

```
Valor original: x = 5, y = 10
Valor trocado: x = 10, y = 5
```

Figura 59

8.2 Manipulação de arquivos

Arquivos em C são usados para armazenar dados de forma permanente, ou seja, mesmo após o encerramento do programa, as informações permanecem salvas no disco.

O uso de arquivos permite salvar dados entre execuções do programa, possibilitando diversas aplicações práticas, como:

- Armazenamento de dados para uso posterior.
- Gravação de pontuações, progresso de jogos ou status de atividades.
- Geração de relatórios, logs ou estruturas simples de banco de dados.

- Leitura de dados de entrada sem depender do teclado.
- Processamento de arquivos de texto, como .txt, .csv e arquivos binários.

Para usar arquivos, precisamos primeiro abrir (fopen) e depois fechar os arquivos (fclose).

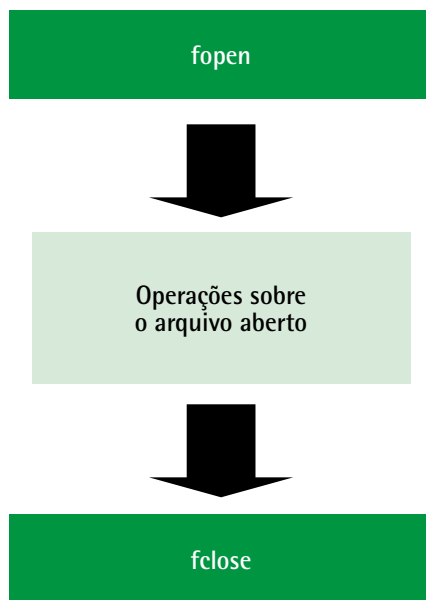


Figura 60

- Usamos fopen, que recebe o modo de abertura do arquivo.
- Usamos fclose para fechar o arquivo.

Quadro 8 – Principais funções de arquivos (stdio.h)

Função	Descrição
fopen()	Abre (ou cria) um arquivo
fclose()	Fecha o arquivo
fprintf()	Escreve dados formatados no arquivo
fscanf()	Lê dados formatados do arquivo
fgetc()	Lê um caractere do arquivo
fputc()	Escreve um caractere no arquivo
fgets()	Lê uma string do arquivo
fputs()	Escreve uma string no arquivo
feof()	Verifica se chegou ao final do arquivo (EOF)

Boas práticas com arquivos:

- Sempre verifique se `fopen()` retornou `NULL`.
- Sempre feche o arquivo com `fclose()`.
- Evite escrever fora do modo adequado (por exemplo, em um arquivo aberto apenas para leitura).
- Use nomes claros para os arquivos e caminhos válidos no sistema operacional.

8.2.1 Leitura e escrita em arquivos

Para escrita/leitura com arquivos, usamos algumas funções semelhantes às aquelas que vimos para escrita/leitura no terminal:

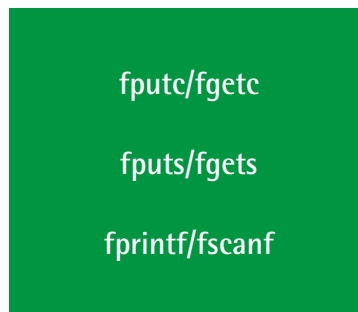


Figura 61

Quadro 9 – Modos de abertura com `fopen()` em C

Modo	Significado	Ação
"r"	Leitura	Abre arquivo existente para leitura
"w"	Escrita	Cria novo arquivo ou apaga o anterior
"a"	Acrescentar (append)	Escreve no final do arquivo sem apagar conteúdo anterior
"r+"	Leitura e escrita	Não apaga conteúdo, mas exige que o arquivo exista
"w+"	Leitura e escrita (limpa tudo)	Cria novo arquivo e apaga tudo, permitindo leitura e escrita
"a+"	Leitura e escrita (append)	Permite ler e escrever no final sem apagar nada

O exemplo a seguir mostra um programa básico de gravação em arquivo usando a função `fopen()` com ponteiros para arquivos. Esse exemplo cria (ou sobrescreve) um arquivo chamado `saida.txt` e escreve nele a frase "Olá, mundo!\n". Por fim, abre o arquivo e mostra a mensagem.

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

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

    FILE *arquivo;

    // Gravação no arquivo
    arquivo = fopen("saida.txt", "w");
    if(arquivo != NULL){
        fprintf(arquivo, "Olá, mundo!\n");
        fclose(arquivo);
    }else{
        printf("Erro ao criar o arquivo.\n");
        return 1;
    }

    // Leitura do conteúdo gravado
    char linha[100];
    arquivo = fopen("saida.txt", "r");
    if(arquivo != NULL){
        printf("Conteúdo do arquivo:\n");
        while(fgets(linha, sizeof(linha), arquivo) != NULL){
            printf("%s", linha);
        }
        fclose(arquivo);
    }else{
        printf("Erro ao abrir o arquivo para leitura.\n");
    }

    return 0;
}
```

Explicação detalhada:

- `FILE *arquivo = fopen("saida.txt","w");`
 - Abre o documento (arquivo) `saida.txt` no modo escrita (`w`).
 - Se o arquivo não existir, ele será criado.
 - Se ele já existir, será apagado e reescrito.
 - `arquivo` é um ponteiro do tipo `FILE` usado para manipulação de arquivos em C.

- **if(arquivo != NULL):** garante que o arquivo foi aberto corretamente. Se `fopen()` retornar `NULL`, houve um erro.
- **fprintf(arquivo,"Olá,mundo!\n");** escreve a frase "Olá, mundo!\n" no arquivo.
- **fclose(arquivo);** fecha o arquivo para garantir que o conteúdo seja salvo corretamente.
- Sempre feche o arquivo após o uso para evitar corrupção de dados.
- **arquivo = fopen("saida.txt","r");**
 - Reabre o arquivo `saida.txt` no modo leitura ("r").
 - Permite ler o conteúdo que foi salvo anteriormente.
- **if(arquivo != NULL)**
 - Verifica se a abertura para leitura foi bem-sucedida.
- **while(fgets(linha,sizeof(linha),arquivo) != NULL)**
 - Lê o conteúdo do arquivo linha por linha com `fgets`.
 - `linha` é um vetor de caracteres que armazena temporariamente o texto lido.
- **printf("%s",linha);**
 - Imprime na tela o conteúdo lido do arquivo.
- **fclose(arquivo);**
 - Fecha o arquivo após a leitura.



Saiba mais

O site a seguir inclui sintaxe, bibliotecas padrão, funções e exemplos. Também é útil para referências da biblioteca padrão C (como `stdio.h`, `stdlib.h` etc.).

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

A seguir, temos um programa básico de leitura em arquivo:

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

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

    FILE *arquivo = fopen("saida.txt", "r");
    char linha[100];

    if(arquivo != NULL){
        while(fgets(linha, 100, arquivo) != NULL){
            printf("%s", linha);
        }
        fclose(arquivo);
    }else{
        printf("Erro ao abrir o arquivo para leitura.\n");
    }

    return 0;
}
```

Explicação:

- `FILE *arquivo = fopen("saida.txt", "r");` abre o arquivo `saida.txt` no modo de leitura ("`r`"). O arquivo é um ponteiro do tipo `FILE`.
- `char linha[100];` cria um vetor de `char` com 100 posições para armazenar cada linha lida do arquivo.

```
while (fgets(linha, 100, arquivo) != NULL) {
    printf("%s", linha);
}
```

- `while` lê linha por linha do arquivo com `fgets()`:
- A variável `linha` é onde será armazenado o texto. 100 é o tamanho máximo da linha (evita estouro).
- `arquivo` é o ponteiro para o arquivo aberto.
- A cada linha lida, imprime no console com `printf("%s", linha);`
- `fclose(arquivo);` fecha o arquivo após a leitura.



Lembrete

Por padrão, C utiliza passagem por valor. Para que uma função modifique a variável original, é necessário usar ponteiros (passagem por referência).

O programa ilustrado a seguir grava informações em um arquivo de texto chamado teste.txt. Ele demonstra como escrever dados em um arquivo usando a função `fprintf()` e o ponteiro `FILE`.

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

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

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

    fprintf(arquivo, "Início do arquivo\n");
    int n = 507;
    fprintf(arquivo, "Valor de n = %d\n", n);

    fclose(arquivo);

    return 0;
}
```

Explicação resumida:

- Cria um arquivo chamado teste.txt e escreve duas linhas nele:
 - "Início do arquivo".
 - "Valor de n = 507".
- Funcionamento:
 - `fopen("teste.txt","w")`:: abre o arquivo para escrita.
 - `fprintf(...)`:: grava texto e valores no arquivo.
 - `fclose(arquivo)`:: fecha o arquivo e salva os dados.

Por fim, o programa a seguir escreve uma mensagem em um arquivo e depois a lê usando ponteiros para manipular o arquivo.

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

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

    FILE *fp = fopen("dados.txt", "w");
    if(fp == NULL) {
        printf("Erro ao abrir o arquivo!\n");
        return 1;
    }

    fprintf(fp, "Olá, usando ponteiros e arquivos!\n");
    fclose(fp);

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

    char c;
    while((c = fgetc(fp)) != EOF) {
        putchar(c);
    }
    fclose(fp);

    return 0;
}
```

Explicação resumida:

- FILE *fp é um ponteiro que gerencia a leitura/escrita no arquivo.
- Abre (ou cria) o arquivo dados.txt no modo de escrita "w".
 - Se o arquivo não abrir corretamente, imprime erro e encerra com return 1.
- Usa fopen, fprintf, fgetc, putchar e fclose para manipular arquivos de texto.
- fprintf(fp,"Olá,usando ponteiros e arquivos!\n");
 - Escreve a mensagem no arquivo.
- fclose(fp);: fecha o arquivo após a escrita.

- `fp = fopen("dados.txt","r");` abre o arquivo no modo leitura "r".
- `fgetc(fp)` lê um caractere do arquivo.
- `putchar(c)` exibe o caractere na tela.
- Esse loop `while` continua até encontrar o fim do arquivo (EOF).



Saiba mais

Existem muitas outras funções e comandos na linguagem C. Para entender melhor, consulte as referências indicadas a seguir:

Manual detalhado de C (não só C++).

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

O Manual do GCC traz a documentação oficial do compilador GNU C.

Disponível em: <https://shre.ink/xzc5>. Acesso em: 25 jun. 2025.

Manual das funções C do Linux (man pages)

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



Resumo

Nesta unidade, estudamos o conceito e a aplicação de funções na linguagem C. Funções são blocos de código reutilizáveis que organizam o programa em partes menores e mais fáceis de manter. Aprendemos a estrutura de uma função (tipo de retorno, nome, parâmetros e corpo), o uso do `return` e a importância do `main()` como ponto de partida da execução.

Exploramos a chamada de funções, incluindo o uso de protótipos e a passagem de parâmetros: por valor (cópia dos dados) e por referência (usando ponteiros). Estudamos também o escopo de variáveis, distinguindo variáveis locais (restritas à função) e globais (acessíveis em todo o programa).

A unidade trouxe diversos exemplos práticos, como funções para calcular quadrado, dobro, área de retângulo, verificar maioria e contar caracteres. Por fim, acentuamos o uso de ponteiros e a manipulação de arquivos, destacando a abertura, a escrita, a leitura e as boas práticas com arquivos em C.



Exercícios

Questão 1. Avalie as afirmativas a seguir sobre o uso de funções na linguagem C.

I – Uma função pode retornar no máximo um valor, utilizando o comando "return". Alternativamente, uma função pode não retornar nenhum valor.

II – A linguagem C não permite o uso de funções recursivas, ou seja, funções que chamam a si mesmas.

III – Na linguagem C, a passagem de argumentos para uma função é sempre feita por referência. Dessa forma, sempre podemos alterar os valores originais das variáveis passadas como argumentos de uma função.

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: uma função pode retornar apenas um valor, utilizando o comando "return". Contudo, esse valor pode ser um ponteiro para uma estrutura de dados complexa. Em alguns casos, essa estrutura pode agrupar diversas variáveis ou múltiplos vetores, significando que, mesmo com essa limitação, podemos fazer com que uma função produza ou altere vários valores, mesmo que esses não possam ser diretamente repassados pelo comando return. Além disso, uma função pode ser do tipo "void", não retornando nenhum valor.

II – Afirmativa incorreta.

Justificativa: podemos definir e utilizar funções recursivas na linguagem C.

III – Afirmativa incorreta.

Justificativa: na linguagem C, funções podem ter argumentos passados de duas formas: por valor ou por referência.

Quando passamos os argumentos por valor, na prática ocorre uma cópia de dados, o que significa que os valores originais não são alterados.

Quando utilizamos a passagem de valores por referência, usamos ponteiros nos argumentos, o que possibilita a modificação de variáveis pertencentes a outros escopos que não o escopo da própria função.

Questão 2. Um dos principais problemas ao trabalhar na escrita de um programa é como organizar o seu código-fonte de forma que ele seja o mais fácil possível de ser lido e compreendido. Essa organização também deve facilitar a manutenção do programa e permitir a sua modificação, com a possibilidade de serem adicionadas novas funcionalidades ao longo do tempo.

Contudo, não queremos apenas adicionar funcionalidades a qualquer custo ou de qualquer maneira, mas da forma mais prática possível, fazendo o menor número possível de alterações. É com essa finalidade que a linguagem C oferece o recurso de funções, o que permite modularizar o código de um programa.

Dessa forma, um problema complexo pode ser dividido em partes menores, mais simples de serem desenvolvidas, compreendidas e, caso seja necessário, modificadas.

Com base no que foi apresentado e nos seus conhecimentos, avalie as afirmativas sobre funções na linguagem C.

I – Uma função em C deve sempre receber pelo menos um argumento.

II – Na linguagem C, variáveis declaradas dentro de uma função têm sempre o escopo global e podem sempre ser utilizadas em outras funções como uma forma de compartilhamento de dados.

III – Na linguagem C, é necessário declarar o tipo associado ao dado que será retornado por uma função (caso a função retorne algum valor). Por exemplo, se uma função é declarada como tendo um tipo de retorno "int", isso significa que ela retorna um valor com o tipo inteiro. Por exemplo, uma função que retorna um número inteiro pode retornar o número 10 ou o número -1, dependendo do seu funcionamento.

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: é possível escrever e utilizar funções que não recebem nenhum argumento. Variáveis declaradas dentro de uma função não têm um escopo global. Os dados armazenados nessas variáveis não são automaticamente compartilhados entre funções.

II – Afirmativa incorreta.

Justificativa: mesma explicação da afirmativa anterior.

III – Afirmativa correta.

Justificativa: devemos informar corretamente o tipo de retorno de uma função, o que permite à linguagem C saber qual é o tipo associado ao dado sendo retornado.

REFERÊNCIAS

BACKES, A. *Estrutura de dados descomplicada: em linguagem C*. São Paulo: Grupo GEN, 2016.

BACKES, A. *Linguagem C: completa e descomplicada*. São Paulo: Grupo GEN, 2018.

CELES, W. *Introdução a estruturas de dados: com técnicas de programação em C*. Rio de Janeiro: Grupo GEN, 2016.

DEITEL, P.; DEITEL, H. C: como programar. 7. ed. São Paulo: Pearson Prentice Hall, 2013.

KERNIGHAN, B. W.; RITCHIE, D. M. *C completo e total*. 2. ed. São Paulo: Pearson Makron Books, 2004.

MANZANO, J. A. N. G.; OLIVEIRA, J. F. *Algoritmos: lógica para desenvolvimento de programação de computadores*. São Paulo: Saraiva, 2016.

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

MIZRAHI, V. V. *Treinamento em linguagem C*. 2. ed. São Paulo: Pearson, 2008.

SOFFNER, R. *Algoritmos e programação em linguagem C*. São Paulo: Saraiva, 2013.

SOUZA, M. A. F. et al. *Algoritmos e lógica de programação: um texto introdutório para a engenharia*. São Paulo: Cengage Learning, 2019.



Informações:
www.sepi.unip.br ou 0800 010 9000