

Bruno Jefferson de Sousa
José Jorge Lima Dias Júnior
Andrei de Araújo Formiga

Introdução a Programação

Editora da UFPB
João Pessoa
2014

Introdução à Programação

Ed. v1.0.2

HISTÓRICO DE REVISÕES

VERSÃO	DATA	MODIFICAÇÕES	NOME
v1.0.2	30/10/2014	cap5 Correção no título do código de troca_main.c; cap5 Remoção de título do código no exercício 4; Revisão geral do Livro; Impressão de mini-sumário por capítulo;	Eduardo
v1.0.1	24/9/2014	cap1: Correção da unidade da altura que estava 1,80 cm; Adição de links no histórico de revisão; Solicitando feedback ao final de cada capítulo;	Eduardo
v1.0.0	11/2013	Primeira versão do livro para impressão.	Bruno, Jorge, Andrei

Sumário

1	Algoritmos	1
1.1	Introdução	1
1.2	O que é um algoritmo?	2
1.3	Características de um algoritmo	4
1.4	Formas de representação	4
1.4.1	Descrição Narrativa	5
1.4.2	Fluxograma	5
1.4.3	Linguagem Algorítmica	6
1.5	Recapitulando	8
1.6	Exercícios Propostos	9
2	Introdução à Programação	11
2.1	Introdução	11
2.2	Níveis das Linguagens de Programação	12
2.3	Tradutores e Interpretadores	13
2.4	Paradigmas de Programação	15
2.5	Linguagem C	16
2.6	Núcleo de um programa	16
2.7	Memória e Variáveis	17
2.7.1	Identificadores	18
2.7.2	Tipos de dados primitivos	19
2.7.3	Declaração de variáveis	21
2.7.4	Constantes simbólicas	22
2.8	Comentários e indentação	23
2.9	Matemática Básica	25
2.10	Entrada e saída de dados	26
2.10.1	Função printf()	27
2.10.2	Função scanf()	28
2.11	Recapitulando	29
2.12	Exercícios Propostos	30

3	Estruturas de Controle	32
3.1	Introdução	32
3.2	Estrutura Sequencial	32
3.3	Estrutura de Decisão	33
3.3.1	Decisão simples	33
3.3.1.1	Expressões lógicas	34
3.3.1.2	Exercício resolvido	36
3.3.1.3	Verificação da condição com expressões aritméticas na Linguagem C	37
3.3.2	Decisão composta	38
3.3.2.1	Exercício resolvido	39
3.3.3	Comando de decisão múltipla	40
3.4	Estrutura de Repetição	42
3.4.1	Comando while	42
3.4.1.1	Exercício resolvido	43
3.4.2	Comando do-while	44
3.4.3	Comando for	45
3.4.4	Laço infinito	46
3.4.5	Exercício Resolvido	47
3.4.6	Comandos de desvio	48
3.4.6.1	Comando break	48
3.4.6.2	Comando continue	49
3.5	Recapitulando	50
3.6	Exercícios Propostos	50
4	Arranjos	52
4.1	Introdução	52
4.2	Vetores	52
4.2.1	Declaração de Vetores	53
4.2.2	Acessando os elementos de um vetor	53
4.2.3	Exercício resolvido	55
4.3	Strings	57
4.3.1	Lendo e imprimindo Strings	57
4.3.2	Manipulando strings	58
4.3.3	Exercício resolvido	59
4.4	Matrizes	59
4.5	Recapitulando	61
4.6	Exercícios Propostos	61

5	Funções	63
5.1	O que são funções?	63
5.1.1	Um exemplo	64
5.2	Parâmetros	67
5.3	Retorno de Valores com Funções	69
5.3.1	Funções, Procedimentos e o Tipo <code>void</code>	70
5.4	Um Exemplo Matemático: Equação de Segundo Grau	73
5.5	Escopo de Variáveis	76
5.5.1	Escopo dos Parâmetros	78
5.5.2	Sombreamento e Sobreposição de Escopos	78
5.6	Passagem de Parâmetros	80
5.6.1	Passagem por Valor	81
5.6.2	Passagem por Referência	82
5.6.2.1	Realizando troca de valores com variáveis globais	83
5.7	Protótipos e Declaração de Funções	85
5.8	Funções Recursivas	87
5.9	Recapitulando	90
5.10	Exercícios Propostos	90
6	Índice Remissivo	92

Prefácio

BAIXANDO A VERSÃO MAIS NOVA DESTE LIVRO

Acesse <https://github.com/edusantana/introducao-a-programacao-livro/releases> para verificar se há uma versão mais nova deste livro (versão atual: **v1.0.2**). Você pode consultar o Histórico de revisões, no início do livro, para verificar o que mudou entre uma versão e outra.

Este livro é destinado a alunos de cursos como Ciência da Computação, Sistemas de Informação, Engenharia da Computação e, sobretudo, Licenciatura em Computação. Ele tem o objetivo de apresentar os principais conceitos da programação de computadores, de modo que sua utilização é mais adequada a disciplinas introdutórias como a de Introdução à Programação. De forma alguma o presente livro tem a pretensão de cobrir todos os assuntos relacionados à área de programação. Sua principal finalidade é a de servir como guia para os alunos que estão dando os primeiros passos nessa área que é tão importante para a ciência da computação.

A disciplina de Introdução à Programação é a primeira de uma sequência de disciplinas que têm o objetivo de tornar os alunos capazes de dominar os fundamentos e as técnicas relacionadas à programação de computadores. Durante o curso de Licenciatura em Computação, especificamente, os alunos terão a chance de conhecer em detalhes algumas das linguagens de programação mais utilizadas atualmente e estarão habilitados a ministrar cursos de programação para diversos públicos.

Mais importante do que conhecer as peculiaridades das linguagens de programação é aprender como funciona a lógica aplicada na elaboração de soluções desenvolvidas a partir de algoritmos computacionais. Sendo assim, o principal objetivo deste livro é ensiná-los a resolver problemas com base nos comandos e mecanismos presentes nas linguagens de programação, desenvolvendo assim o que chamamos de lógica de programação. No decorrer do livro, o leitor aprenderá gradativamente a dar instruções ao computador através de programas que ele próprio será capaz de criar. Para isso, conhecerá a linguagem de programação C, uma das mais utilizadas e mais importantes linguagens na área de Ciência da Computação.

No Capítulo 1 será abordado o conceito de algoritmo, e suas principais formas de representação serão apresentadas. No Capítulo 2, descrevemos todo o processo de tradução de um programa escrito em linguagem de alto nível para um programa equivalente em código de máquina, isto é, a linguagem que os computadores conhecem. Além disso, será apresentado ao leitor a estrutura de um programa na linguagem C, o que o habilitará o leitor a desenvolver seu primeiro programa. O Capítulo 3 consiste no conteúdo do livro que mais desenvolve a lógica de programação. É nele que se encontram as principais instruções de uma linguagem de programação: as estruturas de controle. No Capítulo 4 será apresentado o conceito de arranjos e, por fim, no Capítulo 5, explicaremos como programas complexos podem ser divididos em programas menores, mais fáceis de serem solucionados, através da utilização das funções.

Recomendamos ao aluno, iniciante na programação de computadores, que não se limite à leitura e ao conteúdo deste livro. Pesquise na internet outros materiais, leia outros livros e faça todos os exercícios

propostos. Programação, assim como matemática, requer muito exercício, muita prática. Como mencionado anteriormente, a programação de computadores é uma das subáreas mais importantes da carreira que você escolheu seguir. Boa parte das disciplinas do seu curso depende do conhecimento adquirido em Introdução à Programação. Portanto, dedique o máximo que puder ao aprendizado de uma área que vai permiti-lo transformar sonhos em realidade.

Público alvo

O público alvo desse livro são os alunos de Licenciatura em Computação, na modalidade a distância¹. Ele foi concebido para ser utilizado numa disciplina de *Introdução à Programação*, no primeiro semestre do curso.

Como você deve estudar cada capítulo

- Leia a visão geral do capítulo
- Estude os conteúdos das seções
- Realize as atividades no final do capítulo
- Verifique se você atingiu os objetivos do capítulo

NA SALA DE AULA DO CURSO

- Tire dúvidas e discuta sobre as atividades do livro com outros integrantes do curso
- Leia materiais complementares eventualmente disponibilizados
- Realize as atividades propostas pelo professor da disciplina

Caixas de diálogo

Nesta seção apresentamos as caixas de diálogo que poderão ser utilizadas durante o texto. Confira os significados delas.



Nota

Esta caixa é utilizada para realizar alguma reflexão.



Dica

Esta caixa é utilizada quando desejamos remeter a materiais complementares.

¹ Embora ele tenha sido feito para atender aos alunos da Universidade Federal da Paraíba, o seu uso não se restringe a esta universidade, podendo ser adotado por outras universidades do sistema UAB.

**Importante**

Esta caixa é utilizada para chamar atenção sobre algo importante.

**Cuidado**

Esta caixa é utilizada para alertar sobre algo que exige cautela.

**Atenção**

Esta caixa é utilizada para alertar sobre algo potencialmente perigoso.

Os significados das caixas são apenas uma referência, podendo ser adaptados conforme as intenções dos autores.

Vídeos

Os vídeos são apresentados da seguinte forma:



Figura 1: Como baixar os códigos fontes: <http://youtu.be/Od90rVXJV78>

Nota

Na **versão impressa** irá aparecer uma imagem quadriculada. Isto é o qrcode (http://pt.wikipedia.org/wiki/C%C3%B3digo_QR) contendo o link do vídeo. Caso você tenha um celular com acesso a internet poderá acionar um programa de leitura de qrcode para acessar o vídeo.

Na **versão digital** você poderá assistir o vídeo clicando diretamente sobre o link.

Compreendendo as referências

As referências são apresentadas conforme o elemento que está sendo referenciado:

Referências a capítulos

Prefácio [vi]

Referências a seções

“Como você deve estudar cada capítulo” [vii], “Caixas de diálogo” [vii].

Referências a imagens

Figura 2 [x]



Nota

Na **versão impressa**, o número que aparece entre chaves “[]” corresponde ao número da página onde está o conteúdo referenciado. Na **versão digital** do livro você poderá clicar no link da referência.

Feedback

Você pode contribuir com a atualização e correção deste livro. Ao final de cada capítulo você será convidado a fazê-lo, enviando um feedback como a seguir:

Feedback sobre o capítulo

Você pode contribuir para melhoria dos nossos livros. Encontrou algum erro? Gostaria de submeter uma sugestão ou crítica?



Acesse <https://github.com/edusantana/introducao-a-programacao-livro/issues/new> para realizar seu feedback. Lembre-se de incluir na mensagem a seção, capítulo (**prefacio**) e a versão do livro (**v1.0.2**) alvo de sua contribuição. Você receberá notificações sobre os encaminhamentos que serão dados a partir do seu feedback. Para compreender melhor como feedbacks funcionam consulte o guia do curso.



Nota

A seção sobre o feedback, no guia do curso, pode ser acessado em: <https://github.com/edusantana/guia-geral-ead-computacao-ufpb/blob/master/livro/capitulos/livros-contribuicao.adoc>.



Capítulo 1

Algoritmos

OBJETIVOS DO CAPÍTULO

Ao final deste capítulo você deverá ser capaz de:

- Definir algoritmo
- Descrever suas principais características
- Criar algoritmos utilizando diferentes formas de representação

Sumário

1.1	Introdução	1
1.2	O que é um algoritmo?	2
1.3	Características de um algoritmo	4
1.4	Formas de representação	4
1.4.1	Descrição Narrativa	5
1.4.2	Fluxograma	5
1.4.3	Linguagem Algorítmica	6
1.5	Recapitulando	8
1.6	Exercícios Propostos	9

É comum pensarmos em uma estratégia para executar uma tarefa do nosso dia a dia, mesmo que ela seja muito simples. Ao escovar os dentes, por exemplo, nós seguimos diferentes estratégias. Uns começam com a escovação dos molares e depois partem para os dentes da frente, outros fazem o inverso. Enfim, existem várias formas de escovarmos os dentes, assim como existem várias maneiras de realizarmos diversas atividades. Você sabia que o conjunto de passos para resolver um certo problema ou realizar determinada tarefa chama-se algoritmo? E que eles são importantíssimos para a programação de computadores?

Neste capítulo estudaremos as características dos algoritmos, suas formas de representação e, sobretudo, a relação entre eles e a programação de computadores.

1.1 Introdução

Fonte inspiradora de livros e filmes, o americano Monty Roberts (Figura 1.1 [2]), conhecido como o “encantador de cavalos”, revolucionou a forma de domar cavalos. Ao estudar o comportamento de cavalos selvagens, percebeu que existe entre eles uma linguagem corporal compartilhada. Entendendo tal linguagem, conseguiu rapidamente ganhar a confiança de cavalos arredios e instruí-los a se comportarem como desejava. Além de não usar de violência, essencial no emprego dos métodos convencionais, seu método é capaz de domar cavalos em poucos dias, ao contrário da maioria, que normalmente necessita de várias semanas.



Figura 1.1: Monty Roberts.

Assim como os cavalos, os computadores também são instruídos por meio de uma linguagem particular. Para que eles se comportem como desejamos, basta que sejam comandados a partir de uma linguagem que sejam capazes de entender.

Diferentemente do que ensina o senso comum, os computadores não possuem inteligência. Seu único trabalho é processar dados, conforme uma sequência de instruções que fazem parte do vocabulário da linguagem que eles conseguem compreender. A ilusão de que eles realizam tarefas de forma inteligente é proporcionada através desse conjunto ordenado de instruções, que é denominado de algoritmo. Neste caso, o “domador” do computador, responsável por elaborar o algoritmo que vai orientá-lo na execução de uma determinada tarefa, é chamado de programador de computadores.

1.2 O que é um algoritmo?

A palavra “algoritmo” é derivada do nome Mohammed ibn Musa Al-Khowarizmique, que foi um matemático, astrólogo, astrônomo e autor persa. Ele fez parte de um centro acadêmico conhecido como a Casa da Sabedoria, em Bagdá, por volta de 800 d.C. Seus trabalhos introduziram o cálculo hindu aos árabes e, a partir daí, ao resto da Europa.

Não obstante os algoritmos representam um conceito central na Ciência da Computação, sua atuação não se limita a essa área do conhecimento. Rotineiramente, lidamos com algoritmos na realização das mais variadas tarefas. Eles podem ser utilizados para lavar um carro, preparar um bolo, tomar banho, montar um guarda-roupa, etc. Perceba que os algoritmos não devem ser confundidos com as atividades. Eles se referem aos passos seguidos para que estas sejam realizadas. Como exemplo de algoritmos, podemos citar as instruções para montagem de equipamentos, para utilização de cosméticos como shampoos e condicionadores, para saída de emergência em meios de transporte, receitas culinárias, manuais de uso, entre outros.

A partir do que foi exposto, podemos definir algoritmo como *uma sequência finita, ordenada e não ambígua de passos para solucionar determinado problema ou realizar uma tarefa*.

Na ciência da computação, esse conceito foi formalizado em 1936, por Alan Turing e Alonzo Church, da seguinte forma:

DEFINIÇÃO DE ALGORÍTIMO

Um algoritmo é um conjunto não ambíguo e ordenado de passos executáveis que definem um processo finito.

O exemplo a seguir mostra como pode ser elaborado um algoritmo para realizar uma atividade com a qual lidamos corriqueiramente:

ALGORITMO PARA FRITAR UM OVO

1. Retire o ovo da geladeira.
2. Coloque a frigideira no fogo.
3. Coloque óleo na frigideira.
4. Quebre ovo, separando a casca.
5. Ponha a clara e a gema na frigideira.
6. Espere um minuto.
7. Apague o fogo.
8. Retire o ovo da frigideira.

Agora considere o seguinte problema. Suponha que você dispõe de duas vasilhas de nove e quatro litros respectivamente. Como elas não possuem marcação, não é possível ter medidas intermediárias sobre o volume ocupado. O problema consiste, então, em elaborar uma sequência de passos, por meio da utilização das vasilhas de nove e quatro litros, a fim de encher uma terceira vasilha com seis litros de água. A figura abaixo ilustra dois possíveis passos de um algoritmo para resolver o problema.

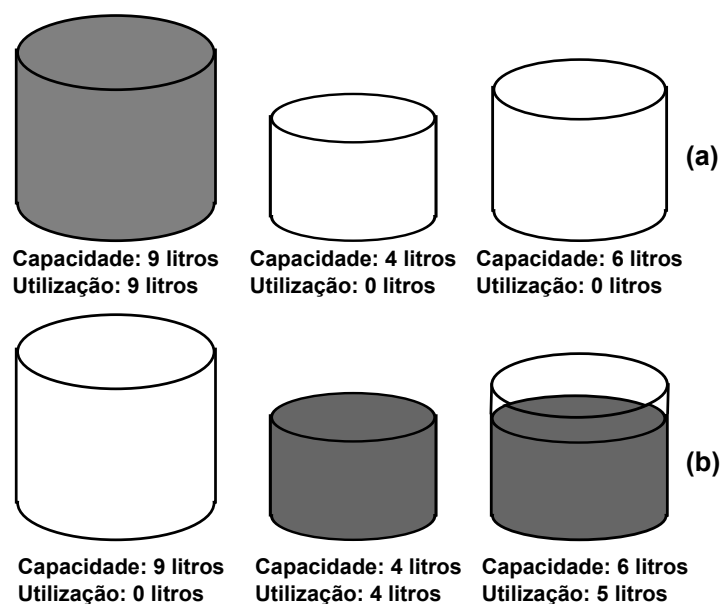


Figura 1.2: Ilustra dois passos possíveis envolvendo as operações de encher e esvaziar as vasilhas. Em (a) apenas a primeira vasilha está cheia. Já em (b) os nove litros da primeira vasilha são colocados nas outras duas.

Uma solução para o problema pode ser alcançada a partir do seguinte algoritmo:

ALGORITMO PARA ENCHER VASILHAS

1. Encha a vasilha de nove litros.
2. Usando a vasilha de nove litros, encha a de quatro.
3. Coloque a quantidade que sobrou (cinco litros) na terceira vasilha ($v3 = 5$).
4. Esvazie a vasilha de quatro litros.
5. Encha novamente a vasilha de nove litros.
6. Usando a vasilha de nove litros, encha a de quatro.
7. Esvazie a de quatro litros.
8. Usando a sobra da de nove litros (cinco litros), encha novamente a de quatro litros.
9. Coloque a sobra da de nove litros (agora um litro) na terceira vasilha ($v3 = 5 + 1 = 6$).

1.3 Características de um algoritmo

Todo algoritmo, seja ele computacional ou não, recebe uma entrada, processa-a e gera uma saída segundo seu conjunto de passos. No caso do algoritmo para fritar ovo, a entrada corresponde à frigideira, ao ovo e ao óleo. O processamento ocorre com a execução de seus passos, gerando como saída o ovo frito.

Os algoritmos computacionais, especificamente, possuem as seguintes características:

Definição

Os passos de um algoritmo devem ser bem definidos, objetivando a clareza e evitando ambiguidades.

Finitude

Um algoritmo deve chegar ao seu fim após um número finito de passos.

Efetividade

Um algoritmo deve ser efetivo, ou seja, suas operações devem ser básicas o suficiente para que possam, em princípio, serem executadas de maneira exata e em um tempo finito.

Entradas

Um algoritmo deve possuir zero ou mais entradas. Estas são insumos ou quantidades que são processados pelos algoritmos durante a execução de seus passos.

Saídas

Um algoritmo deve possuir uma ou mais saídas. Elas representam o resultado do trabalho realizado pelos algoritmos.

1.4 Formas de representação

As formas mais comumente utilizadas para representar algoritmos são as seguintes:

- Descrição narrativa
- Fluxograma
- Linguagem Algorítmica

Todas elas apresentam pontos fortes e fracos, não existindo consenso entre os especialistas sobre a melhor forma de representação. Apresentaremos as nuances de cada uma nas próximas seções.

1.4.1 Descrição Narrativa

Os algoritmos são expressos em linguagem natural (português, inglês, francês, espanhol, etc.). Sua principal desvantagem se encontra no fato da linguagem natural estar bem distante da linguagem utilizada pelos computadores. Logo, a tradução de uma para a outra se torna uma atividade bastante dispendiosa. Além disso, linguagens naturais são mais propensas a ambiguidades. Muitas vezes uma palavra pode ter vários significados, dependendo do contexto no qual são utilizadas. Em contrapartida, é bem mais fácil elaborar um algoritmo por meio de uma linguagem com a qual já temos uma certa familiaridade, do que através de linguagens que não são utilizadas com frequência no dia a dia.

Os exemplos de algoritmos mostrados anteriormente (Algoritmo para fritar um ovo [2] e Algoritmo para encher vasilhas [3]) refletem esta forma de representação.

1.4.2 Fluxograma

Consiste em usar formas geométricas padronizadas para descrever os passos a serem executados pelos algoritmos. As formas apresentadas na Figura 1.3 são as mais comumente utilizadas em fluxogramas.

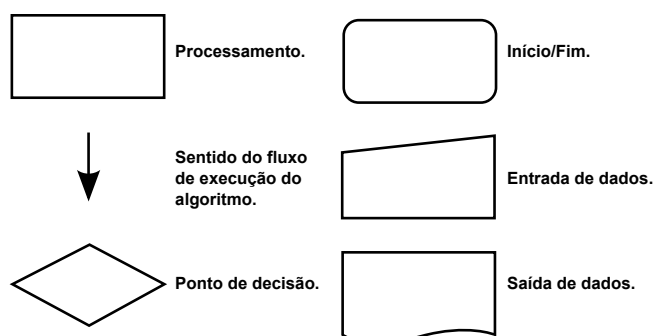


Figura 1.3: Formas geométricas utilizadas em fluxogramas

A vantagem de se fazer uso dos fluxogramas está na facilidade de compreendê-los. Descrições de algoritmos mediante formas gráficas são mais facilmente compreendidas do que descrições que envolvem apenas textos. Além do mais, os fluxogramas possuem um padrão mundial no que se refere à sua simbologia, tornando sua utilização independente das peculiaridades das linguagens naturais.

Para exemplificar o uso de fluxogramas, a Figura 1.4 [6] mostra um algoritmo para calcular a média final de um aluno com base em suas notas e classificá-lo como aprovado ou reprovado. Analisando-a com mais cuidado, é possível perceber que os fluxogramas tendem a crescer bastante quando descrevem algoritmos constituídos de muitos passos, o que dificulta tanto sua construção como sua visualização. Além dessa desvantagem, por impor regras para sua utilização de acordo com cada forma geométrica, há uma limitação no seu poder de expressão, se comparado com a descrição narrativa.

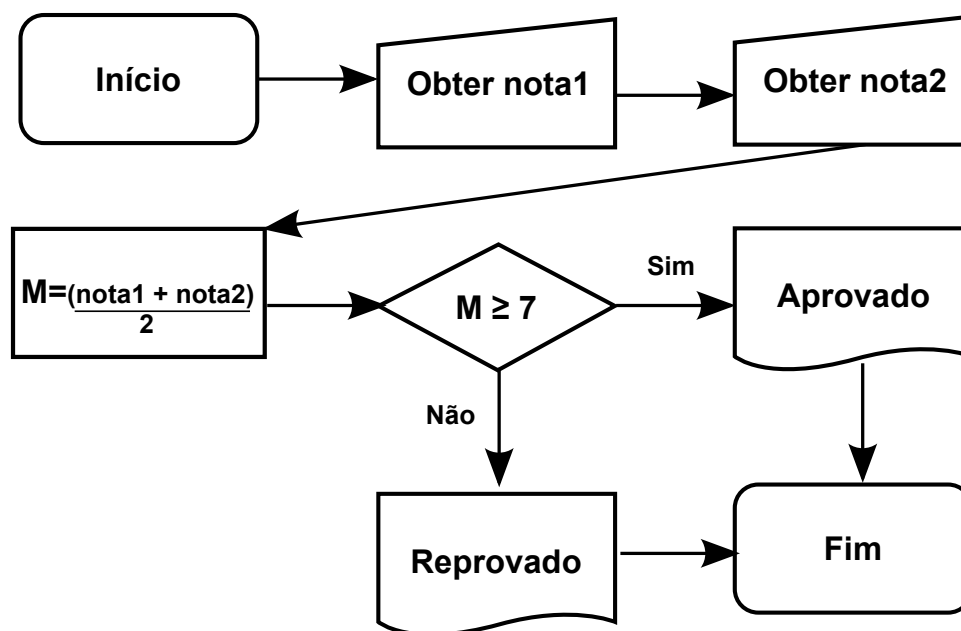


Figura 1.4: Fluxograma para calcular a média de um aluno e classificá-lo

1.4.3 Linguagem Algorítmica

A linguagem que o computador é capaz de compreender tem grande influência na elaboração de algoritmos projetados para ele. Seus passos não podem conter instruções desconhecidas ou fazer referência a símbolos ou expressões que os computadores não conseguem decifrar. Tal linguagem, tantas vezes mencionada neste capítulo, se baseia em conceitos e em arquiteturas de hardware que determinam o funcionamento básico de um computador. Dentre as existentes, a mais utilizada nos computadores atuais é a arquitetura de von Neumann. Seu autor, John Von Neumann (Figura 1.5 [6]), propôs um modelo em que as instruções e os dados ficam juntos na memória.

O processador busca as instruções na memória e as executa uma de cada vez, segundo o seguinte ciclo de execução:

1. Busca instrução;
2. Decodifica instrução;
3. Executa instrução;
4. Volta para o passo 1 para buscar a instrução seguinte na memória.



Figura 1.5: John von Neumann

Para esclarecer como funciona a execução de um algoritmo baseado no ciclo de execução mencionado, considere uma memória com 32 posições para armazenamento, organizada conforme Figura 1.6 [7].

⁰ x = 2	¹ y = 3	² z = x . y	³
⁴	⁵	⁶	⁷
⁸ 2	⁹ 3	¹⁰ 6	¹¹
¹²	¹³	¹⁴	¹⁵
¹⁶	¹⁷	¹⁸	¹⁹
²⁰	²¹	²²	²³
²⁴	²⁵	²⁶	²⁷
²⁸	²⁹	³⁰	³¹

Figura 1.6: Representação de uma memória com 32 posições para armazenamento.

Os números do canto superior direito de cada célula indicam os endereços de memória correspondentes a cada posição da memória representada. Nas três primeiras células constam as instruções que serão executadas e, da oitava à décima, constam valores armazenados nas posições de memória nomeadas por x, y e z, respectivamente.

Supondo que a execução do algoritmo em questão inicia-se com a busca da instrução no endereço 0 (zero), o ciclo de execução continua com a decodificação da instrução $x = 2$, que, após sua realização, resulta no armazenamento do valor 2 na posição de memória de número 8, nomeada de x. O passo 4 então é executado, dando início à busca da próxima instrução. Com isso, a instrução $y = 3$ é encontrada e decodificada, gerando como resultado o armazenamento do valor 3 na posição de número 9, nomeada de y. O mesmo ocorre com a instrução $z = x.y$, que, após sua decodificação, armazena o valor 6 (produto de x por y) na posição de endereço 10 e rotulada de z. O algoritmo em descrição narrativa para a execução das instruções anteriores encontra-se logo abaixo:

ALGORITMO PARA MULTIPLICAR DOIS NÚMEROS

1. Escreva 2 na posição de memória nomeada de x.
2. Escreva 3 na posição de memória nomeada de y.
3. Multiplique x e y e o resultado escreva em z.

Modelos como o mencionado anteriormente não apenas definem a forma como os dados são processados pelo computador, mas também a linguagem que eles são capazes de compreender. Assim sendo, a linguagem utilizada pelos computadores está restrita a um conjunto limitado de instruções, cujo funcionamento depende de sua arquitetura de hardware. As linguagens de programação imperativas (Pascal, C, Cobol etc), por exemplo, foram criadas em função da arquitetura de von Neuman.

A linguagem algorítmica, também chamada de pseudocódigo ou pseudo-linguagem, por sua vez, consiste no emprego de uma linguagem intermediária entre a linguagem natural e uma linguagem de programação. Esse meio termo resulta em uma linguagem que se aproxima das construções de uma linguagem de programação, sem exigir, no entanto, rigidez na definição das regras para utilização de suas instruções. Geralmente, essa forma de representação de algoritmos é uma versão reduzida de linguagens de alto nível como C e Pascal. Segue abaixo o algoritmo da Figura 1.4 [6] em pseudocódigo:

```
1  ALGORITMO
2    DECLARE nota1, nota2, M : NUMÉRICO
3    LEIA nota1
4    LEIA nota2
5    M ← (nota1 + nota2) / 2
6    SE M >= 7.0 ENTÃO
7      ESCRIVA "Aprovado"
8    SENÃO
9      ESCRIVA "Reprovado"
10   FIM-SE
11  FIM_ALGORITMO.
```

As palavras em letras maiúsculas correspondem a palavras reservadas que fazem parte do conjunto de regras que a linguagem algorítmica deve seguir.

Embora sejam mais flexíveis do que as linguagens de programação em relação ao seu uso (a instrução LEIA, por exemplo, muitas vezes é substituída por LER, OBTER, etc.), algumas palavras são necessárias, pois facilitam o entendimento e aproximam o pseudocódigo de um programa de computador. As palavras INÍCIO e FIM, por exemplo, indicam onde começa e termina o algoritmo. Já as instruções LEIA e ESCRIVA referem-se a operações de entrada e saída de dados (ex.: ler dados do teclado ou exibir uma frase no monitor), presentes na maioria das linguagens de programação.

Seguindo com a explicação do algoritmo, perceba que a linha com a instrução $M \leftarrow (nota1 + nota2) / 2$ contém dois símbolos ainda não apresentados. O símbolo / diz respeito à operação aritmética da divisão, ao passo que o símbolo \leftarrow expressa uma operação de atribuição, que pode ser lida da seguinte forma: *A posição de memória, representada simbolicamente por M, recebe o valor da soma de nota1 e nota2, dividido por dois.* Para finalizar, a linha 6 apresenta uma estrutura de controle condicional essencial para as linguagens de programação. Operações de atribuição, expressões e estruturas de controle fazem parte do núcleo das linguagens de programação imperativas e são, portanto, fundamentais para o aprendizado da programação. Todos esses assuntos serão abordados de forma mais aprofundada em capítulos posteriores.

A principal vantagem da forma de representação em linguagem algorítmica está na facilidade com a qual um pseudocódigo pode ser transcrito para uma linguagem de programação. Assim como os

fluxogramas, a desvantagem fica por conta da limitação do seu poder de expressão, devido às regras impostas para a elaboração das instruções.

1.5 Recapitulando

Neste capítulo você estudou algoritmos, suas principais características e suas formas de representação.

Apesar de ser um tema mais abordado na ciência da computação, algoritmos estão presentes nas mais diversas áreas e em várias atividades do cotidiano. Lidamos com eles, por exemplo, quando tomamos banho, cozinhamos, planejamos uma rota para fugirmos do trânsito, consultamos um manual de montagem, enfim, sempre que nos deparamos com um conjunto lógico de passos para realizarmos uma tarefa ou solucionarmos um problema, estamos em contato com algoritmos. É por meio deles que os computadores passam a ilusão de que são inteligentes, realizando tarefas capazes de impressionar qualquer ser humano. No entanto, sabemos que eles apenas processam dados, segundo um conjunto de instruções que lhe são passadas — os algoritmos.

Você viu que os algoritmos computacionais, aqueles elaborados para serem executados em computadores, devem ser claros, ter um número finito de passos, e que estes devem ser simples o suficiente para serem executados de maneira exata e em um tempo finito. Além disso, os algoritmos computacionais devem possuir zero ou mais entradas e uma ou mais saídas.

As formas de representação de algoritmos mais comuns são a linguagem algorítmica, o fluxograma e o pseudocódigo. Da primeira à última há uma aproximação em relação às linguagens de programação, ou seja, o pseudocódigo é a forma de representação que mais se assemelha às linguagens utilizadas na programação de computadores. Na direção inversa, há uma maior liberdade na elaboração de algoritmos, aumentando, assim, a capacidade de expressá-los.

No próximo capítulo abordaremos o processo de tradução de um programa escrito em uma linguagem de alto nível, os paradigmas de programação existentes, e introduziremos os conceitos básicos da programação de computadores. Além disso, você terá o primeiro contato com a linguagem de programação a ser estudada neste livro: a linguagem C.

1.6 Exercícios Propostos

1. Explique, com suas próprias palavras, o que é algoritmo.
 2. Rotineiramente, usamos algoritmos para as mais diversas tarefas. Cite três algoritmos que podemos encontrar no dia a dia.
 3. Em que consiste a característica de efetividade de um algoritmo?
 4. Suponha que o quarto passo de um determinado algoritmo ordene que a execução retorne ao primeiro. Qual característica não está sendo satisfeita por esse algoritmo?
 5. Discorra sobre as formas de representação de algoritmos mais comuns, destacando suas vantagens e desvantagens.
 6. Suponha que você foi premiado com um robô capaz de auxiliá-lo nas tarefas domésticas. Antes que execute determinada atividade, você precisa instruí-lo corretamente através de um algoritmo específico. Sabendo disso, escreva algoritmos, em linguagem natural, para ensiná-lo a realizar cada uma das tarefas abaixo:
-

- a. Trocar a lâmpada do seu quarto.
 - b. Trocar o pneu do seu carro.
 - c. Fazer uma vitamina de banana com açaí.
 - d. Lavar e secar os pratos.
 - e. Calcular quanto você precisa tirar na terceira nota para passar por média em Introdução à Programação.
7. Escreva um algoritmo, utilizando fluxograma, que receba como entrada o peso e altura de uma pessoa, calcule seu IMC (Índice de Massa Corpórea) e exiba sua situação, segundo os seguinte critério:
- Se o $IMC > 25$, a pessoa está acima de seu peso, caso contrário, está abaixo. Onde o $IMC = (Peso)/(Altura^2)$
8. Usando fluxograma, faça um algoritmo que receba como entrada a idade de uma pessoa expressa em anos, meses e dias (Atenção: são 3 entradas) e mostre-a expressa apenas em dias. Considere anos de 365 dias e meses de 30 dias.
9. Considere as instruções armazenadas na memória a seguir:

⁰ x=10	¹ y=5	² z=x/y+z	³
⁴	⁵	⁶	⁷
⁸ 10	⁹ 5	¹⁰ 3	¹¹
¹²	¹³	¹⁴	¹⁵

Considerando que a instrução inicial se encontra no endereço 0 (zero) e as posições 8, 9 e 10 correspondem a x, y e z, respectivamente, explique como funciona a execução das instruções acima, segundo a arquitetura de von Neumann. Antes da execução da instrução de endereço 2 ($z = x/y + z$), a posição de memória referente a z possuía o valor 1 (um).

10. Escreva um algoritmo, em pseudocódigo, que receba como entrada a base e a altura de um triângulo, calcule e exiba sua área.

Feedback sobre o capítulo

Você pode contribuir para melhoria dos nossos livros. Encontrou algum erro? Gostaria de submeter uma sugestão ou crítica?



Acesse <https://github.com/edusantana/introducao-a-programacao-livro/issues/new> para realizar seu feedback. Lembre-se de incluir na mensagem a seção, capítulo (**cap1**) e a versão do livro (**v1.0.2**) alvo de sua contribuição. Você receberá notificações sobre os encaminhamentos que serão dados a partir do seu feedback. Para compreender melhor como feedbacks funcionam consulte o guia do curso.

Capítulo 2

Introdução à Programação

OBJETIVOS DO CAPÍTULO

Ao final deste capítulo você deverá ser capaz de:

- Entender o processo de tradução de programas escritos em linguagem de alto nível para código de máquina
- Compreender o conceito de variável e sua relação com a memória do computador
- Criar instruções que envolvam operações aritméticas
- Utilizar instruções de entrada e saída da linguagem C
- Escrever um programa simples em C

Sumário

2.1	Introdução	11
2.2	Níveis das Linguagens de Programação	12
2.3	Tradutores e Interpretadores	13
2.4	Paradigmas de Programação	15
2.5	Linguagem C	16
2.6	Núcleo de um programa	16
2.7	Memória e Variáveis	17
2.7.1	Identificadores	18
2.7.2	Tipos de dados primitivos	19
2.7.3	Declaração de variáveis	21
2.7.4	Constantes simbólicas	22
2.8	Comentários e indentação	23
2.9	Matemática Básica	25
2.10	Entrada e saída de dados	26
2.10.1	Função printf()	27
2.10.2	Função scanf()	28
2.11	Recapitulando	29
2.12	Exercícios Propostos	30

Agora que você já conhece as características de um algoritmo computacional e é capaz de criar algoritmos utilizando suas diferentes formas de representação, está na hora de você escrever seu primeiro programa. Porém, antes que isso ocorra, você deve aprender alguns conceitos básicos, como o de variável, e conhecer o processo de tradução de um programa escrito em linguagem de alto nível (a linguagem C, por exemplo) para código de máquina, ou seja, para a linguagem que o computador trabalha. Além desses conceitos, este capítulo também aborda a elaborações de expressões aritméticas em C, bem como alguns de seus comandos básicos.

2.1 Introdução

Você viu no capítulo anterior que existe uma linguagem que os computadores são capazes de compreender e que é utilizada na elaboração de algoritmos para instruí-los a executarem as mais diversas tarefas. Essa linguagem é chamada de linguagem de programação e consiste no principal assunto deste capítulo.

Assim como as linguagens naturais (português, inglês, espanhol, etc.), as linguagens de programação têm o objetivo de prover um meio eficaz de comunicação. Elas são constituídas de um conjunto de palavras especiais (vocabulário), que associadas a um conjunto de regras de utilização, determinam como os algoritmos devem ser especificados para que possam ser corretamente decodificados pelo computador.

As linguagens de programação diferem das naturais de várias formas. Primeiramente, apesar de ser possível de serem utilizadas como meio de comunicação entre pessoas, seu principal propósito é possibilitar a comunicação entre uma pessoa e um computador. Além disso, as linguagens naturais são mais tolerantes a erros. Um erro gramatical, por exemplo, não impossibilita uma conversa entre duas pessoas. Já no caso das linguagens de programação, a simples omissão de um ponto e vírgula é capaz até de impedir que a comunicação seja iniciada.

O conteúdo da comunicação por meio de uma linguagem de programação tem um significado especial para a ciência da computação. Enquanto que nos expressamos nas linguagens naturais através de textos e da emissão de sons, nas linguagens de programação nos expressamos através de programas, que nada mais são do que algoritmos escritos em uma linguagem de programação. O estudo das técnicas para elaboração de programas consiste em um dos pilares da ciência da computação, conferindo uma importância particular à disciplina de Introdução à Programação.

Antes que você conheça as peculiaridades de uma linguagem de programação estruturada, como suas principais instruções e regras para a construção de um programa, estudaremos os paradigmas de programação existentes e o processo de tradução de um programa escrito em linguagem de alto nível para um programa em código de máquina.

2.2 Níveis das Linguagens de Programação

Os computadores representam as informações através da manipulação de dois estados. Esse sistema de representação, denominado de sistema binário, decorre do fato da grande maioria dos componentes eletrônicos poder assumir apenas dois valores. Por exemplo, uma lâmpada pode estar no estado "ligado" ou "desligado", um capacitor pode estar "carregado" ou "descarregado" e um circuito elétrico pode estar energizado ou não.

A representação binária utiliza os algarismos "0" e "1", chamados de dígitos binários. Eles são os valores que um bit (menor unidade de informação em um computador) pode assumir e estão associados aos valores de tensão presentes nos circuitos elétricos do computador. Para representar o bit zero, por exemplo, normalmente utiliza-se um valor próximo a zero volts. Para o bit um, utiliza-se um valor um pouco maior, da ordem de poucos volts.

Repare que trabalhar com uma combinação de zeros e uns não é uma tarefa fácil para um ser humano. Para que você perceba a dificuldade, imagine como seria escrever um pseudocódigo substituindo comandos como "LEIA", "ESCREVA" e expressões aritméticas por uma combinação de zeros e uns. O quadro de código binário hipotético abaixo ilustra tal situação, apresentando um algoritmo em pseudocódigo que calcula a média de duas notas lidas da entrada padrão e sua versão hipotética em código binário.

Algoritmo em pseudocódigo

```
ALGORITMO
DECLARE nota1,
        nota2,
        M : NUMÉRICO
LEIA nota1
LEIA nota2
M <= (nota1 + nota2) / 2
FIM_ALGORITMO.
```

Ilustração de um código binário hipotético referente a um algoritmo escrito em pseudocódigo.

```
10100001
10100011 10010001
          10010010
          10010011 11000001
10100100 10010001
10100100 10010010
10010011 11110011 10010001 11110001 10010010 11110010 00000010
10100010
```



Importante

O código binário acima apresentado tem fins meramente didáticos, não sendo elaborado com base em nenhuma máquina real.

Com o intuito de tornar menos complicada, mais eficiente e menos sujeita a erros a tarefa de programar computadores, foram criadas linguagens de programação mais próximas às linguagens naturais. Elas são compostas de um conjunto de palavras-chave, normalmente em inglês, e símbolos que estabelecem os comandos e instruções que podem ser utilizados pelo programador na construção de seus programas.

As linguagens com essa característica são chamadas de linguagens de alto nível, ao passo que as mais próximas da linguagem de máquina (representação binária), são denominadas de linguagens de baixo nível. São exemplos de linguagens de alto nível: Pascal, C (linguagem abordada neste livro), Java, C++ e Python. Como exemplo de linguagem de baixo nível, temos a linguagem Assembly.

2.3 Tradutores e Interpretadores

Se por um lado as linguagens de programação facilitam o trabalho dos programadores, por outro impossibilitam que os programas desenvolvidos nessas linguagens sejam compreendidos pelos computadores, visto que eles são capazes de manipular apenas códigos binários. Dessa forma, os cientistas do passado se depararam com o seguinte desafio: como executar um programa em linguagem de programação, seja ela de baixo ou alto nível, em um computador que trabalha apenas como números binários?

À primeira vista, o desafio em questão parece ser complexo demais para ser solucionado por um iniciante na ciência da computação. Todavia, você vai perceber que a solução para o problema está mais próxima da sua realidade do que imagina. Suponha que você recebeu uma proposta milionária para trabalhar em uma empresa de desenvolvimento de software da China. Seus patrões pagarão suas passagens, hospedagem, transporte e todo o aparato necessário para que você possa ter uma vida tranquila no país asiático. Apesar de a proposta ser atraente, existe um problema: todos os funcionários da empresa falam somente o chinês, inclusive seus patrões. Além disso, o contrato a ser assinado também está escrito em chinês. O que você faria em tal situação?

Recusaria a proposta? É isso mesmo que você está pensando, um tradutor seria a solução para seus problemas.

Do mesmo modo que você precisaria de um tradutor para poder lidar com uma linguagem que não consegue entender, os computadores também necessitam de um tradutor para traduzir um programa escrito em linguagem de programação para um programa correspondente em linguagem de máquina. Dois softwares básicos são responsáveis por realizar a tradução em questão: os tradutores e os interpretadores.

Os tradutores podem ser classificados como montadores e compiladores. Quando o processo de tradução converte um programa que se encontra no nível de linguagem de montagem (representação simbólica da linguagem de máquina, ex.: linguagem Assembly) para a linguagem de máquina, o tradutor utilizado é o montador. Já na tradução de programas em linguagem de alto nível para a linguagem de montagem, o software responsável é o compilador. Perceba que não há tradução direta da linguagem de alto nível para a linguagem de máquina. Para que esta seja alcançada, são necessários vários passos intermediários, sendo um deles a tradução para a linguagem de montagem.

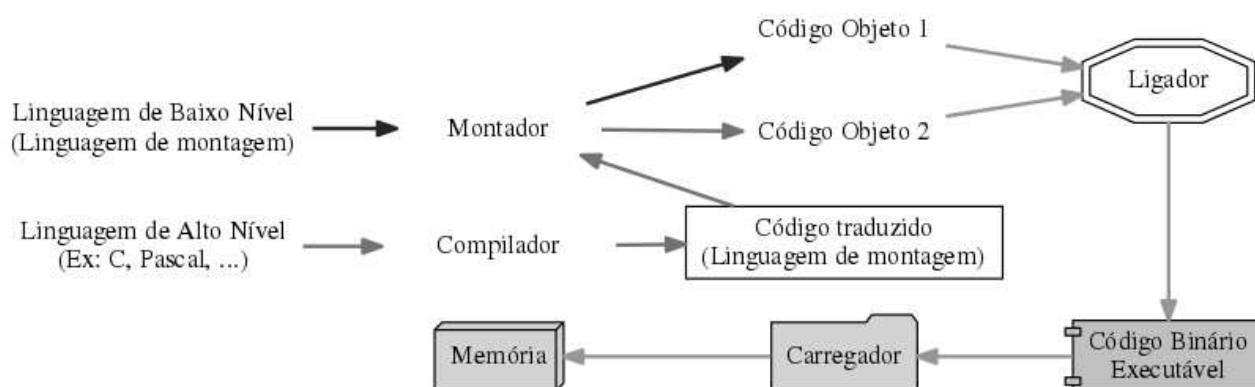


Figura 2.1: Passos no processo de compilação

No processo de compilação, cada parte de um programa (módulo) escrito em linguagem de alto nível é traduzido para um módulo objeto diferente, que consiste em sua representação em linguagem de

montagem. Esse passo no processo de compilação corresponde ao passo 1 da Figura 2.1 [14]. Antes de serem traduzidos para linguagem de máquina pelo montador, é necessário que os vários módulos objetos sejam integrados de modo a formarem um único código. Essa tarefa é realizada no passo 2. O passo 3 é o responsável por carregar o programa na memória, a fim de tornar suas instruções prontas para serem executadas pelo processador.

Os interpretadores, além de realizar a tradução de um programa para a linguagem de máquina, ainda executam suas instruções. Assim que traduz uma instrução, ela é imediatamente executada, gerando assim um ciclo de tradução e execução que prossegue de instrução a instrução até o fim do programa (Figura 2.2 [14]).

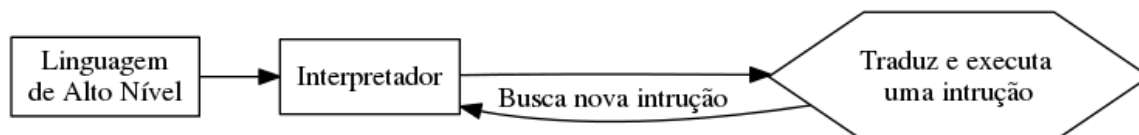


Figura 2.2: Processo de Interpretação

Por não traduzir um programa escrito em linguagem de alto nível diretamente para linguagem de máquina, o processo de compilação tende a ser mais rápido que o processo de interpretação. Além disso, uma vez compilado, um programa pode ser executado várias vezes sem a necessidade de haver uma recompilação. Já na interpretação, cada vez que um programa tiver que ser reexecutado, todo o processo de interpretação deverá ser refeito, independente de ter ocorrido modificações no código fonte do programa desde sua última execução. A vantagem da interpretação fica por conta da possibilidade de testar os programas ao mesmo tempo em que são desenvolvidos.



Importante

A linguagem utilizada neste livro (linguagem C) como ferramenta para iniciá-lo na programação de computadores é uma linguagem compilada, portanto, os programas que você irá desenvolver passarão pelos passos explanados anteriormente.



Nota

Para saber mais sobre o processo de montagem e compilação, leia a Seção 3 do Capítulo 5 do Livro de Introdução ao Computador.

2.4 Paradigmas de Programação

Um paradigma de programação está relacionado com a forma de pensar do programador na construção de soluções para os problemas com os quais se depara. Programar seguindo um determinado paradigma de programação significa representar soluções a partir de uma forma particular de raciocinar na elaboração dos algoritmos. Como os paradigmas mencionados sustentam a atividade de programas, eles influenciam todo o processo de desenvolvimento de software. Alguns dos paradigmas de programação mais utilizados estão relacionados abaixo:

Paradigma imperativo

Representa a computação como ações, enunciados ou comandos que alteram o estado (variáveis) de um programa. Consiste na elaboração de programa a partir de comandos que dizem o que o computador deve fazer a cada momento.

Paradigma estruturado

Soluciona problemas a partir de sua quebra em problemas menores, de mais fácil solução, denominados de sub-rotinas ou subprogramas. Normalmente, o trabalho de cada sub-rotina consiste em receber dados como entrada, processar esses dados e retornar o resultado do processamento para o módulo de software que o executou. Este paradigma ainda defende que todo processamento pode ser realizado pelo uso de três tipos de estruturas: sequencial, condicional e de repetição. É o paradigma adotado neste livro.

Paradigma declarativo

Descreve as características da solução desejada sem especificar como o algoritmo em si deve agir. Em contraste com o paradigma imperativo, que informa ao computador como as instruções devem ser executadas, o paradigma declarativo preocupa-se apenas em definir o que deve ser feito, deixando a cargo de outros softwares decidirem como alcançar a solução descrita. É bastante utilizado no desenvolvimento das páginas web (linguagem html) e na descrição de documentos multimídia através da linguagem Nested Context Language – NCL, adotada pelo padrão brasileiro de TV Digital.

Paradigma orientado a objetos

Enxerga o problema como uma coleção de objetos que se comunicam por meio da troca de mensagens. Os objetos são estruturas de dados que possuem estado (variáveis) e comportamento (lógica).

2.5 Linguagem C

A linguagem C foi desenvolvida por Dennis Ritchie, entre os anos 1971 e 1973, nos laboratórios da AT&T. O objetivo de Ritchie era criar uma linguagem para a implementação de sistemas operacionais e softwares básicos que combinasse a eficiência das linguagens de baixo nível com características das linguagens de alto nível, como legibilidade, portabilidade e manutenibilidade.

A criação da linguagem C é resultado de um processo evolutivo de linguagens, iniciado com uma linguagem chamada BCPL, desenvolvida por Martin Richards. Essa linguagem influenciou a linguagem B, inventada por Ken Thompson, que por sua vez levou ao desenvolvimento de C.

Em 1973, Dennis Ritchie não deixou dúvidas que seu objetivo foi alcançado, desenvolvendo eficientemente parte do sistema Unix na linguagem C. A partir de meados dos anos 80, C começou a ganhar popularidade e, devido à sua flexibilidade em atuar com características de linguagens de alto e baixo nível, foi reconhecida como uma linguagem de propósito geral, sendo utilizada na implementação de uma grande variedade de sistemas.

Devido à importância auferida na área da programação de computadores, C é hoje uma das linguagens mais utilizadas em cursos de programação do mundo inteiro. Sendo assim, ela é a linguagem que guiará você na compreensão das nuances da arte de programar e servirá como ferramenta para elaboração dos seus primeiros programas. A linguagem C será apresentada, de forma conveniente, a partir da próxima seção.

2.6 Núcleo de um programa

A organização da sequência de instruções em um programa obedece a um conjunto de regras estabelecidas pela linguagem de programação. Um programa em C é estruturado em funções, que são, basicamente, trechos de código que podem ser chamados várias vezes para realizar uma certa tarefa. Assim, todas as instruções pertencentes a um programa em C devem estar contidas em uma função.

Além de ser um meio de agrupar trechos de um código, uma função em programação tem características semelhantes a uma função matemática, no sentido de que recebe parâmetros como entrada (seria o domínio da função) e retorna um valor como saída (imagem).

Em C existe uma função especial, denominada de `main` (principal), que determina o início e o fim da execução de um programa. De forma mais específica, a execução de um programa tem seu início com a execução da primeira instrução da função `main` e termina com a execução da sua última instrução. Dessa maneira, todo programa em C deve possuir tal função.



Importante

Você conhecerá mais sobre as funções no Capítulo 6 deste livro. Por enquanto é necessário apenas que você saiba que todo programa em C deve ter uma função `main` e que a execução de um programa inicia e termina com a execução de seus comandos.

Iniciaremos nosso estudo com um programa extremamente simples, que apenas imprime uma mensagem na tela:

```
#include <stdio.h>

int main() {
    printf("Meu primeiro programa!");
    return 0;
}
```

Analisemos o que ocorre em cada linha de código:

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

Esta linha de código solicita ao compilador que inclua no programa a biblioteca padrão para comandos de entrada e saída da linguagem C. Uma biblioteca consiste em um conjunto de arquivos que contém funções que podem ser incorporadas a outros programas. Neste caso, a inclusão da biblioteca `stdio.h` permite que o programa utilize suas funções para ler dados da entrada padrão (teclado) e para escrever dados na saída padrão (tela).

```
int main() {
```

Com esta linha de código definimos a função `main` e demarcamos o seu início com o caractere `{` (abre-chaves). Todo conteúdo de uma função em C fica delimitado por chaves `({})`.

```
printf("Meu primeiro programa!");
```

O programa tem seu início com a execução desta instrução, uma vez que ela é a primeira instrução da função `main`.

A função `printf` tem a finalidade de escrever na tela os dados recebidos por parâmetro. Como resultado de sua execução, neste caso, será exibida a frase "Meu primeiro programa!" no canto superior esquerdo do monitor do computador.

O ponto-e-vírgula no fim da instrução serve para separar esta instrução da próxima, dessa maneira, cada instrução deve terminar com sua utilização.

```
return 0;
```

Essa instrução encerra a execução do programa, de modo que deve ser sempre a última da função `main` (antes do `fecha-chaves`, é claro). O número 0 (zero) serve para indicar ao sistema operacional que o programa terminou com sucesso (números diferentes de zero indicariam um erro). Você entenderá melhor como isso funciona quando abordarmos detalhadamente as funções, no capítulo 6.

2.7 Memória e Variáveis

A memória principal do computador ou memória RAM (Figura 2.3 [18]) é constituída por componentes eletrônicos capazes de armazenar dados. Cada dígito binário (0 ou 1) ocupa uma porção de memória chamada de bit, e um conjunto de 8 bits é denominado de byte. A memória é dividida em células de memória de um byte de tamanho, que podem ser acessadas a partir de um número único que as identifica de forma particular. Esse número é chamado de endereço e tem a mesma função que os endereços de nossas casas, que é identificar de forma única nossas residências, a fim de possibilitar o envio e o recebimento de correspondências. No caso do computador, as correspondências são os dados que serão armazenados nas células de memória.



Figura 2.3: Figure 2.3: Memória RAM.

Uma variável em programação representa, através de símbolos, o conteúdo de uma célula ou posição de memória. Por exemplo, se uma variável de nome `x` possui o valor 10, significa dizer que a posição de memória, representada pelo símbolo `x`, armazena o valor 10. Em programação, podemos enxergar a memória como um conjunto de posições que possuem um endereço e uma representação simbólica (variável), como ilustrado na Figura 2.4 [18].

Endereço	Variável	Valor
0	x	100
1	y	20,5
2	z	aula
...
n	var	-1700,23

Figura 2.4: Representação da memória em função dos endereços, das posições de memória e das variáveis.

Nota



As variáveis podem ter nomes diversos, desde símbolos comuns na matemática, como é o caso das variáveis `x`, `y` e `z`, até nomes como `var`, `endereco`, `cpf`, etc. As regras para dar nome às variáveis serão apresentadas na próxima seção. Perceba também que os valores que as variáveis podem armazenar não se limitam apenas a valores numéricos inteiros. Elas podem armazenar, por exemplo, um conjunto de caracteres, como é o caso da variável `z`, e valores fracionários, como é o caso das variáveis `y` e `var`.

2.7.1 Identificadores

Os nomes que damos às variáveis, rotinas, constantes e demais componentes num programa escrito numa dada linguagem de programação são chamados de identificadores. Na seção anterior, por exemplo, utilizamos os identificadores `x`, `y`, `z` e `var` para dar nome às variáveis da Figura 2.4 [18]. As palavras que possuem significado especial nas linguagens de programação, como é o caso dos nomes dados às estruturas de controle (`for`, `while`, `if`, etc.), tipos de variáveis, dentre outros, são chamadas de palavras-chave.

As regras básicas para formação de identificadores são:

- Os caracteres utilizados são os números, letras maiúsculas, minúsculas e o caractere especial sublinha (`_`);
- O primeiro caractere deve ser uma letra ou o sublinha;
- Não são permitidos espaços em branco;
- Palavras reservadas não podem ser utilizadas como identificadores.

Abaixo, alguns exemplos de identificadores válidos:

```
B  
b  
X2  
computacao  
COMPUTACAO  
nota1  
nota_2  
cpf  
RG
```

Identificadores inválidos:

```
3B -> Não pode começar com número.  
X 2 -> Não pode conter espaço em branco.  
Computação -> Não é permitido utilizar o caractere cedilha.  
COMPUTACÃO -> Caracteres especiais como o til (~) não são permitidos.  
while -> while é uma palavra reservada.  
function -> function também é uma palavra reservada.
```

Uma boa prática de programação é escolher nomes que indiquem a função de uma variável, como por exemplo: `soma`, `ano`, `idade`, `media`, `dataNascimento`, `numero_filhos`, `nota1`, `nota2`, `notaFinal`, `salario`, etc. Também é uma prática bastante difundida iniciar os identificadores com letras minúsculas e usar letras maiúsculas ou sublinha para separar palavras. Por exemplo, para escolher um identificador para uma variável que deve armazenar a data de nascimento de uma pessoa, as duas opções citadas correspondem à `dataNascimento` e `data_nascimento`, respectivamente.



Importante

A linguagem C faz distinção entre letras maiúsculas e minúsculas, sendo assim, variáveis de nomes `var` e `Var` são consideradas como duas variáveis diferentes.

2.7.2 Tipos de dados primitivos

Vimos anteriormente que as variáveis podem armazenar valores de diversos tipos, tais como números inteiros, fracionários e um conjunto de caracteres. Os tipos de dados ou tipo de variáveis são representados de forma diferente em cada linguagem de programação, algumas dando suporte a mais tipos que outras. Embora haja certa variação de uma linguagem para outra, a maioria delas dá suporte a um grupo de tipos básicos, incorporados na própria linguagem, chamados de tipos primitivos. Em C há a possibilidade da criação, por parte do programador, de tipos particulares, denominados de tipos derivados. Estudaremos as formas de definirmos tipos derivados no Capítulo 5.

Existem três tipos primitivos na linguagem C: números inteiros, números de ponto flutuante (números fracionários) e caracteres. Os números fracionários são chamados de números de ponto flutuante devido à forma como eles são armazenados no computador. Portanto, sempre que você ouvir o termo ponto flutuante, tenha em mente que o tipo de dados em questão diz respeito aos números fracionários.

Os tipos de dados primitivos em C estão descritos na tabela abaixo:

Tabela 2.1: Tipos primitivos da linguagem C

Tipo	Tamanho (em bytes)	Função
int	4	Armazenar um número inteiro.
float	4	Armazenar números de ponto flutuante.
double	8	Armazenar números de ponto flutuante com maior precisão.
char	1	Armazenar um caractere.

Como as variáveis de tipos primitivos distintos são representadas na memória de formas diferentes, elas exigem uma quantidade de bytes distinta para seu armazenamento. Uma variável do tipo **int**, por exemplo, ocupa normalmente quatro bytes na memória, ao passo que uma variável do tipo **char** ocupa apenas 1 (um) byte.

É importante salientar que o tipo **char** na linguagem C, diferentemente de outras linguagens, pode também armazenar números inteiros que requerem apenas um byte de memória. O que ocorre é que há uma correspondência entre um caractere e um número inteiro, conforme uma tabela padrão. Por exemplo, quando atribuímos a variáveis do tipo **char** valores como a, b e c, na verdade estamos atribuindo os valores inteiros 97, 98 e 99. Os números inteiros que correspondem aos caracteres estão todos listados em uma tabela padrão, conhecida como tabela ASCII.

O tipo **int** pode ainda ser qualificado de acordo com as seguintes palavras-chave:

short ou long

se referem ao tamanho das variáveis;

signed ou unsigned

indicam, respectivamente, se as variáveis do tipo **int** poderão ser positivas e negativas (com sinal) ou apenas positivas (sem sinal) .

A qualificação de tipo é realizada quando os qualificadores são antepostos aos tipos. Por exemplo, uma variável do tipo **unsigned long int** armazena inteiros positivos de tamanhos grandes, enquanto que uma variável do tipo **signed short int** armazena inteiros positivos e negativos de tamanhos menores.

A tabela a seguir ilustra os valores que normalmente podem ser armazenados nas variáveis do tipo **int** e diversas de suas variações.

Tabela 2.2: Intervalos de valores de tipos inteiros utilizados por grande parte dos compiladores de C.

Tipo	Tamanho (em bytes)	Valores que podem ser armazenados
int	4	-2^{31} a $2^{31} - 1$
short int	2	-2^{15} a $2^{15} - 1$
long int	4	-2^{31} a $2^{31} - 1$
unsigned int	4	0 a $2^{32} - 1$
unsigned short int	2	0 a $2^{16} - 1$
unsigned long int	4	0 a $2^{32} - 1$
signed char	1	-2^7 a $2^7 - 1$

Tabela 2.2: (continued)

Tipo	Tamanho (em bytes)	Valores que podem ser armazenados
unsigned char	1	0 a $2^8 - 1$
long long int	8	-2^{63} a $2^{63} - 1$
unsigned long long int	8	0 a $2^{64} - 1$

**Nota**

Os tamanhos e valores presentes nas tabelas anteriores podem variar de compilador para compilador. Desse modo, eles servem apenas como um guia de referência para de norteá-lo na escolha dos tipos adequados aos programas que você desenvolverá.

2.7.3 Declaração de variáveis

Cada variável utilizada na elaboração de um programa precisa ser definida com antecedência. Para isso, o programador precisa definir o identificador da variável e o seu tipo por meio do que chamamos de declaração de variáveis. Sua forma geral é a seguinte:

```
tipo_da_variável identificador;
```

O exemplo a seguir declara, na linguagem C, as variáveis `x` e `y` como sendo do tipo **int**.

```
int x, y;
```

A declaração de variáveis, além de estabelecer uma interpretação sobre os bits armazenados na memória, também é responsável por alocar espaço para armazenamento desses bits. No exemplo anterior, a declaração das variáveis `x` e `y` resulta na alocação de 4 bytes (provavelmente) para cada uma delas, bem como determina que os bits a serem armazenados nos espaços alocados deverão ser interpretados como números inteiros. Seguem abaixo alguns exemplos de declarações de variáveis:

```
int idade;
int numeroFilhos;
int dia, mes, ano;
float altura;
float nota, media;
```

Os tipos das variáveis tem uma relação muito próxima com a função que elas exercem em um programa. Caso precisemos armazenar e realizar cálculos sobre a idade de alguém, deveremos declarar a variável `idade` como **int**, visto que a idade corresponde a um número inteiro positivo. Do mesmo modo, como sabemos que a altura de uma pessoa é um número fracionário (ex.: 1,80 m), devemos declarar a variável `altura` como sendo do tipo **float**. Variáveis do mesmo tipo podem ser declaradas em uma mesma linha, sendo separadas por vírgulas, como na declaração das variáveis `dia`, `mes` e `ano` do exemplo anterior. Já variáveis de tipos diferentes devem ser declaradas obrigatoriamente de forma separada.

Um programa elaborado com base no paradigma estruturado pode ser visto como uma sequência de transições de estado do início até o fim de sua execução. Se pudéssemos tirar uma "foto" da execução de um programa em determinado momento, o que observaríamos seria o conjunto de suas variáveis

e os valores nelas armazenados no momento, isto é, o estado do programa. Se os programas podem mudar de estado, então deve existir um comando nas linguagens de programação que permitam alterar o conteúdo de uma variável. Tal comando é denominado de **atribuição**, e o exemplo a seguir mostra como ele é utilizado em pseudocódigo.

```
idade ← 18
```

Essa instrução deve ser lida como "a variável `idade` recebe o valor 18 (dezoito)". Em C, o comando correspondente é:

```
idade = 18;
```

**Nota**

Podemos utilizar o comando de atribuição no momento da declaração de uma variável. Esse procedimento é chamado de inicialização de variáveis.

Embora C e outras linguagens de programação utilizem o operador aritmético da igualdade para representar a atribuição, as semânticas de ambos não podem ser confundidas. Por exemplo, em matemática a equação $x=x+1$, onde x pertence aos Reais, nunca pode ser satisfeita, visto que um número real não poder ser igual a ele próprio mais um. Já em programação, o comando `x =x + 1` quer dizer que a variável x irá receber o conteúdo armazenado nela própria mais um. Supondo que x possuía conteúdo igual a 10 antes da execução da atribuição em questão, após sua execução x seria igual a 11 ($x =10 + 1 =11$). A tabela abaixo ilustra o cenário apresentado.

Comando	Valor atual de x
<code>int x;</code>	Indefinido
<code>x =10;</code>	10
<code>x =x + 1;</code>	11

Essa forma de atribuição é um artifício bastante empregado na programação, sendo denominado de **incremento**.

2.7.4 Constantes simbólicas

Muitas vezes declaramos algumas variáveis que não devem ser modificadas durante a execução de um programa. É o caso das variáveis abaixo:

```
PI = 3.14159;  
ACELERACAO_GRAVIDADE = 9.8;  
VELOCIDADE_LUZ = 300000;
```

Não faz sentido alterar o valor de uma variável que representa a aceleração da gravidade, por exemplo, pois o valor da constante gravitacional, como seu próprio nome já diz, permanece sempre o mesmo. Para casos como esse é preferível que usemos **constantes simbólicas** no lugar de variáveis. A linguagem C permite que um identificador seja associado a uma constante através da diretiva `#define`, cuja sintaxe é descrita abaixo:

```
#define nome_constante valor_constante;
```

Dessa forma, a definição da constante `PI` mencionada acima poderia ser realizada através da linha de código:

```
#define PI = 3.14159;
```

Quando o programa que contém essa instrução é compilado, o compilador substitui todas as ocorrências de `PI` pelo seu valor associado.

Outra utilidade proveniente da definição de constantes diz respeito à facilidade de modificação de um programa. Imagine que você desenvolveu um programa para o registro contábil de uma locadora de veículos e que em vários trechos de código você usou o valor da diária de locação para realizar diversos cálculos. Suponha agora que o dono da locadora aumentou o valor da diária de locação e que você foi chamado para modificar o programa a fim de adequá-lo ao novo valor. Dessa forma, você terá que alterar cada ocorrência contendo o valor antigo e o seu trabalho será proporcional ao número de ocorrências desse valor. Utilizando constantes simbólicas, você precisaria apenas alterar a definição da constante, conforme sugere o quadro abaixo:

```
#define VALOR_LOCACAO 80.0  
#define VALOR_LOCACAO 100.0
```

Apesar das regras para definição dos nomes das constantes simbólicas serem as mesmas daquelas utilizadas para identificadores, é uma boa prática de programação defini-las com letras maiúsculas, separando as palavras que as compõem, se houverem, pelo caractere sublinha (`_`).

2.8 Comentários e indentação

À medida que um programa cresce, ele vai ficando cada vez mais difícil de ser lido e consequentemente de ser entendido. É comum um programador ter grandes dificuldades para compreender seus próprios programas após passar alguns dias sem trabalhar em suas linhas de código. Por isso, algumas medidas devem ser tomadas no sentido de preparar um código-fonte legível. Existem várias formas de aprimorar a legibilidade de um programa, contudo nos restringiremos aos comentários e à indentação.

Explicar o código-fonte em linguagem natural é uma estratégia óbvia para facilitar sua compreensão. Esse é o papel dos comentários em uma linguagem de programação. Em C, qualquer sequência de caracteres localizada entre os delimitadores `/*` e `*/` é um comentário. Por exemplo:

```
z = x + y; /* z é o resultado da soma entre x e y. */
```

A explicação da linha de código acima, embora desnecessária devido à simplicidade da instrução, é um comentário na linguagem C. Outras linguagens podem usar delimitadores distintos para expressar os comentários. A linguagem C ainda possui o delimitador `//`, muitas vezes chamado de delimitador de comentário de linha. Os caracteres colocados à sua frente e na mesma linha em que ele se encontra são considerados comentários e ignorados pelo compilador. Exemplo de sua utilização:

```
int idade; // Variável inteira para representar  
           // a idade de uma pessoa.
```

Perceba que para comentar mais de uma linha com o delimitador `//`, precisamos utilizá-lo em cada linha que se deseja comentar. Dessa maneira, quando se deseja comentar mais de uma linha é mais adequado o uso dos delimitadores `/*` e `*/`.

Outra forma de tornar um programa mais legível é organizar as instruções de modo a refletir a hierarquia entre elas. Por exemplo:

```
#include <stdio.h>

/* Código não-indentado */
int main() {
int x, y, z;

x = 10;
y = 2;
z = x / y;

if (x > 5) {
printf("x é maior que cinco.");
}

return 0;
}
```

Repare que a declaração de variáveis, os comandos de atribuição e os comandos `if` (apresentado no Capítulo 3) e `return` estão todos dentro da função `main`. Dizemos então que eles são hierarquicamente subordinados à função referida. Da mesma forma, o comando `printf` está subordinado ao comando `if`. Uma maneira de destacar a hierarquização das instruções é alinhar os comandos com o mesmo nível de hierarquia (inserindo espaços nas instruções de nível inferior), o que chamamos de **indentação**. Para que você visualize o resultado do processo de indentação, considere o código do exemplo anterior depois de corretamente indentado:

```
#include <stdio.h>

/* Código indentado */
int main() {
    int x, y, z;

    x = 10;
    y = 2;
    z = x / y;

    if (x > 5) {
        printf("x é maior que cinco.");
    }

    return 0;
}
```

2.9 Matemática Básica

O computador foi criado com o intuito inicial de realizar contas. Portanto, é importante que saibamos como instruí-lo a computar as operações aritméticas básicas. E essa não vai ser uma tarefa difícil, já que as expressões aritméticas em programação são bastante semelhantes às expressões utilizadas na matemática. Em C, os operadores matemáticos utilizados são os seguintes:

Tabela 2.3: Operadores aritméticos

Operador	Operação
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
%	Resto da divisão

A utilização dos operadores em C ocorrem da forma com a qual estamos acostumados: colocamos um operador entre dois operandos e vamos construindo as expressões. À medida que as expressões vão ficando mais complexas, podemos utilizar os parênteses para agrupar operadores e operandos. Diferentemente do que ocorre na matemática, em C não se utilizam colchetes e chaves para o agrupamento de expressões que já estão entre parênteses. Estes devem ser os substitutos dos primeiros quando houver necessidade. Por exemplo, a expressão matemática $\frac{(x+y)-(a+b)}{2}$, em C se tornaria:

```
( (x+y) - (a+b) ) / 2
```

Veja alguns exemplos de como os operadores aritméticos devem ser usados em C (os resultados são apresentados ao lado de cada operação):

```
x = 4 * 5; // 20
x = x / 2; // 10
y = x % 4; // 2
z = x * y - 5; // 15
z = x * (y - 5); // -30
z = ((2 + 3) * 4 - 2) / 2; // 9
```

A precedência dos operadores aritméticos, isto é, a ordem em que eles são avaliados, pode ser alterada do mesmo modo que o fazemos quando trabalhamos com expressões na matemática: utilizamos os parênteses para que algumas operações sejam realizadas antes que outras. É o que ocorre na expressão acima na expressão $z = x * (y - 5)$. Caso os parênteses fossem omitidos¹, a primeira operação a ser realizada seria a que multiplica x por y , depois, do seu resultado seria subtraído cinco. Com a inserção dos parênteses, ocorre primeiro a subtração para depois ser realizada a multiplicação.

A linguagem C possui ainda operadores especiais que resultam da combinação de operadores aritméticos com operadores de atribuição. São eles:

Tabela 2.4: Operadores aritméticos de atribuição e operadores de incremento.

Operador	Operação equivalente
$x += y$	$x = x + y$
$x -= y$	$x = x - y$
$x *= y$	$x = x * y$
$x /= y$	$x = x / y$
$x \% = y$	$x = x \% y$
$x++$	$x = x + 1$

¹ Expressão resultante: $z = x * y - 5$

Tabela 2.4: (continued)

Operador	Operação equivalente
<code>++x</code>	<code>x = x + 1</code>
<code>x--</code>	<code>x = x - 1</code>
<code>--x</code>	<code>x = x - 1</code>

Os primeiros cinco operadores são denominados de **operadores aritméticos de atribuição**, ao passo que os quatro últimos são chamados de **operadores de incremento**.

Aqui cabe destacar as diferenças entre os operadores de incremento quanto à localização dos operadores aritméticos. Considere os exemplos a seguir:

```
x = 0;
y = 6;
z = 2;
(a) x = y / ++z; // incremento antes
// y = 6, z = 3, x = 2
```

```
x = 0;
y = 6;
z = 2;

(b) x = y / z++; // incremento depois
// y = 6, z = 3, x = 3
```

Nos exemplos apresentados, temos dois algoritmos que se diferenciam apenas pela utilização dos operadores de incremento de adição. Na expressão (a), a variável `y` é dividida por `++z`, enquanto que na expressão (b) ela é dividida por `z++`. A diferença é sutil, mas é determinante no resultado da expressão. No primeiro caso, `z` é incrementada antes da divisão, logo $x = 6 \div (2 + 1) = 6 \div 3 = 2$. Na expressão (b), `z` é incrementado depois da divisão, o que resulta em $x = 6 \div 2 = 3$. Repare que em ambos os casos o valor de `z` é incrementado, de modo que após as instruções (a) e (b) o valor de `z` é igual a 3.

2.10 Entrada e saída de dados

Imagine que você desenvolveu um programa para controlar suas finanças pessoais e com ele intenciona conter seus gastos e ainda guardar uma parte do que ganha na poupança. Esse programa necessita de interação? Como você informará suas receitas e despesas? Como ele apresentará o resultado dos seus cálculos com a finalidade de auxiliá-lo no controle de suas finanças?

As respostas de todas essas perguntas convergem para a seguinte conclusão: um programa de computador é praticamente inútil se não apresentar algum tipo de interação com o usuário. No cenário anterior, por exemplo, você precisa informar ao programa quais são as suas receitas e despesas. Além disso, é necessário que ele o deixe a par dos resultados dos seus cálculos, caso contrário ele não terá serventia alguma.

Os mecanismos que as linguagens de programação oferecem para interação com o usuário estão presentes em suas bibliotecas de entrada e saída. Em C, as funções responsáveis pelas operações básicas de entrada e saída se encontram na biblioteca **stdio**, que é utilizada por meio da diretiva:

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

Vimos na seção 2.6 uma forma de exibir na tela uma sequência de caracteres através da função `printf()`, que, além de imprimir caracteres, também é capaz de exibir o conteúdo de variáveis de diversos tipos. Os detalhes de sua utilização, bem como uma função similar para entrada de dados são apresentados nas seções posteriores.

2.10.1 Função `printf()`

A função de saída `printf()` permite que dados sejam escritos na saída padrão, que normalmente é a tela do computador. Uma chamada da função `printf` tem o seguinte formato:

```
int printf(string_de_formato, arg1, arg2, ..., argn)
```

Isso quer dizer que a função `printf` irá escrever na saída padrão os argumentos `arg1, arg2, ..., argn` de acordo com o que está especificado no parâmetro `string_de_formato`. Além disso, o tipo **int** indica que a função retorna um número inteiro, que neste caso corresponde ao número de caracteres impressos. O exemplo a seguir ilustra a utilização da função `printf()`:

```
#include <stdio.h>

int main() {
    int idade;
    float altura;
    idade = 18;
    altura = 1.90;

    printf("Tenho %d anos e %.2f de altura.", idade, altura);

    return 0;
}
```

Após a execução do código acima será exibida na tela a seguinte frase:

Tenho 18 anos e 1.90 de altura.

Os caracteres `%d` e `%.2f` são denominados de especificadores de formato e têm o objetivo de definir o formato das variáveis que serão escritas na saída padrão. De outro modo, podemos entendê-los como "guardadores de lugar" para as variáveis a serem exibidas. No exemplo acima, no lugar do `%d` será colocada a primeira variável passada por parâmetro (`idade`) e no lugar do `%.2f` a segunda variável (`altura`). Além disso, elas deverão ser dos tipos `int` e `float`, respectivamente. O ponto seguido de um número antes do código de formato indica a quantidade de casas decimais a serem exibidas (quando aplicados a variáveis do tipo ponto-flutuante) e são denominados de especificadores de precisão. No exemplo anterior eles foram os responsáveis pela exibição da variável `altura` com duas casas decimais.

A tabela abaixo lista os especificadores de formato mais comuns utilizados na função `printf()`.

Tabela 2.5: Especificadores de formatos mais utilizados na função `printf()`

Código	Formato
<code>%d</code> ou <code>%i</code>	Inteiro (int) decimal
<code>%ld</code> ou <code>%li</code>	Inteiro (long int) decimal
<code>%u</code>	Inteiro sem sinal
<code>%c</code>	Caractere
<code>%s</code>	Cadeira de caracteres
<code>%f</code>	Número de ponto-flutuante

2.10.2 Função `scanf()`

A função de entrada `scanf()` possibilita a leitura de dados da entrada padrão, ou seja, do teclado. O que ela faz é interromper a execução do programa até que o usuário digite algo e depois pressione a tecla Enter. Depois que o programa retoma sua execução, o conteúdo digitado é armazenado em uma ou mais variáveis. Uma chamada da função `scanf` tem o seguinte formato:

```
int scanf(string_de_formato, arg1, arg2, ..., argn)
```

O parâmetro `string_de_formato` especifica os tipos de dados que serão lidos e os parâmetros `arg`, `arg2`, ..., `argn` correspondem aos endereços das variáveis nas quais serão armazenados os valores digitados pelo usuário. A função `scanf()` retorna um valor inteiro que indica o número de variáveis que tiveram valores atribuídos, sendo utilizado para verificar algum problema na entrada de dados. O exemplo a seguir ilustra a utilização da função `scanf()`:

```
#include <stdio.h>

int main() {
    int idade;
    float altura;

    printf("Informe sua idade: ");
    scanf("%d", &idade)
    printf("Informe sua altura: ");
    scanf("%f", &altura);

    printf("\nVocê tem %d anos e %.2f de altura.", idade, altura);

    return 0;
}
```

Ao contrário do exemplo da seção anterior, os dados a serem exibidos não estão pré-determinados, isto é, as variáveis não possuem valores a priori. As atribuições apenas ocorrem quando o usuário entra com valores via teclado. No exemplo, depois que a sequência de caracteres "Informe sua idade:" é exibida, a execução do programa é interrompida até que o usuário digite um valor. Quando isso ocorre, ele é armazenado no endereço da variável `idade`, obtido quando ela é precedida pelo caractere `&`. Por conseguinte, o `printf()` ao final do código irá exibir os dados informados pelo usuário e não os dados pré-determinados pelo programador.

**Importante**

A tecla Enter também possui um caractere que a representa, a saber, o caractere especial '\n'. Portanto, quando o '\n' é escrito na saída padrão, o efeito gerado é o mesmo da digitação da tecla Enter.

A função `scanf()` também pode ler numa mesma linha diversos dados, armazenando-os em diferentes variáveis. As leituras do código anterior, por exemplo, podem ser reescritas da seguinte forma:

```
printf("Informe sua idade e sua altura:");  
scanf("%d %.2f", &idade)
```

Para que esse código funcione como desejado, o usuário precisa digitar um número inteiro seguido de um espaço e depois um número de ponto-flutuante. O espaço é requerido porque ele é utilizado na especificação do formato (entre o `%d` e o `%.2f` há um espaço). Assim como `printf()`, a função `scanf()` também possui uma lista de especificadores de formato. Os mais utilizados seguem abaixo:

Tabela 2.6: Especificadores de formatos mais utilizados da função `scanf()`

Código	Significado
<code>%d</code> ou <code>%i</code>	Leitura de um inteiro (int) decimal
<code>%ld</code> ou <code>%li</code>	Leitura de um inteiro (long int) decimal
<code>%u</code>	Leitura de um inteiro sem sinal
<code>%c</code>	Leitura de um único caractere
<code>%s</code>	Leitura de uma cadeia de caracteres
<code>%f</code>	Leitura de um número de ponto-flutuante

2.11 Recapitulando

Neste capítulo você pôde conhecer um pouco mais sobre o funcionamento dos computadores, mais especificamente sobre a forma com a qual eles decodificam as instruções que lhes são passadas. Vimos, portanto, que as linguagens de programação podem ser classificadas em linguagens de alto nível ou baixo nível, de acordo com sua proximidade em relação à linguagem que os computadores podem compreender: o código de máquina.

Como programar em linguagem de baixo nível é uma tarefa árdua, foram criadas as linguagens de alto nível para facilitar a vida dos programadores. Desse modo, surgiu a necessidade de um software que fosse capaz de realizar a tradução de programas escritos em linguagem de alto nível para programas equivalente em código de máquina. Esses softwares são os tradutores e interpretadores.

Aprendemos que a maneira de pensar de um programador na resolução de um problema está relacionada com um paradigma de programação. Se um programador utilizar a linguagem C, por exemplo, ele vai raciocinar segundo os paradigmas imperativo e estruturado.

Falando em linguagem C, conhecemos a estrutura de um programa escrito nessa linguagem, seus tipos primitivos, como podem ser elaborados nomes para as variáveis e como estas podem ser declaradas. Enfim, você deu os primeiros passos para a elaboração do seu primeiro programa.

No próximo capítulo você vai estudar as estruturas de controle em C. Elas simplesmente são as instruções mais importantes para o desenvolvimento da lógica de programação. Por isso, estude atentamente o próximo capítulo e tente elaborar o maior número de programas possível para consolidar o aprendizado.

2.12 Exercícios Propostos

1. Diferencie linguagem de programação de alto nível de linguagem de programação de baixo nível. Dê exemplos de linguagens que se enquadram em ambos os tipos.
 2. Qual é o principal objetivo dos tradutores e interpretadores?
 3. Defina montador e compilador, enfatizando suas diferenças.
 4. Explique como funcionam os interpretadores.
 5. Quais as vantagens da compilação em relação à interpretação?
 6. O que é um paradigma de programação? Cite exemplos.
 7. Quais dos seguintes itens não podem ser utilizados como identificadores na linguagem C? Explique por quê?
 - a. 3x
 - b. Inflação
 - c. COMPUTACAO
 - d. nota_1
 - e. nota 2
 - f. prof.
 - g. \$4
 - h. RG
 - i. main
 - j. return
 8. Descreva os tipos primitivos de C, destacando os valores que eles podem armazenar.
 9. Qual a diferença entre os operadores prefixo e sufixo de incremento?
 10. Qual é valor de $(x1 + x2)$ após a execução dos grupos de comandos abaixo:
 - a. `y = 6;`
 - b. `z = 8;`
 - c. `c = 2;`
 - d. `x1 = ((y * z) - z) / c;`
 - e. `x2 = (z / 2) / y++;`
-

11. Considerando as atribuições $x = 20$ e $y = 2$, calcule o resultado de cada uma das expressões abaixo:
- a. $(x-- + x * (x \% y))$
 - b. $(x-- + x * (x \% y))$
 - c. $(x-- + x * (x \% 3))$
 - d. $(--x + x * (x \% 3))$
 - e. $(--x + x * (x \% x))$
12. Faça um programa em C que solicite ao usuário que digite o ano de seu nascimento, armazene o valor digitado em uma variável e em seguida imprima na saída padrão a sua idade.

Feedback sobre o capítulo

Você pode contribuir para melhoria dos nossos livros. Encontrou algum erro? Gostaria de submeter uma sugestão ou crítica?



Acesse <https://github.com/edusantana/introducao-a-programacao-livro/issues/new> para realizar seu feedback. Lembre-se de incluir na mensagem a seção, capítulo (**cap2**) e a versão do livro (**v1.0.2**) alvo de sua contribuição. Você receberá notificações sobre os encaminhamentos que serão dados a partir do seu feedback. Para compreender melhor como feedbacks funcionam consulte o guia do curso.

Capítulo 3

Estruturas de Controle

OBJETIVOS DO CAPÍTULO

Ao final deste capítulo você deverá ser capaz de:

- Entender as estruturas sequenciais, de seleção e de repetição;
- Escrever estruturas de seleção utilizando os comandos if, if-else e switch da linguagem C;
- Escrever estruturas de repetição utilizando os comandos for, while e do-while da linguagem C.

Sumário

3.1	Introdução	32
3.2	Estrutura Sequencial	32
3.3	Estrutura de Decisão	33
3.3.1	Decisão simples	33
3.3.2	Decisão composta	38
3.3.3	Comando de decisão múltipla	40
3.4	Estrutura de Repetição	42
3.4.1	Comando while	42
3.4.2	Comando do-while	44
3.4.3	Comando for	45
3.4.4	Laço infinito	46
3.4.5	Exercício Resolvido	47
3.4.6	Comandos de desvio	48
3.5	Recapitulando	50
3.6	Exercícios Propostos	50

3.1 Introdução

Vimos no Capítulo 1 [1] que os algoritmos são instruções que contêm passos para solucionar um determinado problema. Vimos também que estes algoritmos podem ser representados através de linguagens de programação, como por exemplo, a linguagem C, que estamos aprendendo aqui. Estes passos são executados na sequência que eles aparecem. Entretanto, em muitas situações, é necessário alterar o fluxo de execução destas instruções. Pode ser que seja necessário executar um passo, ou um conjunto deles, apenas se uma determinada condição `for` verdadeira, ou talvez, pode ser que seja preciso repetir um conjunto de passos várias vezes até uma determinada condição. Neste sentido, este capítulo irá explicar as diferentes estruturas de controle existentes nos algoritmos e seus respectivos comandos na linguagem C.

3.2 Estrutura Sequencial

Um algoritmo que possui uma estrutura sequencial significa que suas instruções são executadas na sequência em que elas aparecem, sem nenhuma alteração no seu fluxo, a não ser, claro, que exista alguma instrução explícita para a mudança deste fluxo. Vejamos o código em C na Figura 3.1 [33] abaixo.

```
void main() {  
    int x, y, soma;  
    scanf(&x) ;  
    scanf(&y) ;  
    soma = x + y;  
    printf("%d", soma) ;  
}
```

Execução das
Instruções




Figura 3.1: Estrutura sequencial na linguagem C

Este algoritmo irá ler dois valores e guardá-los, respectivamente, nas variáveis `x` e `y`. Após isso, a variável inteira `soma` receberá a soma dos valores de `x` e `y`. Em seguida, será mostrada na saída padrão, o resultado desta soma. Perceba que os passos do algoritmo são executados de cima para baixo.

Entretanto, em alguns momentos os problemas que queremos resolver requerem a alteração no fluxo normal de execução do algoritmo. Na próxima seção, iremos aprender como executar um conjunto de instruções de acordo com uma determinada condição.

3.3 Estrutura de Decisão

Como foi dito, muitas vezes é necessário criar blocos de instruções no algoritmo que são executados apenas se uma determinada condição `for` verdadeira. Veja o algoritmo abaixo:

```
"Se hoje não chover, então João irá à praia".
```

No algoritmo acima, João irá à praia se, e somente se, não chover hoje. Significa que esta instrução de João ir à praia só será executada se a condição de não chover for verdadeira. Este tipo de estrutura é chamado de **estrutura de decisão**, também conhecido como estrutura de seleção ou condicional.

Podemos ter três tipos de estrutura de decisão: decisão simples, decisão composta e decisão múltipla. Vamos ver adiante estes três tipos e quais são os comandos na linguagem C, respectivamente, para cada um destes tipos.

3.3.1 Decisão simples

Quando queremos que uma determinada instrução ou um conjunto de instruções execute apenas se uma determinada condição for verdadeira. A estrutura da decisão simples é a seguinte:

```
SE condição ENTÃO instrução
```

Uma condição deve ter como resultado apenas dois valores possíveis: verdadeiro ou falso. A instrução só será executada se a condição tiver o valor verdadeiro.

Vamos analisar o exemplo a seguir. Este algoritmo lê um valor digitado pelo usuário e armazena na variável *x*. Em seguida, o comando *SE* verifica se o valor de *x* é menor que 20. Caso seja, a instrução *ESCREVA* é executada, mostrando na tela a frase "o valor de X é menor que 20".

Exemplo 3.1 Algoritmo que comparava valor lido com 20

```
LEIA x
SE x < 20 ENTÃO
ESCREVA "O valor de x é menor que 20."
```

Para cada linguagem de programação há uma sintaxe para criar estruturas de controle. Na Linguagem C, a estrutura de decisão simples possui a seguinte forma:

```
if (expressão)
    instrução;
```

Na linguagem C, a condição é definida como uma expressão, que pode ser lógica ou aritmética. Ao ser executada, a expressão é verificada. Se o resultado desta for verdadeiro, a instrução que vem após a expressão é executada.

Entretanto, muitas vezes queremos que mais de uma instrução seja executada caso a condição seja verdadeira. Neste caso, devemos utilizar um bloco de instruções dentro do comando *if*, como é mostrado abaixo.

Estrutura de decisão simples com blocos na linguagem C

```
if (expressão) {
    instrução 1;
    instrução 2;
    ...
}
```

As duas formas, com e sem bloco, se diferenciam apenas pelo fato de que a primeira possui apenas uma instrução a ser executada caso a condição seja verdadeira. No segundo caso, um bloco de instruções será executado. Na linguagem C, um bloco de instruções deve estar entre chaves.

**Importante**

Sempre que você precisar executar um bloco de instruções, utilize as chaves para delimitar o início e o fim deste bloco.

Como foi dito, a expressão pode conter uma expressão lógica ou aritmética. As expressões aritméticas foram vistas no capítulo 2. Vamos ver agora como funcionam as expressões lógicas na linguagem C.

3.3.1.1 Expressões lógicas

As expressões lógicas são usualmente utilizadas para fazer comparações entre operandos. Para isso, estas expressões são compostas por operadores lógicos e relacionais, e possuem apenas dois valores possíveis: verdadeiro ou falso. Por exemplo, quando queremos saber se um valor é maior, menor, igual ou diferente de um outro valor.

Na linguagem C, os seguintes operadores relacionais podem ser utilizados:

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

Considere o código `menorq20.c` [35] abaixo. Este reflete o mesmo algoritmo de Exemplo 3.1 [34], sendo que agora, implementado na linguagem C. Na linha 6, estamos pedindo para o usuário entrar com o valor de `x`. Na linha 8, temos um comando `if`, onde há uma expressão relacional `x < 20`. Portanto, essa expressão é verificada e caso seja verdadeira, será mostrado na saída padrão "O valor de `x` é menor que a 20." (linha 9). Caso a expressão seja falsa, o algoritmo se encerra sem mostrar nada na saída padrão, pois a instrução após o comando `if` não é executada.

Código fonte `code/cap3/menorq20.c`

menorq20.c

```
1  #include <stdio.h>
2
3  int main() {
4      int x;
5
6      scanf("%d", &x);
7
8      if (x < 20)
9          printf("O valor de x e' menor que 20.");
10
11     return 0;
12 }
```

Em outros casos, necessitamos utilizar operadores lógicos nas expressões para avaliar mais de uma expressão relacional. Por exemplo, digamos que no problema acima queremos verificar se o valor digitado para a variável `x` está dentro do intervalo entre 10 e 20. Neste caso, precisamos que a

condição verifique as duas expressões relacionais: $(x > 10)$ e $(x < 20)$. Portanto, precisamos conectar as duas expressões relacionais utilizando um operador lógico E. A tabela abaixo apresenta os operadores lógicos possíveis:

Operador em C	Operador em linguagem algorítmica
&&	E
	OU
!	NÃO

Portanto, o algoritmo para resolver o problema acima, na linguagem C, é o seguinte:

Código fonte code/cap3/entre10e20.c

entre10e20.c

```
1 #include <stdio.h>
2
3 int main() {
4     int x;
5
6     scanf("%d", &x);
7
8     if (x > 10 && x < 20)
9         printf("x esta' entre 10 e 20.");
10
11     return 0;
12 }
```

Perceba que agora a condição do comando 'if' possui duas expressões relacionais conectadas por um operador lógico E (&&). Nesse caso, se ambas as expressões forem verdadeiras, será mostrada na saída padrão "x esta' entre 10 e 20". Caso alguma das expressões seja falsa, nada será mostrado, pois o resultado da expressão completa é falso. Vejamos a tabela abaixo, denominada de tabela verdade, que mostra a relação lógica entre duas expressões e seus respectivos resultados.

Expressão 1	Operador	Expressão 2	Resultado
Verdadeiro	E	Verdadeiro	Verdadeiro
Verdadeiro	E	Falso	Falso
Falso	E	Falso	Falso
Verdadeiro	OU	Verdadeiro	Verdadeiro
Verdadeiro	OU	Falso	Verdadeiro
Falso	OU	Falso	Falso
Verdadeiro	NÃO	-	Falso
Falso	NÃO	-	Verdadeiro

3.3.1.2 Exercício resolvido

ER 3.1. Considere quatro variáveis a, b, c e d com valores iniciais de 5, 7, 3 e 9. Dada as condições abaixo, indique se o resultado final da expressão será verdadeiro ou falso.

- a. $(a != 3 \ || \ b < 10 \ || \ c == 5)$

- b. $(d > 8 \ \&\& \ c == 3 \ || \ a >= 10)$
- c. $!(d == 12 \ \&\& \ a != 10)$
- d. $(c == 4 \ || \ d <= 6) \ \&\& \ (a >= 5 \ \&\& \ b != 9) \ || \ (!(a < 5))$

Resposta:

- a. Neste caso temos três expressões lógicas. A primeira $(a != 3)$ é verdadeira. A segunda $(b < 10)$ é verdadeira, e a terceira $(c == 5)$ é falsa. Como as expressões estão conectadas por um operador OU ($||$), então basta que uma das expressões seja verdadeira para o resultado da expressão completa ser verdadeira.
- b. Temos neste caso três expressões. Para um melhor entendimento, vamos utilizar uma tabela. As duas primeiras expressões $(d > 8 \ \text{e} \ c == 3)$ são verdadeiras e estão conectadas pelo operador lógico $\&\&$, logo $R1 \ \&\& \ R2$ é verdadeiro. A terceira expressão $(a >= 10)$, por sua vez, é falsa. Então, resolvendo $R3 \ || \ R4$, temos o resultado final como verdadeiro.

Rs	Expressão	Resultado
R1	$d > 8$	VERDADEIRO
R2	$c == 3$	VERDADEIRO
R3	$R1 \ \&\& \ R2$	VERDADEIRO
R4	$a >= 10$	FALSO
R5	$R3 \ \ R4$	VERDADEIRO

- c. Utilizando novamente a tabela, temos que a primeira expressão $(d == 12)$ é falsa. A segunda expressão $(a != 10)$ verdadeira. A relação entre $R1 \ \&\& \ R2$ é falsa, pois apenas $R2$ é verdadeira. A última expressão é uma negação de $R3$, ou seja, se $R3$ é falso, então $R4$ é verdadeiro.

R	Expressão	Resultado
R1	$d == 12$	FALSO
R2	$a != 10$	VERDADEIRO
R3	$R1 \ \&\& \ R2$	FALSO
R4	$!R3$	VERDADEIRO

- d. Vamos utilizar novamente a tabela para nos auxiliar. Temos que prestar bastante atenção nos parênteses das expressões que podem ser utilizadas para explicitar a precedência da avaliação.

Rs	Expressão	Resultado
R1	$c == 4$	FALSO
R2	$d <= 6$	FALSO
R3	$R1 \ \ R2$	FALSO
R4	$a >= 5$	VERDADEIRO
R5	$b != 9$	VERDADEIRO
R6	$R4 \ \&\& \ R5$	VERDADEIRO
R7	$a < 5$	FALSO
R8	$!R7$	VERDADEIRO
R9	$R3 \ \&\& \ R6$	FALSO
R10	$R9 \ \ R8$	VERDADEIRO

3.3.1.3 Verificação da condição com expressões aritméticas na Linguagem C

Anteriormente, dissemos que a expressão dentro de um comando `if` pode ser lógica ou aritmética. Vimos como funciona nos casos de expressões lógicas. Nos casos de expressões aritméticas, na linguagem C, *Falso* assume o valor zero, e *Verdadeiro* assume qualquer valor diferente de zero. Neste sentido, quando utilizamos uma expressão aritmética dentro da condição de um comando `if` para verificar se esta é verdadeira ou falsa, temos que ter o cuidado de analisar o valor resultante. Vamos verificar o exemplo no código abaixo.

Código fonte code/cap3/if5.c

if5.c

```
1  #include <stdio.h>
2
3  int main() {
4      int x = 5;
5
6      if (x)
7          printf("Isto sera' mostrado");
8
9      if (x - 5)
10         printf("Isto nao sera' mostrado");
11
12     return 0;
13 }
```

Inicialmente a variável inteira `x` recebe o valor 5 (linha 4). Na linha 6 existe uma estrutura de decisão simples, onde há a verificação da expressão que está entre parênteses. Nesse caso, a expressão é apenas a própria variável `x`, logo o resultado da expressão é o valor desta, que é 5. Considerando o que foi dito, quando o resultado for **diferente de zero**, ele é considerado **verdadeiro**. Logo, o resultado da expressão também é verdadeiro, e então a instrução que vem após a condição é executada.

Já na linha 9, também há outra estrutura de decisão simples, na qual a condição a ser avaliada é a expressão `x - 5`. O resultado dessa expressão é **zero**, fazendo com seja avaliada como **falsa**. Consequentemente, a instrução que vem após a condição não é executada.

3.3.2 Decisão composta

Em alguns momentos, ao termos uma estrutura de decisão, queremos que uma outra instrução ou um outro bloco de instruções seja executado caso a condição de decisão seja falsa. Esta estrutura é chamada de decisão composta.

O pseudocódigo abaixo exemplifica uma estrutura de decisão composta.

Exemplo 3.2 Pseudocódigo com decisão composta

```
LEIA nota
SE nota >= 7 ENTÃO
    ESCREVA "Aprovado"
SENÃO
    ESCREVA "Reprovado"
```

Na linguagem C, utilizamos a palavra `else`, após a instrução ou bloco de instruções do `if`, para definir que queremos executar um outro conjunto de instruções, caso a expressão condicional seja falsa.

```
if (expressão)
    instrução 1;
else
    instrução 2;
```

Quando possuímos apenas uma instrução a ser executada, não precisamos utilizar o delimitador de bloco de instruções (as chaves). Neste caso, a condição é verificada. Caso seja positiva, a instrução 1 é executada, caso contrário, a instrução 2 é executada. Caso queiramos que um conjunto de instruções seja executado, devemos utilizar então as chaves, como mostra o código abaixo.

```
if (condição) {
    instrução 1;
    instrução 2;
    ...
} else {
    instrução 3;
    instrução 4;
    ...
}
```

Neste caso temos dois blocos de instruções: um para o bloco do `if`, que será executado caso a condição seja verdadeira, e o bloco de instruções do `else`, caso a condição seja falsa.

**Importante**

Tabule as instruções que estão dentro dos blocos, colocando-os mais a direita (utilize a tecla TAB do teclado). Esta organização do código chama-se **indentação**. Dessa maneira, seu código se torna mais legível, e conseqüentemente mais fácil de encontrar possíveis erros.

Vamos traduzir o pseudocódigo apresentado no Exemplo 3.2 [38] para a linguagem C.

Código fonte code/cap3/nota7.c

nota7.c

```
1  #include <stdio.h>
2
3  int main() {
4      float nota;
5
6      scanf("%f", &nota);
7
8      if (nota >= 7)
9          printf("Aprovado");
10     else
11         printf("Reprovado");
12
13     return 0;
14 }
```

3.3.2.1 Exercício resolvido

ER 3.2. Escreva um programa para ler 2 números inteiros do teclado (A e B), verificar e imprimir qual deles é o maior, ou a mensagem "A=B", caso sejam iguais.

Resposta:

Código fonte code/cap3/comparaab.c

comparaab.c

```
1  #include <stdio.h>
2
3  int main() {
4      int a, b;
5
6      scanf("%i", &a);
7      scanf("%i", &b);
8
9      if (a > b)
10         printf("A e' maior que B.");
11     else if (b > a)
12         printf("B e' maior que A.");
13     else
14         printf("A = B");
15
16     return 0;
17 }
```

3.3.3 Comando de decisão múltipla

Uma outra forma de escrever uma estrutura de condição é utilizando o comando de decisão múltipla `switch`. Este tipo de estrutura condicional tem a mesma função do `if-else-if`, com a diferença que o comando `switch` não aceita expressões, apenas constantes. A vantagem de se utilizar este comando é a legibilidade do código quando conhecemos os possíveis valores para uma determinada variável. Vamos ver o formato de uso do comando `switch`.

```
switch (variável) {
    case VALOR1:
        instrução1;
        instrução2;
        break;
    case VALOR2:
        instrução3;
        instrução4;
        break;
    default:
        instrução5;
        instrução6;
        break;
}
```

Uma variável do tipo `char` ou `int` é colocada entre parênteses após o comando `switch`. Os valores desta variável que serão avaliados logo em seguida, através das declarações `case`. Para cada

possível valor da variável, existe uma declaração `case` correspondente. Caso o valor seja aquele que corresponde na declaração `case`, então as instruções abaixo dela serão executadas até encontrar o comando `break`. A declaração `default` é opcional, e é executada apenas se a variável não for igual a nenhuma das constantes definidas nas declarações `case`.

Importante



O comando `break` tem a função de interromper um determinado fluxo de execução. Este comando será melhor explicado na seção 3.4.5 que fala sobre os comandos de desvio. O importante a saber por hora é que o comando `break` deve ser utilizado ao final das instruções de cada declaração `case`. Caso não seja colocado, as instruções das outras declarações `case` também serão executadas.

Para um melhor entendimento, vamos analisar o código `semana.c` [41] abaixo. A ideia deste programa é que o usuário digite o valor numérico correspondente ao dia da semana e o programa mostre por extenso este dia. Uma variável chamada `semana` do tipo `int` é declarada (linha 4) e guardará o valor que o usuário irá digitar (linha 7). Em seguida, o comando `switch` foi utilizado (linha 9). Para cada dia da semana existe uma declaração `case` correspondente. Isto significa que se o usuário digitou o valor 1, a instrução da linha 11 será executada, mostrando na saída a string "Domingo". Caso o valor digitado seja 2, a instrução da linha 14 é executada, mostrando na saída a string "Segunda-feira". A mesma ideia acontece para os outros 5 dias da semana. Caso o valor digitado pelo usuário não esteja entre 1 e 7, as instruções da declaração `default` serão executadas, mostrando a string "Numero fora do intervalo permitido."

Código fonte `code/cap3/semana.c`

`semana.c`

```
1  #include <stdio.h>
2
3  int main() {
4      int semana;
5
6      printf("Digite um numero de 1 a 7: ");
7      scanf("%d", &semana);
8
9      switch (semana) {
10         case 1:
11             printf("Domingo");
12             break;
13         case 2:
14             printf("Segunda-feira");
15             break;
16         case 3:
17             printf("Terca-feira");
18             break;
19         case 4:
20             printf("Quarta-feira");
21             break;
22         case 5:
23             printf("Quinta-feira");
24             break;
25         case 6:
```

```
26         printf("Sexta-feira");
27         break;
28     case 7:
29         printf("Sabado");
30         break;
31     default:
32         printf("Numero fora do intervalo permitido.");
33         break;
34 }
35
36 return 0;
37 }
```

3.4 Estrutura de Repetição

Considere um algoritmo em que você precise repetir um determinado conjunto de passos, por exemplo, um algoritmo para retirar uma lâmpada do bocal. Um passo deste algoritmo é realizar um movimento com a mão para girar a lâmpada. Este passo deve ser repetido até que a lâmpada desencaixe do bocal. Neste sentido, existem as estruturas de repetições nas linguagens de programação para permitir que uma instrução ou um bloco de instruções seja repetido em um algoritmo computacional. Estas estruturas também são conhecidas como estruturas de iteração ou estruturas de laço.

Vamos tomar como base Exemplo 3.2 [38], onde uma nota é solicitada ao usuário. Caso a nota seja maior que 7, é mostrado “Aprovado” na tela, caso contrário, é mostrado "Reprovado". O algoritmo utilizado, até então, só permite que uma nota seja digitada, ou seja, quando o usuário digita a nota, o programa apresenta o resultado e em seguida fecha. Mas digamos que agora queremos que o programa continue executando, solicitando notas e apresentando o resultado, até que o usuário digite o valor -1 para sair do programa. Este algoritmo é mostrado no pseudocódigo abaixo. Perceba que temos uma condição `nota <> -1` que será avaliada antes de executar as instruções que estão dentro do bloco do ENQUANTO. Desse modo, enquanto a nota digitada pelo usuário for diferente de -1, o programa irá solicitar uma nota e apresentar o resultado.

Exemplo 3.3 Pseudocódigo com estrutura de repetição

```
LEIA nota
ENQUANTO nota <> -1 FAÇA
    SE nota >= 7 ENTÃO
        ESCRIVA "Aprovado"
    SENÃO
        ESCRIVA "Reprovado"

    LEIA nota
FIM-ENQUANTO
```

Na linguagem C há três opções diferentes para se criar estruturas de repetição. São os comandos `while`, `do-while` e `for`. Veremos cada um deles em detalhes a seguir.

3.4.1 Comando while

Podemos usar o comando `while` quando desejamos que uma ou mais instruções sejam repetidas até que uma determinada condição seja atendida. A estrutura do `while` na linguagem C é bastante parecida com a do pseudocódigo apresentado anteriormente. Veja abaixo:

```
while (expressão) {  
    instrução 1;  
    instrução 2;  
    ...  
}
```

As instruções serão executadas repetidamente enquanto a expressão for verdadeira. Assim que essa condição tornar-se falsa, o laço para. Vejamos o exemplo abaixo, que consiste no pseudocódigo do Exemplo 3.3 [42] escrito na linguagem C.

Código fonte `code/cap3/whilenota.c`

whilenota.c

```
1 #include <stdio.h>  
2  
3 int main() {  
4     float nota;  
5  
6     scanf("%f", &nota);  
7     while (nota != -1) {  
8         if (nota >= 7)  
9             printf("Aprovado\n");  
10        else  
11            printf("Reprovado\n");  
12  
13            scanf("%f", &nota);  
14    }  
15  
16    return 0;  
17 }
```

Vamos tentar entender o código acima. Inicialmente, uma variável `nota`, do tipo `float`, é declarada (linha 4). Logo depois ocorre a leitura da primeira nota. Caso ela seja diferente de `-1`, o bloco de instruções dentro do `while` (linhas 8 a 14) será executado. O comando `while` fará com que as instruções em seu corpo sejam executadas, repetidamente, enquanto a condição `nota != -1` for verdadeira.

Vejamos outro exemplo. O programa `mostrado10vezes.c` [43] abaixo escreve 10 vezes na tela "Isto sera' mostrado 10 vezes.". A condição de parada do comando `while` é `cont <= 10`, o que significa que enquanto o valor de `cont` for menor ou igual a 10, o bloco de instruções será executado. Para que o laço tenha fim, a variável `cont` precisa ser incrementada até que alcance o valor 11.

Código fonte `code/cap3/mostrado10vezes.c`

mostrado10vezes.c

```
1 #include <stdio.h>  
2  
3 int main() {
```



```
4     int cont = 1;
5
6     while (cont <= 10) {
7         printf("Isto sera' mostrado 10 vezes.\n");
8         cont++;
9     }
10
11     return 0;
12 }
```

3.4.1.1 Exercício resolvido

ER 3.3: Escreva um programa que leia várias notas de alunos de uma turma. O programa deve ler notas até que o usuário digite o valor -1 . Após isso, o programa deve mostrar a média dessas notas.

Resposta:

Neste programa não sabemos previamente a quantidade de notas que o usuário precisa. Precisamos, nesse caso, utilizar um laço de repetição que fique lendo notas até que o valor -1 seja passado pelo usuário. Desse modo, podemos utilizar o comando `while` (linha 7) com a condição de parada `nota != -1`, ou seja, o usuário seguirá entrando com as notas enquanto os valores digitados forem diferentes de -1 . Dentro do laço, é solicitado que o usuário digite a nota (linha 8). Na linha 9, o valor digitado pelo usuário é armazenado na variável `nota`. Para calcular a média das notas, precisamos de duas informações: a soma de todas as notas e a quantidade de notas lidas. Utilizamos a variável `soma` para armazenar a soma das notas (linha 12) e a variável `cont` para armazenar a quantidade de notas (linha 13). A média das notas será mostrada depois do laço, ou seja, quando o usuário digitar -1 para a nota. Como o cálculo da média precisa da quantidade total das notas, seu cálculo e o comando para sua exibição devem ser executados após o término do laço (linha 17).

Código fonte `code/cap3/notas_alunos_c.c`

notas_alunos.c

```
1  #include <stdio.h>
2
3  int main() {
4      float nota = 0, soma = 0;
5      int cont = 0;
6
7      while (nota != -1) {
8          printf("Entre com a nota: ");
9          scanf("%f", &nota);
10
11          if (nota != -1) {
12              soma += nota;
13              cont++;
14          }
15      }
16
17      printf("Media das notas: %.2f", soma / cont);
18
19      return 0;
20 }
```

3.4.2 Comando do-while

Assim como o `while`, o comando `do-while` também é uma estrutura de repetição. Com semelhanças que vão além dos seus nomes, sobretudo em termos de funcionalidade, a principal diferença entre eles é que, no caso do `do-while`, o bloco de instruções dentro do laço é executado pelo menos uma vez, mesmo que a condição seja falsa. Isso acontece porque a condição de parada só é avaliada depois que as instruções são executadas. Segue abaixo a estrutura do comando `do-while`.

```
do {  
    instrução 1;  
    instrução 2;  
    ...  
} while (condição);
```

O comando começa com a palavra `do`, seguida do bloco de instruções a ser repetido. Após a execução dessas instruções, a condição é avaliada. Caso seja verdadeira, as instruções são executadas novamente, caso contrário, o laço se encerra. Para entender melhor, observe o exemplo abaixo. Este exemplo faz exatamente o mesmo que o exemplo `mostrado10vezes.c` [43], porém, utilizando o comando `do-while`.

Código fonte `code/cap3/do_while.c`

do_while.c

```
1  #include <stdio.h>  
2  
3  int main() {  
4      int cont = 1;  
5  
6      do {  
7          printf("Isto sera' mostrado 10 vezes.\n");  
8          cont++;  
9      } while (cont <= 10);  
10  
11     return 0;  
12 }
```

3.4.3 Comando for

Este comando também serve para criar um laço de repetição e geralmente é utilizado quando conhecemos a quantidade de vezes que queremos que as instruções sejam repetidas.

O formato do comando `for` é o seguinte:

```
for (expressão1; condição; expressão2) {  
    instrução1;  
    instrução2;  
    ...  
}
```

Em `expressão1`, uma variável, geralmente um contador, recebe um valor inicial. Essa variável será incrementada ou decrementada de acordo com a expressão definida em `expressão2`. O laço ficará executando as instruções (`instrução1` e `instrução2`) até que a condição seja falsa.

Vejamos o exemplo abaixo para entender a sequência de operações que são realizadas durante a execução do comando `for`. O programa `for_cont.c` [46] abaixo tem o mesmo resultado de `do_while.c` [45] mostrado anteriormente. Inicialmete, uma variável inteira é declarada (linha 4). Diferentemente do programa `do_while.c` [45], não atribuímos nenhum valor inicial à variável, visto que ela será inicializada na primeira expressão do comando `for`. Na linha 6 existe um comando `for`, cuja primeira expressão é a inicialização da variável `cont`. Em seguida, a condição `cont <= 10` é avaliada. Como inicialmente `cont = 0`, expressão é avaliada como verdadeira, e a instrução na linha 7 é executada. No fim da execução das instruções (nesse caso, especificamente, há apenas uma instrução) contidas dentro do `for`, a terceira expressão do `for` é avaliada, que nesse caso é um incremento (`cont++`). Então a variável `cont` é incrementada e passa a valer 2. Em seguida, a condição é verificada novamente e, caso seja verdadeira, executa novamente a instrução dentro do `for`. Essa sequência de passos é repetida até que a condição seja avaliada como falsa.

Código fonte `code/cap3/for_cont.c`

for_cont.c

```
1 #include <stdio.h>
2
3 int main() {
4     int cont;
5
6     for (cont = 1; cont <= 10; cont++)
7         printf("Isto sera' mostrado 10 vezes.\n");
8
9     return 0;
10 }
```

Para entender melhor o funcionamento do comando `for`, vamos ver outro exemplo que utiliza decremento (`--`) ao invés do incremento. A ideia é muito parecida com a do exemplo anterior, entretanto, ao invés de incremento, temos um decremento da variável `cont`. No comando `for`, a variável `cont` é inicializada com o valor 10. Em seguida, a condição `cont > 0` é avaliada. Como o valor de `cont` é 10, a condição é verdadeira, e a instrução dentro do `for` é executada (linha 7). Ao fim da execução dessa instrução, a variável `cont` é decrementada e passa a valer 9. Novamente a condição `cont > 0` é verificada e continua sendo verdadeira, o que faz com que a instrução da linha 7 seja executada novamente. O laço continua até a variável `cont` passar a valer 0, caso no qual a condição `cont > 0` será falsa.

Código fonte `code/cap3/for_cont_decremento.c`

for_cont_decremento.c

```
1 #include <stdio.h>
2
3 int main() {
4     int cont;
5
6     for (cont = 10; cont > 0; cont--)
7         printf("Valor de cont: %i\n", cont);
8
9     return 0;
10 }
```

Uma particularidade do comando `for` é que nenhum dos três elementos que o compõe é obrigatório, ou seja, podemos omitir qualquer um desses elementos ou até mesmo uma combinação deles.

Contudo, essa é uma prática que deve ser evitada.

3.4.4 Laço infinito

Quando a condição de parada de uma determinada estrutura de repetição nunca é alcançada, as instruções do laço são executadas indefinidamente, e consequentemente o programa nunca chega ao seu fim. Tal comportamento é algo que deve ser evitado em programação (na grande maioria das situações) e, por ser um problema tão recorrente, recebe um nome especial: laço infinito. O exemplo abaixo ilustra esse tipo de laço:

Código fonte code/cap3/laco_infinito.c

laco_infinito.c

```
1  #include <stdio.h>
2
3  int main() {
4      int cont;
5
6      for (cont = 1; ; cont++)
7          printf("Laco infinito.\n");
8
9      return 0;
10 }
```

Observe que na linha 6 temos um comando `for` cuja condição de parada não foi definida. Consequentemente, o laço entra em loop e a execução da instrução na linha 7 é executada indefinidamente.



Atenção

Ao criar uma estrutura de repetição, observe bem a condição de parada para verificar se ela realmente será alcançada em algum momento. Sem esse cuidado você corre o risco de criar um laço infinito, e seu programa, possivelmente, não terá o resultado esperado.

3.4.5 Exercício Resolvido

E.R 3.4. Analise o programa abaixo. O objetivo do programa é ler 50 valores e mostrar, ao final da leitura, o menor deles. Entretanto, ele possui um problema. Identifique e corrija o erro.

Código fonte code/cap3/menor_deles.c

menor_deles.c

```
1  #include <stdio.h>
2
3  int main() {
4      int quantidade = 1;
5      float valor, menor;
6
7      printf("Informe um valor: ");
8      scanf("%f", &menor);
9
10     while (quantidade < 50) {
```

```
11     printf("Informe um valor: ");
12     scanf("%f", &valor);
13     if (valor < menor)
14         menor = valor;
15 }
16
17 printf("Menor valor lido: %f", menor);
18
19 return 0;
20 }
```

Resposta:

Vamos analisar o código considerando seu objetivo: ler 50 valores e apresentar o menor deles. Vamos iniciar nossa análise na estrutura de repetição `while`, na linha 10. A condição é `quantidade < 50`. Logo percebemos que a variável `quantidade`, que tem o valor inicial de 1 na sua declaração, não é alterada em nenhum momento dentro do laço, o que sugere algum problema. Para o laço ter um fim, é necessário que a variável `quantidade` atinja o valor 50 em algum momento, e não é isso que está acontecendo. Portanto, temos um laço infinito. Para corrigir o problema, devemos incluir uma linha dentro do laço a fim de incrementar o valor da variável `quantidade`.

O código abaixo apresenta a solução do problema com a inclusão da linha 12, na qual a variável `quantidade` é incrementada. Dessa forma, o laço deixou de ser infinito, uma vez que ele atingirá o valor 50 em algum momento, tornando falsa a condição `quantidade < 50` do laço.

Código fonte `code/cap3/menor_deles_resposta.c`

menor_deles_resposta.c

```
1  #include <stdio.h>
2
3  int main() {
4      int quantidade = 1;
5      float valor, menor;
6
7      printf("Informe um valor: ");
8      scanf("%f", &menor);
9
10     while (quantidade < 50) {
11         printf("Informe um valor: ");
12         scanf("%f", &valor);
13
14         if (valor < menor)
15             menor = valor;
16
17         quantidade++; // Solucao do problema
18     }
19
20     printf("Menor valor lido: %.2f", menor);
21
22     return 0;
23 }
```

3.4.6 Comandos de desvio

Vimos no último exemplo o que é um laço infinito. Em alguns momentos, precisamos utilizar um comando para realizar um desvio dentro do laço de repetição. Esses comandos são: `break` e `continue`. Ambos podem ser utilizados em qualquer estrutura da repetição.

3.4.6.1 Comando `break`

O comando `break` interrompe a execução do laço, fazendo com que as instruções dentro do laço após esse comando não sejam executadas. Vamos ver o exemplo `break_interrompe.c` [49]. Apesar de não haver condição de parada no `for`, ele irá parar quando o comando `break` for executado, ou seja, quando o valor da variável `cont` for igual a 10. Caso houvesse mais alguma outra instrução após a linha 8, dentro do bloco de instruções do `for`, ela não seria executada.

Código fonte `code/cap3/break_interrompe.c`

`break_interrompe.c`

```
1 #include <stdio.h>
2
3 int main() {
4     int cont;
5
6     for (cont = 1; ; cont++) {
7         printf("Valor de cont: %i\n", cont);
8         if (cont == 10) break;
9     }
10
11     return 0;
12 }
```

3.4.6.2 Comando `continue`

O comando `continue` faz com que o fluxo de execução “salte” para a avaliação da condição de parada do laço, no caso do `while` e `do-while`, e para a expressão de incremento e decremento, no caso do comando `for`. Isso significa que as instruções após esse comando não são executadas.

Agora vamos analisar o programa `continue_desvio.c` [49] abaixo. Esse programa mostra na tela os números ímpares no intervalo de 1 a 20. A variável `cont` é inicializada com o valor 1 no `for`. Como a condição é verdadeira, as instruções dentro do bloco são executadas. Agora atenção para o comando `if` (linha 7). Na condição do `if`, há uma expressão `cont % 2 == 0`, o que significa que se o resto da divisão inteira entre a variável `cont` e 2 for 0, o comando `continue` é executado. Entretanto, o resto da divisão será 1. Nesse caso, a instrução da linha 8 é executada, mostrando o valor 1 na saída. Em seguida, a expressão de incremento do `for` é avaliada, e `cont` passa a valer 2. Como `cont` ainda é menor ou igual a 20, as instruções do bloco são executadas novamente. Mais uma vez, na linha 6, a condição do `if` é avaliada. Todavia, o resto da divisão de `cont` e 2 agora é igual a 0, e então a instrução `continue` é executada. Com sua execução, o fluxo de volta para o `for`, e a expressão de incremento é avaliada. Note que a instrução da linha 7, nesse caso, não será mais executada. O valor de `cont` então é incrementado para 3, repetindo o que foi explicado quando `cont` era igual a 1.

Código fonte `code/cap3/continue_desvio.c`

continue_desvio.c

```
1  #include <stdio.h>
2
3  int main() {
4      int cont;
5
6      for (cont = 1; cont <= 20; cont++) {
7          if (cont % 2 == 0) continue;
8          printf("Valor de cont: %i\n", cont);
9      }
10
11     return 0;
12 }
```

3.5 Recapitulando

Este capítulo apresentou as estruturas de controle da linguagem C e como elas podem ser utilizadas.

A estrutura sequencial significa que o fluxo de execução das instruções segue uma linha sequencial, que no caso da linguagem C, é de cima para baixo e da esquerda para a direita. Entretanto, podemos mudar o fluxo de execução dessas instruções utilizando as estruturas de decisão e de repetição. No caso da estrutura de decisão, aprendemos a utilizar os comandos `if`, `else` e `switch` para executar uma instrução ou um bloco de instruções caso uma determinada condição seja verdadeira.

Aprendemos também a construir laços de repetição com os comandos `while`, `do-while` e `for`. Vimos que todos esses comandos proporcionam a repetição de instruções até que uma determinada condição seja falsa. Diante disso, é importante ter bastante atenção na condição de parada dessas estruturas de repetição, para não criar um laço infinito.

No próximo capítulo estudaremos os arranjos: uma estrutura de dados que tem o objetivo de representar um conjunto de valores do mesmo tipo. Vamos também aprender a manipular as cadeias de caracteres, também conhecidas como strings.

3.6 Exercícios Propostos

1. Escreva um programa que verifique se um número digitado pelo usuário é menor, igual ou maior que zero.
2. Dado o algoritmo abaixo, explique o que acontece se o valor lido para a variável `x` for: 3, 1 e 0. Explique o porquê.

```
#include <stdio.h>

int main() {
    int x;

    scanf(&x);
    if (x) printf("verdadeiro");

    return 0;
}
```

```
}
```

3. Escreva um programa que informe se um dado ano é ou não bissexto. Obs.: um ano é bissexto se ele for divisível por 400 ou se ele for divisível por 4 e não por 100.
4. Escreva um programa que mostre todos os números pares no intervalo de 1 a 40 de forma decrescente, utilizando o comando `while`. Depois faça o mesmo, mas desta vez, utilizando o comando `for`.
5. Um determinado banco abriu uma linha de crédito para os funcionários públicos. Porém, o valor máximo da prestação não poderá ultrapassar 30% do salário deste funcionário. Faça um programa para ajudar este banco. O programa deve permitir o usuário entrar com o salário do funcionário e o valor da prestação e informar se o empréstimo pode ou não ser concedido.
6. Escreva um programa que leia o mês do ano em valor numérico e exiba este mês por extenso (utilize o comando `switch`).
7. Faça três programas que mostrem de 1 a 10 na tela, utilizando, em cada um, uma estrutura de laço de repetição diferente.
8. Escreva um programa que mostre na tela os números múltiplos de 3 no intervalo de 2 a 100.
9. Escreva um programa para ler dois números inteiros M e N e, a seguir, imprimir os números pares existentes no intervalo [M, N].
10. A organização de um evento esportivo deseja um programa que faça a leitura do nome e a pontuação de cada um dos 10 participantes e exiba o nome do vencedor. Elabore este programa.
11. O supermercado Excelente Preço está precisando ser informatizado. Neste sentido, o dono quer um programa que leia os preços dos produtos até que seja informado o valor zero. No final o programa deve informar o total da compra e perguntar a forma de pagamento. As opções da forma de pagamento são: 1) A vista; 2) No cartão de crédito. Se a opção escolhida for a vista, então o programa informa o valor da compra com um desconto de 5%. Caso a compra seja no cartão de crédito, o programa informa o valor da compra dividido em 4 vezes.

Feedback sobre o capítulo

Você pode contribuir para melhoria dos nossos livros. Encontrou algum erro? Gostaria de submeter uma sugestão ou crítica?



Acesse <https://github.com/edusantana/introducao-a-programacao-livro/issues/new> para realizar seu feedback. Lembre-se de incluir na mensagem a seção, capítulo (**cap3**) e a versão do livro (**v1.0.2**) alvo de sua contribuição. Você receberá notificações sobre os encaminhamentos que serão dados a partir do seu feedback. Para compreender melhor como feedbacks funcionam consulte o guia do curso.

Capítulo 4

Arranjos

OBJETIVOS DO CAPÍTULO

Ao final deste capítulo você deverá ser capaz de:

- Apresentar os conceitos de vetores e matrizes
- Apresentar o conceito de strings e como manipulá-las

Sumário

4.1	Introdução	52
4.2	Vetores	52
4.2.1	Declaração de Vetores	53
4.2.2	Acessando os elementos de um vetor	53
4.2.3	Exercício resolvido	55
4.3	Strings	57
4.3.1	Lendo e imprimindo Strings	57
4.3.2	Manipulando strings	58
4.3.3	Exercício resolvido	59
4.4	Matrizes	59
4.5	Resumindo	61
4.6	Exercícios Propostos	61

4.1 Introdução

Até agora vimos que uma variável armazena um único valor por vez. Por exemplo, em um programa para ler as notas de vários alunos, cada nota é armazenada em uma variável, e assim que as notas de um novo aluno são lidas, as notas do aluno anterior são perdidas. Em alguns problemas, é necessário armazenar todos ou um conjunto de valores lidos sem que haja perda de informação. Nesse caso, seria inviável declarar uma variável distinta para armazenar cada valor, quando a quantidade de valores a serem manipulados for relativamente grande. Para situações como essas utilizamos Arranjos (em inglês *Arrays*), que consistem em estruturas de dados capazes de agrupar em uma única variável

vários elementos de um mesmo tipo. O conceito de Arranjos, bem como as diferentes formas de utilizá-los serão discutidos em detalhes no decorrer deste capítulo, que ainda apresentará as cadeias de caracteres, conhecidas como *strings*.

4.2 Vetores

Os arranjos podem ter diferentes dimensões. Um tipo especial de arranjo com apenas uma dimensão é chamado de **vetor**. Portanto, vetores são arranjos unidimensionais que representam um conjunto de variáveis com o mesmo tipo, as quais são acessadas através de um índice que as identificam. A Figura 4.1 [52] ilustra o conceito de vetor, apresentando um vetor de inteiros com cinco elementos, cada um com seu índice correspondente. O índice do primeiro elemento é sempre zero.

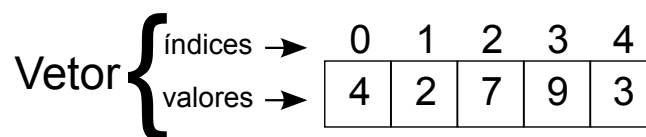


Figura 4.1: Vetor com cinco elementos.

Nota



O conceito de arranjo nas diferentes linguagens de programação é o mesmo. Entretanto, em algumas linguagens alguns detalhes podem ser diferentes. Na linguagem C, por exemplo, o índice inicial é sempre zero. Já na linguagem Pascal, o índice inicial é definido pelo próprio programador.

4.2.1 Declaração de Vetores

Na linguagem C, devemos declarar um vetor da seguinte forma:

```
tipo_vetor nome_vetor[quantidade_elementos];
```

O `tipo_vetor` é o tipo de cada elemento do vetor. O `nome_vetor` é o nome da variável que irá identificar o vetor. A `quantidade_elementos` representa a quantidade máxima de elementos que o vetor poderá armazenar. Observe que essa quantidade deve ficar entre colchetes. Os índices do vetor irão de zero até `quantidade_elementos - 1`. O compilador irá reservar o espaço de memória correspondente ao que o programador definiu em `quantidade_elementos`.

Vamos ver alguns exemplos de declarações de vetores:

```
int idades[50];
char nomes[200];
float precos[30];
```

No exemplo acima temos três declarações de vetores diferentes. Na primeira linha temos a declaração de um vetor chamado `idades` que terá no máximo 50 elementos do tipo `int`, com índices de 0 a 49. Na segunda linha temos um vetor chamado `nomes` com 200 elementos do tipo `char`, com índices de 0 a 199. Por fim, temos na última linha um vetor com identificador `precos`, com espaço para armazenar 30 elementos do tipo `float`, cujos índices variam de 0 a 29.

4.2.2 Acessando os elementos de um vetor

Uma vez que um vetor foi declarado, poderemos armazenar e ler os valores dos elementos desse vetor. Para isso, devemos identificar o elemento do vetor que queremos acessar através do nome do vetor seguido do índice do elemento entre colchetes. Observe o exemplo abaixo:

```
1 ...
2 int main() {
3     int v[5], i;
4     for (i = 0 ; i < 5 ; i++)
5         scanf("%d", &v[i]);
6 ...
```

Na linha 3, um vetor `v` de 5 elementos do tipo `int` é declarado. Em seguida, temos um comando `for` que irá se repetir 5 vezes, com a variável `i` variando de 0 a 4. No corpo da estrutura de repetição, na linha 5, ocorre a leitura de valores inteiros, na qual cada valor lido é armazenado em um elemento do vetor `v` com índice igual ao valor da variável `i` em cada iteração. Para um melhor entendimento, considere a situação hipotética a seguir. Suponha que o usuário digitou os seguintes valores na leitura de dados: 4, 3, 7, 2 e 9. Nesse caso, o vetor ficaria da seguinte forma após a leitura dos valores:

$$\text{Vetor } v \left\{ \begin{array}{l} v[0] = 4 \\ v[1] = 3 \\ v[2] = 7 \\ v[3] = 2 \\ v[4] = 9 \end{array} \right.$$

Figura 4.2: Configuração do vetor `v` após a leitura dos valores.

Uma vez que os elementos de um vetor foram inicializados, isto é, receberam algum valor através de uma operação de escrita na memória, podemos acessar tais valores para diversos fins. Por exemplo, podemos implementar um laço para percorrer todo o vetor e imprimir na tela os valores de cada um de seus elementos. O trecho de código a seguir ilustra como os elementos de um vetor são acessados e mostra como um laço para percorrer um vetor pode ser implementado:

```
1 ...
2 printf("%d", v[0]); //mostra o valor 4 na saída
3 printf("%d", v[2]); // mostra o valor 7 na saída
4 for (i = 0 ; i < 5 ; i++)
5     printf("%d ", v[i]); //mostra todos os valores
6 printf("%d", v[5]);
7 ...
```

Na linha 2 estamos pedindo para ser mostrado o valor do elemento de índice 0 do vetor `v`. Sendo assim, o valor 4 é mostrado na tela. Na linha 3, é mostrado o valor do elemento de índice 2 do vetor `v`, cujo valor é 7. Já na linha 4 temos um laço de repetição que irá mostrar todos os valores dos elementos do vetor `v`. Em seguida, na linha 6, estamos tentando acessar o elemento de índice 5, que não existe. O programa irá compilar normalmente, entretanto, ocorrerá um erro em tempo de execução assim que o programa tentar acessar o elemento inexistente do vetor.

**Cuidado**

Tenha cuidado ao acessar elementos de um vetor cujo índice não existe. Caso isso aconteça, o programa irá compilar, entretanto, ocorrerá um erro em tempo de execução.

Vamos fazer um outro exemplo para fixar melhor o assunto. Digamos que queremos um programa para ler 20 valores do tipo inteiro e que após isso, sejam mostrados esses mesmos valores na ordem inversa da qual foram lidos. O exemplo abaixo mostra a solução do problema.

```
1 int main() {
2     int i, valores[20];
3     for (i = 0 ; i < 20 ; i++ ) //primeira etapa
4         scanf("%d", &valores[i]);
5     for (i = 19 ; i >= 0 ; i--) //segunda etapa
6         printf("%d ", valores[i]);
7     return 0;
8 }
```

Podemos dividir o problema em duas etapas. A primeira é para “montar” um vetor com 20 valores digitados pelo usuário. A segunda parte é para mostrar os valores desse vetor na ordem inversa da qual foram lidos. Inicialmente, declaramos o vetor ``valores` com 20 elementos do tipo `int`. Para resolver cada etapa do problema, utilizamos um laço de repetição com o comando `for`. Na linha 3, o laço `for` é utilizado para ler os valores, cada um sendo armazenado em um elemento do vetor. Na linha 5, temos um outro laço `for`, cuja variável de controle `i` é inicializada com o valor 19, que representa o índice do último elemento do vetor. A condição de parada do laço é que a variável `i` seja maior ou igual a zero e a última expressão do `for` é o decremento da variável `i`. Isso significa que o valor da variável `i` irá de 19 a 0 dentro do laço de repetição. Consequentemente, os valores dos elementos do vetor `valores` irão ser mostrados na ordem inversa da que foram lidos.

4.2.3 Exercício resolvido

ER 4.1

Escreva um programa que leia 20 notas de alunos de uma turma. O programa deve calcular a média da turma e apresentar na tela apenas as notas dos alunos que ficaram acima da média calculada.

Resposta

A primeira etapa para resolver esse problema é analisar se precisamos realmente utilizar um arranjo. Mas antes disso, vamos tentar dividir o problema em subproblemas menores para facilitar a elaboração da solução.

Subproblema	Descrição
Subproblema 1	Ler 20 notas de uma turma
Subproblema 2	Calcular a média da turma considerando as 20 notas lidas
Subproblema 3	Apresentar na tela as notas da turma que estão acima da média calculada

O primeiro subproblema é ler 20 notas dos alunos. A princípio conseguiríamos ler as 20 notas utilizando um laço de repetição e apenas uma variável. Entretanto, se utilizarmos apenas uma variável para ler as notas, só teremos a última nota armazenada ao final do laço. Como precisamos de todas as notas para saber quais delas estão acima da média calculada, a solução do subproblema 3 requer que um vetor seja utilizado ao invés de apenas uma variável. Resolveremos, a partir de agora, a questão gradativamente a partir das soluções dos subproblemas. Vejamos o trecho de código abaixo que resolve o subproblema 1 da questão.

```
1 int main() {
2
3     float nota[20];
4     int i;
5
6     for (i = 0 ; i < 20 ; i++ )
7         scanf("%f", &nota[i]);
8     ...
```

Como foi explicado, precisaremos de um vetor para armazenar todas as notas da turma. Portanto, na linha 3 declaramos um vetor de `float` chamado `notas` com 20 elementos. Utilizamos a estrutura de repetição `for` para ler as 20 notas passadas como entrada para o programa (linha 6 e 7).

O subproblema 2 consiste em calcular a média das 20 notas lidas. Para isso, precisamos primeiramente somar todas as notas lidas. Uma vez que temos essas notas armazenadas em um vetor, poderíamos resolver esse subproblema de duas maneiras. Uma delas é criar, após o laço de leitura das notas, um outro laço para acessar os elementos do vetor para realizar a soma das notas. A segunda forma, mais interessante, é somar as notas à medida que forem lidas, no mesmo laço de repetição. Dessa forma, teremos apenas um laço de repetição, tornando nosso código mais eficiente. Verifiquemos como ficou o trecho de código anterior com a inclusão da solução para o subproblema 2.

```
1 int main() {
2
3     float nota[20], media, soma = 0;
4     int i;
5
6     for (i = 0 ; i < 20 ; i++ ) {
7         scanf("%f", &nota[i]);
8         soma += nota[i];
9     }
10
11     media = soma / 20;
12     ...
```

Perceba que incluímos na declaração as variáveis `media` e `soma`, para armazenar, respectivamente, os valores da média e da soma das notas. Verifique também, na linha 8, que a soma das notas é realizada à medida que cada nota é lida. O cálculo da média, na linha 11, só pode ser feita depois que todas as notas forem lidas e acumuladas na variável `soma`. Desse modo, ela deve ficar fora do laço.

O subproblema 3 consiste em apresentar na tela as notas da turma que ficaram acima da média. Uma vez que já temos a média calculada, podemos resolver facilmente esse problema. Para tanto, precisamos percorrer todo o vetor e comparar cada elemento deste com a média. Caso o valor do elemento seja maior que a média, ele deve ser apresentado na tela. A versão final do programa está descrita abaixo:

```
1 int main() {
```

```
2     float nota[20], media, soma = 0;
3     int i;
4
5     for (i = 0 ; i < 20 ; i++) {
6         scanf("%f", &nota[i]);
7         soma += nota[i];
8     }
9
10    media = soma / 20;
11
12    for (i = 0 ; i < 20 ; i++)
13        if (nota[i] > media)
14            printf("%f ", nota[i]);
15
16    return 0;
17 }
```

As linhas 12 a 14 resolvem o subproblema 3, utilizando o comando `for` para percorrer todo o vetor de notas e comparar cada elemento com a média. Para realizar tal comparação, utilizamos o comando `if`. Caso o elemento `nota[i]` seja maior que a média, ele será exibido na tela.

4.3 Strings

Na maioria dos programas que implementamos, precisamos lidar com cadeias de caracteres no processamento e armazenamento das informações que nos são fornecidas. Nas linguagens de programação, essas cadeias são chamadas de **strings**. Como não há, na linguagem C, um tipo de dados específico para representar as strings, utilizamos vetores de elementos do tipo `char` para a manipulação de cadeias de caracteres. Em outras palavras, quando queremos declarar uma variável para armazenar uma cadeia de caracteres, declaramos um vetor do tipo `char`.

No exemplo abaixo, temos a declaração de três strings: `nome`, `logradouro` e `bairro`. Perceba que cada string é um array de `char`. Significa dizer que a string `nome` tem a capacidade de armazenar 60 caracteres, a string `logradouro` 200 caracteres e a string `bairro` 40 caracteres.

```
char nome[60];
char logradouro[200];
char bairro[40];
```

Na linguagem C, toda string tem um caractere especial que determina o seu fim. Esse caractere é o `\0`, que significa **NULO**, e ele é inserido automaticamente pelo compilador no último elemento do vetor de elementos do tipo `char`. Por essa razão, deve-se levar isso em consideração na hora de declarar suas strings. Assim sendo, se você deseja que uma string armazene `N` caracteres, você deve declará-la com um tamanho `N+1`.

4.3.1 Lendo e imprimindo Strings

Para ler e imprimir strings na linguagem C, podemos utilizar as funções `scanf()` e `printf()` que já conhecemos. Entretanto, como muitas vezes precisamos manipular strings no nosso programa, as linguagens de programação possuem funções pré-definidas para serem utilizadas pelo programador,

facilitando a manipulação das cadeias de caracteres. A linguagem C, por exemplo, possui as funções `gets()` e `puts()`, elaboradas especialmente para ler e imprimir strings, respectivamente. Vejamos um exemplo.

```
1 int main() {  
2  
3     char s[7];  
4  
5     gets(s);  
6     puts(s);  
7  
8     printf("%c", s[4]);  
9     printf("%c", s[2]);  
10  
11     return 0;  
12 }
```

Na linha 3, declaramos uma string `s` que armazena 6 caracteres (além do caractere NULO). Em seguida, utilizamos a função `gets()`, que faz a leitura de uma string, esperando que o usuário a digite e tecele ENTER. Na linha 6, utilizamos a função `puts()`, que imprime na saída padrão a string `s` lida. Digamos que o usuário digite a string `'BRASIL'`. Dessa forma, a cadeia de caracteres será armazenada na memória da seguinte forma:

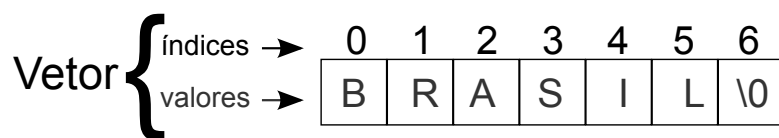


Figura 4.3: Cadeia de caracteres.

Como uma string em C trata-se na verdade de um vetor de caracteres, podemos acessá-los individualmente do mesmo modo que acessamos os elementos de um vetor qualquer. Ao serem executadas as linhas 8 e 9 do código anterior, por exemplo, são exibidos na saída padrão os caracteres “I” e “A”, respectivamente.

4.3.2 Manipulando strings

Existem ainda outras funções interessantes para manipular strings na linguagem C. A tabela abaixo apresenta algumas dessas funções, todas presentes na biblioteca `string.h`.

Função	Descrição
<code>strlen(s)</code>	Retorna a quantidade de caracteres da string <code>s</code>
<code>strcpy(s1, s2)</code>	Copia o conteúdo da string <code>s2</code> para <code>s1</code>
<code>strcat(s1, s2)</code>	Concatena (junta) o conteúdo da string <code>s2</code> em <code>s1</code>
<code>strchr(s, c)</code>	Retorna a posição (inteiro) do caractere <code>c</code> na string <code>s</code>

Para entender melhor o uso destas funções, considere o exemplo a seguir.

Código fonte `code/cap4/manipulacao_string.c`

manipulacao_string.c

```
1 #include <stdio.h>
2
3 int main() {
4     char str1[100], str2[100];
5
6     gets(str1);
7     gets(str2);
8
9     printf("%d", strlen(str1));
10    printf("%d", strlen(str2));
11
12    strcat(str1, str2);
13    printf("%d", strlen(str1));
14
15    return 0;
16 }
```

Inicialmente declaramos duas strings: `str1` e `str2`. Nas linhas 6 e 7, utilizamos o comando `gets()` para que o usuário informe as duas strings que serão armazenadas nas variáveis mencionadas. As linhas 9 e 10 imprimem os tamanhos das strings `str1` e `str2`, enquanto que a linha 12 é responsável por concatenar as strings `str2` e `str1`, ou seja, `str1` passa a ter o conteúdo anterior com a adição do conteúdo de `str2`. Por fim, a linha 13 exibe o tamanho de `str1` após a sua concatenação com `str2`.

4.3.3 Exercício resolvido

ER 4.2

Escreva um programa que leia uma string e substitua todos os espaços por um traço (caractere “-”).

Resposta

Primeiramente, declaramos uma string `s` com capacidade para armazenar 40 caracteres e utilizamos a função `gets()`, na linha 5, a fim de que o usuário digite uma string. Tendo a string digitada armazenada na variável `s`, podemos agora manipulá-la. Para resolver essa questão, é importante ter entendido o conceito de que uma string é um vetor de caracteres. Na linha 7, utilizamos um `for` para percorrer os elementos da string. Veja que a condição de parada do comando `for` é `i < strlen(s)`. Isso significa que o bloco de instruções será repetido até o final da string, uma vez que `strlen()` retorna a quantidade de caracteres de uma string. Dentro do `for`, verificamos se o caractere da posição `i` é igual a um espaço. Caso seja, esse elemento do array recebe o caractere “-”. Finalmente, depois do comando `for`, utilizamos a função `puts()` para imprimir a nova string com os espaços trocados por “-”.

Código fonte `code/cap4/resolvido4-2.c`

resolvido4-2.c

```
1 int main() {
2     char s[40];
3     int i;
4 }
```



```

5     gets(s);
6
7     for (i=0; i < strlen(s); i++) {
8         if (s[i] == ' ')
9             s[i] = '-';
10    }
11
12    puts(s);
13
14    return 0;
15 }

```

4.4 Matrizes

Como foi dito no início do capítulo, arranjos podem ter várias dimensões. Quando possuem mais de uma dimensão, eles são chamados de **matrizes**. O conceito de matrizes em programação é bem semelhante ao homônimo da matemática. A figura a seguir apresenta o formato de uma matriz $m \times n$, onde m representa o número de linhas e n o número de colunas.

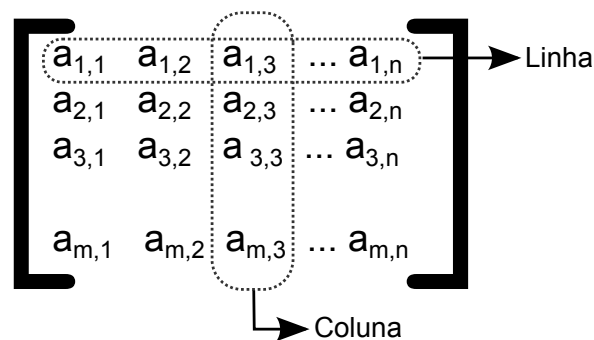


Figura 4.4: Matriz $M \times N$

Analisemos os exemplos abaixo. A primeira matriz é 3×3 , pois possui 3 linhas e 3 colunas. A segunda matriz é 2×3 , pois possui 2 linhas e 3 colunas. Se quisermos saber o elemento da primeira matriz que possui índice $A_{2,3}$, basta seguirmos em direção à 2ª linha e depois em direção à 3ª coluna. Logo o valor de $A_{2,3}$ é 1.

$$\begin{bmatrix} 4 & 2 & 3 \\ 8 & 0 & 1 \\ 9 & 1 & 6 \end{bmatrix} \quad \begin{bmatrix} 5 & 1 & 0 \\ 9 & 2 & 1 \end{bmatrix}$$

Matriz 3×3 Matriz 2×3

Figura 4.5: 4.5. Exemplos de matrizes

Podemos representar essas matrizes na linguagem C acrescentando mais um índice entre colchetes no identificador do arranjo. Abaixo temos alguns exemplos de como declaramos matrizes:

```
int matriz[3][3];
int outra_matriz[2][3];
float matriz_de_float[30][20];
char nomes[10][50];
```

Na primeira linha temos a declaração de uma matriz de inteiros com 3 linhas e 3 colunas, na segunda novamente uma matriz de inteiros, só que agora com 2 linhas e 3 colunas, e na terceira uma matriz de elementos do tipo `float` com 30 linhas e 20 colunas. Agora atenção para última linha. Trata-se da declaração de um arranjo de strings. Lembre-se que uma string é um arranjo de caracteres. Se declararmos uma matriz de `char`, então teremos na prática um vetor de strings, onde cada linha da matriz é uma cadeia de caracteres.

Já sabemos como declarar matrizes, agora aprenderemos a montá-las a partir da leitura de dados da entrada padrão. Para isso, precisaremos utilizar dois comandos `for` aninhados. Considere o exemplo abaixo:

Código fonte `code/cap4/matriz_populando.c`

matriz_populando.c

```
1 int main() {
2     int matriz[20][30];
3     int i, j;
4
5     for (i=0; i < 20; i++)
6         for (j = 0; j < 30; j++)
7             scanf("%d", &matriz[i][j]);
8
9     return 0;
10 }
```

Na linha 2, temos uma declaração de uma matriz 20 x 30. Se quisermos pedir para o usuário digitar os valores da matriz, precisaremos utilizar um `for` para percorrer as linhas e um `for` para percorrer as colunas da matriz. Portanto, na linha 5, temos o primeiro `for` onde `i` irá variar de 0 a 19, justamente os índices que representam a posição dos elementos nas linhas da matriz. Na linha 6, temos um outro `for` dentro do primeiro, onde `j` irá variar de 0 a 29, que são justamente os índices que representam a posição dos elementos nas colunas da matriz. Perceba que quando `i = 0`, `j` irá variar de 0 a 29. Depois `i` passa a valer 1 e novamente o `j` irá variar de 0 a 29 novamente. Isso acontecerá repetidamente até `i` atingir o valor 19. Em suma, o código anterior preenche os elementos da matriz linha por linha. Inicia preenchendo a linha de índice 0, em seguida preenche a linha de índice 1, seguindo de forma sucessiva até que a linha de índice 19 seja preenchida.

4.5 Recapitulando

Como vimos neste capítulo, arranjos consistem em um conjunto de elementos do mesmo tipo, que podem ter diferentes dimensões e são acessados por um índice. Os arranjos de uma dimensão são chamados de vetores e quando possuem mais de duas dimensões são chamados de matrizes.

Aprendemos também que strings são cadeias (arranjos) de caracteres, ou seja, um vetor de elementos do tipo `char`. Devido à sua importância para a programação de computadores, as linguagens de programação disponibilizam um conjunto de funções para facilitar sua manipulação. Algumas das

funções mais utilizadas na linguagem C foram conhecidas neste capítulo, contudo, deve ficar claro ao leitor que existe um número bem maior de funções com tal objetivo.

4.6 Exercícios Propostos

1. Crie um programa que armazene números em dois vetores inteiros de cinco elementos cada, depois gere e imprima o vetor soma dos dois.
2. Crie um programa que armazene 10 números em um vetor A, e gere um vetor B onde cada elemento é o quadrado do elemento de A.

Exemplo:

```
A[1] = 4    B[1] = 16
A[2] = 3    B[2] = 9
A[3] = 6    B[3] = 36
```

3. Escreva um programa para ler uma string qualquer e exiba as seguintes informações: quantidade de caracteres, primeira e última letra.
4. Escreva um programa para ler uma frase de no máximo 70 caracteres e exibir a quantidade de vogais dessa frase.
5. Escreva um programa que leia uma string qualquer e mostre-a invertida.

Exemplo:

```
Entrada: casa <ENTER>
Saída: asac
```

6. Um palíndromo é uma cadeia de caracteres que representa a mesma palavra nos sentidos direto e inverso. Por exemplo, “asa” é um palíndromo, porque o inverso dela também é “asa”. Faça um programa que leia uma string e diga se esta é ou não um palíndromo.
7. Escreva um programa para ler 9 números inteiros para preencher uma matriz D 3x3, ou seja, com 3 linhas e 3 colunas (considere que não serão informados valores duplicados). A seguir, ler um número inteiro X e escrever uma mensagem indicando se o valor de X existe ou não na matriz D.

Feedback sobre o capítulo

Você pode contribuir para melhoria dos nossos livros. Encontrou algum erro? Gostaria de submeter uma sugestão ou crítica?



Acesse <https://github.com/edusantana/introducao-a-programacao-livro/issues/new> para realizar seu feedback. Lembre-se de incluir na mensagem a seção, capítulo (**cap4**) e a versão do livro (**v1.0.2**) alvo de sua contribuição. Você receberá notificações sobre os encaminhamentos que serão dados a partir do seu feedback. Para compreender melhor como feedbacks funcionam consulte o guia do curso.

Capítulo 5

Funções

OBJETIVOS DO CAPÍTULO

Ao final deste capítulo você deverá ser capaz de:

- Criar e usar funções e procedimentos em C
- Identificar quando escrever uma função ou um procedimento
- Entender as regras de escopo para variáveis em C
- Entender os diferentes tipos de passagem de parâmetro e quando utilizá-los
- Trabalhar com funções recursivas

Sumário

5.1	O que são funções?	63
5.1.1	Um exemplo	64
5.2	Parâmetros	67
5.3	Retorno de Valores com Funções	69
5.3.1	Funções, Procedimentos e o Tipo <code>void</code>	70
5.4	Um Exemplo Matemático: Equação de Segundo Grau	73
5.5	Escopo de Variáveis	76
5.5.1	Escopo dos Parâmetros	78
5.5.2	Sombreamento e Sobreposição de Escopos	78
5.6	Passagem de Parâmetros	80
5.6.1	Passagem por Valor	81
5.6.2	Passagem por Referência	82
5.7	Protótipos e Declaração de Funções	85
5.8	Funções Recursivas	87
5.9	Recapitulando	90
5.10	Exercícios Propostos	90

Quando elaboramos um algoritmo para resolver um problema, e quando escrevemos um programa para implementar este algoritmo, muitas vezes identificamos uma parte do algoritmo que deve ser realizada várias vezes para chegar ao resultado. Ou, ao trabalhar com vários algoritmos diferentes, identificamos algumas partes em comum entre eles. Nestes casos, é uma boa ideia isolar essas partes que se repetem, de maneira que elas possam ser realizadas sem repetir o mesmo código várias vezes. Esse objetivo pode ser atingido com o uso de **funções** e **procedimentos**.

Neste capítulo, vamos estudar as funções e procedimentos como uma forma de reaproveitar o código de tarefas que se repetem em um programa. Funções são importantes para a modularização de um programa, ou seja, para dividir o programa em partes menores, cada parte com uma função específica; sem funções ou procedimentos, um programa maior e mais complicado ficaria organizado em uma única parte que faz tudo e é mais difícil de entender. Neste capítulo, vamos entender quando utilizar as funções ou procedimentos para melhor dividir o código dos nossos programas.

Importante



Neste capítulo, vamos **descartar a utilização de pseudocódigo**. Agora que você já possui um conhecimento básico sobre a linguagem C e, provavelmente, escreveu alguns programas nela, consideramos que não haverá mais necessidade de apresentar sintaxes para pseudocódigos. Portanto, deste capítulo em diante, o conteúdo apresentado utilizará somente a sintaxe da linguagem C.

5.1 O que são funções?

Funções e procedimentos podem ser compreendidos como trechos reutilizáveis de código. Uma função ou procedimento pode, então, ser utilizado várias vezes por um mesmo programa. Isso simplifica a criação de programas maiores, dividindo-os em unidades menores que trabalham em conjunto. Funções e procedimentos são bastante semelhantes e, posteriormente, vamos entender as diferenças entre eles.

Revisão

Lembre-se que durante o curso você já utilizou funções diversas vezes, o que nós sabemos sobre funções até agora?

- Uma função implementa um comportamento que pode ser reutilizado;



- Para executar uma função, utilizamos o nome da função e passamos alguns parâmetros entre parênteses e separados por vírgula. Exemplo: `printf("R$ %1.2f", preco);`
 - A função `printf` é utilizada para imprimir texto na saída;
 - As funções `scanf` e `getchar` para ler dados da entrada;
 - As funções `strlen`, `strcpy`, `strcat` e `strchr` são utilizadas para manipular strings;
 - As funções são agrupadas em módulos. Exemplo: `stdio`;
-

Quando realizamos um processo com um determinado objetivo, é comum identificarmos partes que se repetem. Quando se trata de processos no mundo físico, muitas vezes criamos máquinas que ajudam a realizar as partes que se repetem.

Por exemplo, pense no processo de limpar as roupas que usamos para podermos usá-las novamente. Este processo envolve lavar as roupas com água e sabão, secar as roupas lavadas e, finalmente, passar as roupas para evitar que fiquem amassadas. Para reduzir a quantidade de trabalho necessária para realizar esse processo, foi inventada uma máquina de lavar, que realiza a lavagem das roupas. Uma pessoa que tem uma máquina de lavar pode usá-la repetidamente e, sempre que precisa de roupas limpas, precisa apenas se encarregar de enxugar e passar as roupas; a lavagem fica por conta da máquina. Dizemos inclusive que a **função** da máquina é lavar as roupas.

As funções e os procedimentos funcionam desta forma, capturando partes menores de um algoritmo que podem ser utilizadas por outros algoritmos, economizando o trabalho de sempre refazer uma determinada tarefa.

5.1.1 Um exemplo

Vejamos um exemplo. Queremos criar um programa que calcule a média final de um aluno em uma disciplina com três provas; o programa deve pedir ao usuário que entre com as notas das três provas, calcular a média aritmética das três notas e apresentar o resultado. O programador que resolveu o problema decidiu imprimir um separador na tela entre cada entrada de dados e o resultado final, desta forma, produzindo o seguinte código:

Código fonte code/cap5/media_sem_proc.c

Cálculo da média de um aluno em uma disciplina

```
1  #include <stdio.h>
2
3  int main () {
4      float nota1, nota2, nota3, media;
5
6      printf("Entre a nota da primeira prova: ");
7      scanf("%f", &nota1);
8
9      printf("\n"); // ❶
10     printf("=====\n"); // ❷
11     printf("\n"); // ❸
12
13     printf("Entre a nota da segunda prova: ");
14     scanf("%f", &nota2);
15
16     printf("\n");
17     printf("=====\n");
18     printf("\n");
19
20     printf("Entre a nota da terceira prova: ");
21     scanf("%f", &nota3);
22
23     printf("\n");
24     printf("=====\n");
25     printf("\n");
26
27     media = (nota1 + nota2 + nota3) / 3.0;
28     printf("Media: %f\n", media);
29 }
```

```

30     return 0;
31 }

```

❶, ❷, ❸ Código que imprime um separador na tela.

Resultado da execução do programa

Entre a nota da primeira prova: 7.0

=====

Entre a nota da segunda prova: 8.0

=====

Entre a nota da terceira prova: 9.6

=====

Media: 8.200000

É fácil de notar que o código usado para imprimir um separador se repete três vezes no programa. Podemos melhorar esse programa utilizando um procedimento que terá a tarefa de imprimir o separador. Isso teria o seguinte resultado:

Código fonte code/cap5/media_proc.c

Cálculo da média usando procedimento

```

1  #include <stdio.h>
2
3  void imprime_separador() {    // ❶
4      printf("\n");            // ❷
5      printf("=====\n");      // ❸
6      printf("\n");           // ❹
7  }
8
9  int main () {
10     float nota1, nota2, nota3, media;
11
12     printf("Entre a nota da primeira prova: ");
13     scanf("%f", &nota1);
14
15     imprime_separador();      // ❺
16
17     printf("Entre a nota da segunda prova: ");
18     scanf("%f", &nota2);
19
20     imprime_separador();
21
22     printf("Entre a nota da terceira prova: ");
23     scanf("%f", &nota3);

```

```
24
25     imprime_separador();
26
27     media = (nota1 + nota2 + nota3) / 3.0;
28     printf("Media: %f\n", media);
29
30     return 0;
31 }
```

- ❶ Definição do procedimento `imprime_separador`.
- ❷, ❸, ❹ Código no **corpo** do procedimento.
- ❺ Chamada do procedimento `imprime_separador`.

A linha `void imprime_separador()` inicia a definição do procedimento chamado `imprime_separador`; o **corpo** do procedimento é o conjunto de comandos entre chaves que vem logo após o nome do procedimento. Os parênteses e a palavra `void` no início serão explicados depois. Dizemos que o uso de um procedimento é uma **chamada** ao mesmo. No exemplo anterior, o código do programa chama `imprime_separador` três vezes.

É importante notar que além de economizar a repetição de linhas de código, o programa usando o procedimento `imprime_separador` também é mais fácil de entender porque transmite melhor as partes que compõem o programa. A melhor modularização do código do programa é um motivo extremamente importante para usar funções e procedimentos, principalmente à medida que os programas se tornam maiores e mais complexos.

5.2 Parâmetros

O exemplo usando `imprime_separador` é o caso mais simples, mas menos interessante do uso de procedimentos e funções, quando o código a ser reutilizado é sempre o mesmo. Na maioria das situações de interesse, queremos utilizar uma função ou procedimento em situações com algumas diferenças. Para tornar um procedimento (ou função) mais flexível, é preciso que informações sejam passadas para o procedimento. Isso é feito com o uso de **parâmetros**.

Já vimos muitas vezes o uso de procedimentos com parâmetros. Por exemplo, `printf` é um procedimento da biblioteca padrão da linguagem C que imprime a *string* passada como parâmetro. Assim, `printf("ola, mundo!");`

é uma chamada ao procedimento `printf` com parâmetro `"ola, mundo!"`.

Como exemplo do uso de parâmetros em um procedimento, voltemos ao exemplo do cálculo das médias. Vamos utilizar o `separador` para fazer uma visualização simples das notas e da média, imprimindo uma barra de tamanho proporcional a cada valor.

Código fonte `code/cap5/media_param.c`

Uso de um procedimento com parâmetro


```
1  #include <stdio.h>
2  #include <math.h>
3
4  void imprime_separador(float nota) {    // ❶
5      int i;
6      printf("\n");
7      for (i = 0; i < (int) lround(nota * 5.0); i++) {
8          printf("=");
9      }
10     printf(" %3.2f / 10.0\n", nota);
11     printf("\n");
12 }
13
14 int main () {
15     float nota1, nota2, nota3, media;
16
17     printf("Entre a nota da primeira prova: ");
18     scanf("%f", &nota1);
19
20     imprime_separador(nota1);    // ❷
21
22     printf("Entre a nota da segunda prova: ");
23     scanf("%f", &nota2);
24
25     imprime_separador(nota2);
26
27     printf("Entre a nota da terceira prova: ");
28     scanf("%f", &nota3);
29
30     imprime_separador(nota3);
31
32     media = (nota1 + nota2 + nota3) / 3.0;
33     printf("Media: %3.2f\n", media);
34
35     imprime_separador(media);
36
37     return 0;
38 }
```

- ❶ Definição do procedimento `imprime_separador`, com o parâmetro `nota`, do tipo `float`.
- ❷ Chamada do procedimento `imprime_separador`, passando o argumento `nota1`.

Resultado da execução do programa `code/cap5/media_param.c`

Entre a nota da primeira prova: 6.2

===== 6.20 / 10.0

Entre a nota da segunda prova: 7.8

```
===== 7.80 / 10.0
```

Entre a nota da terceira prova: 9.2

```
===== 9.20 / 10.0
```

Media: 7.73

```
===== 7.73 / 10.0
```

Como se pode ver no resultado da execução do programa, o novo procedimento `imprime_separador` imprime uma barra de tamanho proporcional à nota passada como argumento (são 5 caracteres = para cada ponto da nota). Note a diferença de nomenclatura: o procedimento `imprime_separador` é definido com um **parâmetro** de nome `nota`, mas é chamado com um **argumento**, o valor que é comunicado ao procedimento. No exemplo acima, `imprime_separador` é chamado com os argumentos `nota`, `nota2`, `nota3` e `media`. É necessário especificar o tipo de cada parâmetro que deve ser passado para o procedimento; no caso, o parâmetro `nota` do procedimento `imprime_separador` é do tipo `float`.

Na chamada de um procedimento com um parâmetro, o controle do programa executa o código do procedimento, atribuindo o valor passado como argumento à variável `nota` dentro do procedimento `imprime_separador`. Ou seja, o seguinte código

```
int main() {
    float nota1;

    printf("Entre a nota da primeira prova: ");
    scanf("%f", &nota1);

    imprime_separador(nota1);

    // resto do programa ...
}
```

funciona como se fosse o seguinte (utilizando o procedimento):

```
int main() {
    float nota1;

    printf("Entre a nota da primeira prova: ");
    scanf("%f", &nota1);

    float nota = nota1;    // ❶
    int i;                // ❷
    printf("\n");
    for (i = 0; i < (int) lround(nota * 5.0); i++) {
        printf("=");
    }
    printf(" %3.2f / 10.0\n", nota);
    printf("\n");

    // resto do programa ...
}
```

- ❶ Atribuição do valor do argumento `nota1` para o parâmetro `nota`.
- ❷ Resto do corpo do procedimento `imprime_separador`.

Este exemplo é apenas uma ilustração; na prática, o chamado do procedimento não funciona exatamente desta forma. Em particular, a variável `nota`, que designa o parâmetro do procedimento, só existe enquanto o procedimento executa. Isso será detalhado mais tarde.

O uso de parâmetros nos procedimentos os tornam muito mais flexíveis para uso em diferentes situações. Mas assim como é útil que o código que chama o procedimento comunique informações para o procedimento chamado, muitas vezes também é útil que o procedimento comunique algo de volta para o código que o chamou; neste caso, passamos dos procedimentos para as funções.

5.3 Retorno de Valores com Funções

Até agora só temos usado procedimentos como ferramenta de modularização do código, mas muitas vezes é útil chamar um procedimento que retorna alguma informação de volta para o código que o chamou. Esta é a diferença entre procedimentos e funções: as funções retornam algum valor. Desta forma, as funções em linguagem C são similares às funções da matemática: uma função como

$$f: \mathbb{Z} \rightarrow \mathbb{Z}, \quad f(x) = x^2 + 2$$

tem um parâmetro x (um inteiro), e retorna um determinado valor que depende do parâmetro passado; por exemplo,

$$f(5) = 5^2 + 2 = 27$$

É fácil escrever a mesma função em linguagem C:

```
int f(int x) {           // ❶
    return x * x + 2;    // ❷
}
```

- ❶ Definição da função `f`. A definição começa com `int f(...)`, significando que o tipo de retorno da função é `int`.
- ❷ A palavra-chave `return` determina o valor de retorno da função `f`, que será o resultado da expressão `x * x + 2`.

A função `f` do exemplo faz o mesmo que a versão matemática: dado o valor do parâmetro x , retorna um valor que é igual a x ao quadrado, somado a dois. Note que é preciso especificar o tipo do valor que é retornado pela função e, por isso, toda função começa com o tipo de retorno antes do nome. Especificar os tipos dos parâmetros e o tipo do valor de retorno também é similar às funções na matemática, para as quais devemos especificar os conjuntos domínio e contra-domínio.

5.3.1 Funções, Procedimentos e o Tipo `void`

Neste ponto pode surgir uma pergunta: se é preciso especificar o tipo do valor retornado por uma função antes do nome da função (por exemplo `int f(...)`), por que nos procedimentos usa-se a palavra-chave `void`?

A verdade é que, embora algumas linguagens de programação façam uma distinção entre procedimentos e funções, na linguagem C existem apenas funções. Como a diferença entre procedimentos e funções é apenas o fato de retornar ou não um valor, os procedimentos em C são considerados funções que retornam um valor *vazio*. É isso que significa o `void` no início da definição de um procedimento como `imprime_separador`, que vimos anteriormente; a rigor, `imprime_separador` é uma função, mas retorna um valor *vazio*, ou seja, nenhum valor. O tipo `void` na linguagem C é um tipo especial que denota a ausência de valores.

Como procedimentos em C são funções, também é possível usar `return` em procedimentos, mas apenas para terminar sua execução e retornar imediatamente. Isso às vezes é útil para terminar um procedimento em pontos diferentes do seu final. Também pode-se utilizar `return` ao final do procedimento, mas este uso é supérfluo e não é recomendado.

O seguinte exemplo demonstra o uso do `return` em procedimentos. Continuando no tema relacionado ao cálculo de médias, queremos detectar se uma das notas entradas pelo usuário é uma nota inválida antes de fazer o cálculo da média. Neste caso, o programa deve apenas imprimir se algum valor negativo foi entrado pelo usuário. O procedimento `possui_negativo` será responsável por imprimir uma mensagem caso um dos valores seja negativo.

Código fonte `code/cap5/retorno_proc.c`

Uso de `return` em um procedimento

```
1  #include <stdio.h>
2
3  void possui_negativo(float n1, float n2, float n3) {
4      if (n1 < 0.0) {
5          printf("Numero negativo encontrado!\n");
6          return;    // ❶
7      }
8
9      if (n2 < 0.0) {
10         printf("Numero negativo encontrado!\n");
11         return;
12     }
13
14     if (n3 < 0.0) {
15         printf("Numero negativo encontrado!\n");
16         return;
17     }
18
19     printf("Nenhum numero negativo encontrado\n");    // ❷
20 }
21
22 int main() {
23     float nota1, nota2, nota3;
24
25     printf("Entre as três notas, separadas por espaços: ");
26     scanf("%f %f %f", &nota1, &nota2, &nota3);
27
28     possui_negativo(nota1, nota2, nota3);
29
30     printf("Media: %f\n", (nota1 + nota2 + nota3) / 3.0);
31
32     return 0;
```

33 }

- ❶ Uso do `return` para sair prematuramente do procedimento.
- ❷ Este comando de impressão será executado se nenhuma das condições testadas em `possui_negativo` for verdade, ou seja, se nenhum dos valores dos parâmetros for negativo.

Resultado de duas execuções do programa `code/cap5/retorno_proc.c`

```
Entre as três notas, separadas por espacos: 4.5 5.6 8.9
Nenhum numero negativo encontrado
Media: 6.333333
```

```
Entre as três notas, separadas por espacos: 4.5 -6.7 9.9
Numero negativo encontrado!
Media: 2.566667
```

O procedimento `possui_negativo` deve verificar se um dos três números passados como argumentos, mas basta achar um número entre eles para que o resultado possa ser impresso imediatamente e o procedimento pode retornar; por isso, usamos `return` assim que o primeiro valor negativo é encontrado.

Esse exemplo ainda tem um problema: como pode ser visto nos exemplos de execução, mesmo que o usuário entre um valor negativo, a média aritmética das três notas ainda é impressa na tela (o usuário apenas é avisado que um dos valores foi negativo). Isso é uma indicação que seria melhor que `possui_negativo` fosse uma função, e que o programa principal verificasse o valor retornado e tomasse uma decisão. Se fizermos essas alterações ficamos com o seguinte programa:

Código fonte `code/cap5/retorno_func.c`

Reescrevendo o exemplo anterior para usar uma função

```
1  #include <stdio.h>
2
3  int possui_negativo(float n1, float n2, float n3) { // ❶
4      if (n1 < 0.0 || n2 < 0.0 || n3 < 0.0) // ❷
5          return 1; // ❸
6
7      return 0; // ❹
8  }
9
10 int main() {
11     float nota1, nota2, nota3;
12
13     printf("Entre as três notas, separadas por espacos: ");
14     scanf("%f %f %f", &nota1, &nota2, &nota3);
15
16     if (possui_negativo(nota1, nota2, nota3) == 1) // ❺
17         printf("Nao e' possivel calcular a media, uma ou mais ↵
18             notas sao negativas\n");
19     else
20         printf("Media: %f\n", (nota1 + nota2 + nota3) / 3.0);
```

```
21     return 0;  
22 }
```

- ❶ A função `possui_negativo` agora retorna um inteiro de valor 1 caso um dos valores dos parâmetros seja negativo, e 0 caso contrário (todos são positivos).
- ❷ Teste para identificar se um ou mais dos parâmetros informados são negativos.
- ❸ A função retorna 1 se um dos números passados para a função for negativo.
- ❹ Caso nenhum dos números seja negativo, o controle passa para o comando `return` ao final da função e o valor 0 é retornado para indicar que nenhum número negativo foi encontrado.
- ❺ O programa principal verifica o valor de retorno da função `possui_negativo` e imprime informações adequadas na tela para cada caso.

Resultado de duas execuções do programa `code/cap5/retorno_func.c`

```
Entre as três notas, separadas por espacos: 6.8 9.7 -2.3  
Nao e' possivel calcular a media, uma ou mais notas sao ←  
negativas
```

```
Entre as três notas, separadas por espacos: 6.8 9.7 7.2  
Media: 7.900000
```

Como pode-se ver nos dois exemplos de execução do programa, a saída agora é mais adequada. Caso uma das notas informadas seja negativa, o programa não imprime um valor de média, apenas avisando o usuário do erro na entrada de dados. O código da função `possui_negativo` também foi simplificado pelo uso do operador lógico OU.

Como funções e procedimentos são tratados de maneira uniforme na linguagem C (e em muitas outras linguagens atuais), a partir de agora vamos usar o termo **função** tanto para funções como para procedimentos. Isso não deve gerar nenhuma confusão. Em caso de dúvida, basta olhar para a definição da função no código-fonte do programa; se a função for declarada com o tipo de retorno `void`, então é um procedimento.

5.4 Um Exemplo Matemático: Equação de Segundo Grau

Nesta seção, veremos mais um exemplo do uso de funções em um programa para calcular as raízes de uma equação de segundo grau. Neste exemplo, as funções não serão utilizadas várias vezes, mas o programa principal será mais claro e mais fácil de entender graças à melhor modularização conseguida com o uso das funções.

Lembremos que um *polinômio de segundo grau* é uma soma de três termos em potências de uma variável, por exemplo

$$P(x) = ax^2 + bx + c$$

onde a , b e c são coeficientes constantes. Uma *equação de segundo grau* é formada ao igualar um polinômio de segundo grau a zero, ou seja

$$ax^2 + bx + c = 0$$

Sabemos que uma equação de segundo grau pode ter até duas **raízes** reais; cada raiz é um valor da variável x que satisfaz a equação. Essas raízes podem ser encontradas pela chamada *fórmula de Bhaskara*. A fórmula consiste em calcular um valor auxiliar chamado de Δ (*delta*), e usar o valor calculado para identificar quantas raízes reais distintas podem ser encontradas para a equação:

- se $\Delta < 0$, a equação não tem raízes reais;
- se $\Delta = 0$, a equação possui uma raiz real;
- se $\Delta > 0$, a equação possui duas raízes reais distintas.

A fórmula para calcular Δ é

$$\Delta = b^2 - 4ac$$

No caso $\Delta \geq 0$, as raízes da equação são dadas por

$$x = \frac{-b \pm \sqrt{\Delta}}{2a}$$

uma das raízes sendo obtida pelo uso do sinal positivo em $-b \pm \sqrt{\Delta}$, enquanto que a outra raiz é calculada pelo uso do sinal negativo. Se $\Delta = 0$, ambos os valores serão iguais.

Como exemplo do uso da fórmula de Bhaskara, considere a equação:

$$x^2 - 5x + 6 = 0$$

Nesta equação os coeficientes são $a = 1$, $b = -5$ e $c = 6$. Calculamos o Δ usando os valores dos coeficientes:

$$\Delta = b^2 - 4ac = (-5)^2 - 4 \times 1 \times 6 = 25 - 24 = 1$$

E assim, podemos calcular as raízes:

$$x = \frac{-b \pm \sqrt{\Delta}}{2a} = \frac{5 \pm \sqrt{1}}{2} = \frac{5 \pm 1}{2}$$

Ou seja, as raízes são $6/2 = 3$ e $4/2 = 2$.

Agora vamos escrever um programa que resolve equações do segundo grau, usando o processo mostrado acima.

Código fonte code/cap5/equacao.c

Cálculo de raízes de uma equação de segundo grau

```
1 #include <stdio.h>
2 #include <math.h>
3
4 float calculo_delta(float a, float b, float c) { // ❶
5     return (b * b) - (4 * a * c);
6 }
7
8 float raiz_positiva(float a, float b, float delta) { // ❷
9     return (-b + sqrt(delta)) / 2 * a;
10 }
11
12 float raiz_negativa(float a, float b, float delta) { // ❸
13     return (-b - sqrt(delta)) / 2 * a;
```

```
14 }
15
16 int main() {
17     float a, b, c;
18     float delta;
19
20     printf("Entre os coeficientes A, B e C, nesta ordem: ");
21     scanf("%f %f %f", &a, &b, &c);
22
23     delta = calculo_delta(a, b, c);
24
25     if (delta < 0.0)
26         printf("Delta negativo, nao existem raizes\n");
27     else if (delta == 0)
28         printf("Delta = 0, uma raiz: %f\n", raiz_positiva(a, b ↵
29             , delta));
30     else
31         printf("Raizes: %f e %f\n", raiz_positiva(a, b, delta) ↵
32             , raiz_negativa(a, b, delta));
33
34     return 0;
35 }
```

- ❶ Função que calcula o valor do Δ .
- ❷ Função que calcula a raiz positiva da equação.
- ❸ Função que calcula a raiz negativa da equação.

Resultado de três execuções do programa code/cap5/equacao.c

```
Entre os coeficientes A, B e C, nesta ordem: 1.0 -5.0 6.0
Raizes: 3.000000 e 2.000000
```

```
Entre os coeficientes A, B e C, nesta ordem: 1.0 -6.0 9.0
Delta = 0, uma raiz: 3.000000
```

```
Entre os coeficientes A, B e C, nesta ordem: 4.0 -3.0 7.0
Delta negativo, nao existem raizes
```

Podemos ver neste exemplo funções para calcular o valor do Δ e das duas raízes da equação. O programa obtém o valor do Δ e verifica se a equação tem nenhuma, uma ou duas raízes, e imprime o resultado de acordo com isso. Embora cada função seja usada apenas uma vez, o programa principal é mais claro e mais fácil de entender porque cada função faz uma parte do processo necessário, ao invés de ter todo o código junto na função `main`. Funções são importantes não só para reutilizar código e diminuir esforço de programação, mas também para melhorar a modularização do programa e torná-lo mais fácil de ser lido. Em situações práticas, muitas vezes é necessário ler um código que já foi produzido antes e entendê-lo, seja para consertar defeitos encontrados ou para estender suas funcionalidades. Tornar um programa mais legível auxilia e reduz o custo relacionado à *manutenção* do mesmo.

Entretanto, este último exemplo pode parecer estranho do ponto de vista da modularização, já que duas de suas funções são quase idênticas. As funções que calculam o valor das raízes, `raiz_positiva` e `raiz_negativa`, mudam apenas em uma operação. Podemos pensar em como reescrever o programa para usar apenas uma função ao invés de duas funções quase idênticas. A repetição desnecessária de código pode ser um problema para a manutenção de um programa.

A chave para criar uma só função que calcula os dois valores é criar um novo parâmetro que indica qual das duas raízes deve ser calculada. Vamos usar um parâmetro chamado `sinal` que indica, pelo seu valor, se será usada uma soma ou subtração no cálculo da raiz. Se `sinal` for 1, será usada uma soma, e se for `-1` será usada uma subtração. O código resultante é mais compacto e evita repetições:

Código fonte code/cap5/equacao2.c

Cálculo de raízes de uma equação de segundo grau

```
1  #include <stdio.h>
2  #include <math.h>
3
4  float calculo_delta(float a, float b, float c) {
5      return (b * b) - (4 * a * c);
6  }
7
8  float raiz(float a, float b, float delta, int sinal) {
9      if (sinal == 1)
10         return (-b + sqrt(delta)) / 2 * a;
11     else
12         return (-b - sqrt(delta)) / 2 * a;
13 }
14
15 int main() {
16     float a, b, c;
17     float delta;
18
19     printf("Entre os coeficientes A, B e C, nesta ordem: ");
20     scanf("%f %f %f", &a, &b, &c);
21
22     delta = calculo_delta(a, b, c);
23
24     if (delta < 0.0)
25         printf("Delta negativo, nao existem raizes\n");
26     else if (delta == 0)
27         printf("Delta = 0, uma raiz: %f\n", raiz(a, b, delta, ↵
28             1));
29     else
30         printf("Raizes: %f e %f\n", raiz(a, b, delta, 1), raiz ↵
31             (a, b, delta, -1));
32
33     return 0;
34 }
```

É comum quando escrevemos programas nos concentrarmos, inicialmente, em fazê-lo funcionar, ou seja, resolver o problema desejado. Entretanto, é importante depois reler o código escrito e revisá-lo para torná-lo mais claro, mais legível e mais fácil de manter. Esse processo de revisar o código de um programa sem mudar sua funcionalidade é muito importante na programação de computadores, e normalmente recebe o nome de **refatoração**.

5.5 Escopo de Variáveis

Quando trabalhamos com programas compostos por várias funções, nos deparamos com questões relativas à *visibilidade* das variáveis em diferentes partes do programa. Ou seja, se uma variável é visível ou acessível em certas partes de um programa.

Um programador iniciante poderia escrever o seguinte programa para calcular a média aritmética de três notas:

Código fonte code/cap5/media_erro.c

Cálculo da média usando código incorreto

```
1  #include <stdio.h>
2
3  float calc_media() {
4      return (nota1 + nota2 + nota3) / 3.0;  // ❶
5  }
6
7  int main() {
8      float nota1, nota2, nota3;
9
10     printf("Entre as três notas: ");
11     scanf("%f %f %f", &nota1, &nota2, &nota3);
12
13     printf("Média: %f\n", calc_media());
14
15     return 0;
16 }
```

❶ Esta linha contém erros e o programa não será compilado.

O raciocínio do programador é "se as variáveis `nota1`, `nota2` e `nota3` existem na função `main` e a função `calc_media` é chamada dentro de `main`, as variáveis `nota1`, `nota2` e `nota3` não deveriam ser visíveis dentro de `calc_media`?"

Acontece que isso não é válido na linguagem C, e qualquer compilador da linguagem vai acusar erros de compilação neste programa, avisando que as variáveis `nota1`, `nota2` e `nota3` não foram declaradas.

Para entender como funciona a visibilidade das variáveis em um programa na linguagem C, precisamos falar sobre as regras de **escopo** desta linguagem. O escopo de uma variável é a parte do programa na qual ela é visível e pode ser acessada.

A linguagem C usa um conjunto de regras de escopo que recebe o nome de *escopo estático* ou *escopo léxico*. Essas regras são bastante simples de entender e aplicar, como veremos a seguir.

Em programas na linguagem C existem dois tipos de escopo (regiões de visibilidade):

- escopo global;
- escopos locais.

Existe apenas um escopo global e, como indicado pelo seu nome, ele contém elementos que são visíveis em todo o programa. Já os escopos locais são vários e particulares: basicamente, cada função define um escopo local que corresponde com o *corpo* da função.

Desta forma, variáveis declaradas no escopo global (ou seja, "fora" de qualquer função) são visíveis em todo programa, enquanto variáveis declaradas dentro de uma função são visíveis apenas dentro da mesma função. No exemplo anterior, as variáveis `nota1`, `nota2` e `nota3` são visíveis apenas dentro da função `main`, e por isso não podem ser acessadas dentro da função `calc_media`.

Isso pode ser resolvido mudando as variáveis `nota1`, `nota2` e `nota3` para o escopo global, ou seja, tornando-as *variáveis globais*, como no seguinte programa:

Código fonte `code/cap5/media_globais.c`

Cálculo de raízes de uma equação de segundo grau

```
1 #include <stdio.h>
2
3 float nota1, nota2, nota3; // ❶
4
5 float calc_media() {
6     return (nota1 + nota2 + nota3) / 3.0; // ❷
7 }
8
9 int main() {
10     printf("Entre as três notas: ");
11     scanf("%f %f %f", &nota1, &nota2, &nota3); // ❸
12
13     printf("Media: %f\n", calc_media());
14
15     return 0;
16 }
```

- ❶ Declaração das variáveis `nota1`, `nota2` e `nota3` como variáveis globais. Note que elas estão declaradas "fora" de qualquer função.
- ❷ Código dentro de `calc_media` que usa as variáveis globais. Neste programa, as variáveis estão visíveis e não ocorrerá um erro durante a compilação.
- ❸ Código dentro de `main` que usa as variáveis globais. Variáveis globais são visíveis em todo o programa, incluindo na função principal.

Este programa agora compila corretamente e funciona para o cálculo da média. Mas é importante observar que esse tipo de prática *não é recomendada*. Entre as boas práticas da programação está a sugestão de usar variáveis globais apenas quando absolutamente necessário. Como variáveis globais podem ser acessadas e ter seu valor alterado por qualquer parte do programa, fica difícil saber que partes podem influenciar ou serem influenciadas pelas variáveis globais, o que torna todo o programa mais difícil de entender. Para o exemplo das notas, é melhor e mais de acordo com boas práticas de programação comunicar as notas para a função `calc_media` usando parâmetros, como segue:

Código fonte `code/cap5/media_param2.c`

Cálculo de raízes de uma equação de segundo grau

```
1 #include <stdio.h>
2
3 float calc_media(float n1, float n2, float n3) {
4     return (n1 + n2 + n3) / 3.0;
5 }
6
7 int main() {
8     float nota1, nota2, nota3;
9
10    printf("Entre as três notas: ");
11    scanf("%f %f %f", &nota1, &nota2, &nota3);
12
13    printf("Media: %f\n", calc_media(nota1, nota2, nota3));
14
15    return 0;
16 }
```

Este código funciona corretamente e evita o uso desnecessário de variáveis globais.

5.5.1 Escopo dos Parâmetros

Uma pergunta que pode surgir (especialmente após o exemplo anterior) é “qual o escopo dos parâmetros das funções?” A resposta é simples: para questões de visibilidade, o escopo dos parâmetros das funções é o escopo local da função da qual eles pertencem. Ou seja, os parâmetros de uma função funcionam exatamente como variáveis locais declaradas dentro da função.

5.5.2 Sombreamento e Sobreposição de Escopos

O que acontece se duas variáveis tiverem o mesmo nome em um só programa? A resposta depende de onde as variáveis em questão são declaradas.

Não podem existir duas variáveis de mesmo nome em um mesmo escopo; um programa que tente declarar duas variáveis de mesmo nome no mesmo escopo ocasionará um erro quando for compilado. Assim, não podem existir duas variáveis globais de nome *x* ou duas variáveis de nome *y* em uma mesma função.

Em escopos diferentes a regra muda: variáveis de mesmo nome podem existir em um programa se forem declarados em escopos distintos. Isso é bastante útil: imagine um programa com 20 ou 30 mil linhas de código (o que hoje em dia é considerado um programa de *pequeno a médio* porte); um programa deste tamanho precisa usar um grande número de variáveis, se cada uma delas precisasse ter um nome diferente de todas as outras, seria muito difícil dar nomes a vários milhares de variáveis. Imagine que um programa deste tamanho pode ter mais de mil laços `for`, cada um com uma variável de controle, e cada uma dessas variáveis teria que ter um nome diferente. Por isso, as regras de escopo também são úteis para estabelecer espaços locais onde os nomes não entram em conflitos com os nomes de outros escopos locais.

Quando temos duas variáveis de mesmo nome em diferentes escopos locais, ou seja, duas funções diferentes, o resultado é simples, já que essas variáveis de mesmo nome nunca seriam visíveis no mesmo local do programa. Mas e se tivermos duas variáveis de mesmo nome, sendo uma variável local e uma global? Neste caso, dentro da função que declara a variável com mesmo nome da global,

existirão duas variáveis que poderiam ser visíveis com o mesmo nome. O que acontece nesses casos é chamado de **sombreamento**: a variável do escopo local *esconde* a variável do escopo global. Vamos ilustrar essa regra com um exemplo:

Código fonte code/cap5/sombra.c

Exemplo de sombreamento de variáveis globais

```
1  #include <stdio.h>
2
3  int x = 5;           // ❶
4
5  void f() {
6      int x = 60;      // ❷
7      int y = x * x;    // ❸
8
9      printf("x = %d, y = %d\n", x, y);
10 }
11
12 int g() {
13     int y = x * x;    // ❹
14
15     return y;
16 }
17
18 int main() {
19     f();
20
21     printf("g = %d\n", g());
22
23     return 0;
24 }
```

- ❶ Declaração da variável global `x`.
- ❷ Declaração da variável local `x` na função `f`.
- ❸ Declaração da variável local `y` na função `f`.
- ❹ Declaração da variável local `y` na função `g`.

Vemos no exemplo que existe uma variável global chamada `x` e uma variável local `x` na função `f`. A função `f` também tem uma variável local chamada `y`, e há uma variável local de mesmo nome na função `g`. As variáveis chamadas `y` em `f` e `g` não interferem, pois são escopos totalmente diferentes.

Já as variáveis chamadas `x` interferem, já que uma está no escopo global e outra está no escopo local da função `f`. A questão é: o que é impresso pelo programa? Isso depende dos valores de `x` dentro da função `f` e na função `g` (que usa `x` para calcular o valor de `y`, que é retornado). A execução do programa imprime o seguinte:

Resultado da execução do programa code/cap5/sombra.c

```
x = 60, y = 3600
g = 25
```

A primeira linha é o que é impresso na função `f`. Como existe uma variável local `x` declarada em `f`, dentro da função `f` a variável `x` tem o valor 60, como declarado; o valor de `y` calculado em `f` é, então, $60 \times 60 = 3600$. Já na função `g` não existe uma variável `x` local, então o valor de `x` dentro de `g` é o valor da variável global `x`, que é igual a 5; desta forma, `y` em `g` tem valor $5 \times 5 = 25$. Isso explica a saída do programa como visto acima.

**Nota**

Uma consequência da regra de sombreamento é que dentro de funções que tenham variáveis locais que escondem variáveis globais de mesmo nome, é impossível acessar ou utilizar as variáveis globais escondidas. No exemplo anterior, dentro da função `f` é impossível ter acesso à variável global `x`.

5.6 Passagem de Parâmetros

Com o que vimos até agora sobre parâmetros de funções, eles funcionam de maneira simples: o código que chama uma função especifica expressões para cada um dos argumentos da função. Os valores de cada expressão são calculados e transmitidos como o valor dos parâmetros declarados na função.

Entretanto, isso não é suficiente para todos os casos em que podemos querer usar parâmetros. Por exemplo, digamos que temos uma situação em que é necessário trocar o valor de duas variáveis, e que isso é necessário várias vezes ao longo do programa. Para evitar repetição, a melhor solução é escrever uma função que realiza a troca do valor de duas variáveis. O exemplo a seguir mostra o que acontece quando tentamos fazer isso apenas com o que vimos até agora:

Código fonte `code/cap5/troca.c`

Tentativa de trocar o valor de duas variáveis usando uma função

```
1  #include <stdio.h>
2
3  void troca_variaveis(int a, int b) {
4      int temp = a;
5      a = b;
6      b = temp;
7
8      printf("Dentro de troca_variaveis: a = %d, b = %d\n", a, b);
9  }
10
11 int main() {
12     int x = 5;
13     int y = 7;
14
15     printf("Antes da troca: x = %d, y = %d\n", x, y);
16     troca_variaveis(x, y);
17     printf("Depois da troca: x = %d, y = %d\n", x, y);
18 }
```

Resultado da execução do programa

Antes da troca: x = 5, y = 7

Dentro de `troca_variaveis`: `a = 7, b = 5`
Depois da troca: `x = 5, y = 7`

Como se vê no resultado da execução do programa, embora as variáveis `a` e `b` realmente troquem de valor dentro da função `troca_variaveis`, isso não afeta o valor das variáveis `x` e `y` em `main`. Isso acontece porque, normalmente, os parâmetros em C são passados **por valor**, ou seja, apenas o valor de `x` e `y` são copiados para `a` e `b`. Precisamos que as variáveis na função `troca_variaveis`, de alguma maneira, afetem as variáveis que foram usadas como parâmetro, e para isso é necessário usar o modo de passagem de parâmetros chamado de passagem **por referência**. A seguir, vamos ver em maiores detalhes como funcionam esses dois modos de passagem de parâmetros.

5.6.1 Passagem por Valor

A passagem de parâmetros por valor é a situação padrão na linguagem C. Este modo de passagem de parâmetros comunica apenas valores entre o código chamador e a função chamada.

A passagem por valor funciona da seguinte forma: para uma função `f` com `N` parâmetros, uma chamada de `f` deve conter `N` expressões como argumentos (se o número de argumentos não corresponder ao número de parâmetros declarados, o compilador acusará um erro no programa). Então o seguinte processo de chamada de função acontece:

1. O valor de cada uma das `N` expressões usadas como argumento é calculado e guardado;
2. `N` variáveis locais são criadas para a função chamada, uma para cada parâmetro da função, e usando o nome declarado na função;
3. Os valores calculados no passo 1 são atribuídos às variáveis criadas no passo 2.
4. O corpo da função `f` é executado.

Como as variáveis criadas para os parâmetros são locais, elas deixam de existir quando a função chamada termina, e isso não tem nenhum efeito nas expressões que foram usadas para atribuir valor aos parâmetros ao início da função. Isso significa que o programa para troca de valor de variáveis mostrado acima funciona de maneira similar ao seguinte programa (no qual colocamos o código da função `troca_variaveis` diretamente na função `main`):

Código fonte `code/cap5/troca_main.c`

Troca do valor de duas variáveis usando outra variável temporária

```
#include <stdio.h>

int main() {
    int x = 5;
    int y = 7;

    printf("Antes da troca: x = %d, y = %d\n", x, y);

    // troca_variaveis
    int a = x, b = y;
    int temp = a;
    a = b;
    b = temp;
```

```
printf("Dentro de troca_variaveis: a = %d, b = %d\n", a, b);  
// fim de troca_variaveis  
  
printf("Depois da troca: x = %d, y = %d\n", x, y);  
}
```

Neste caso, fica claro que as variáveis `x` e `y` são usadas apenas para obter o valor inicial das variáveis `a` e `b`, e portanto a mudança de valor das duas últimas não deve afetar `x` e `y`.

A passagem de parâmetros por valor é simples e funciona bem na maioria dos casos. Mas em algumas situações pode ser desejável ter uma forma de afetar variáveis externas à uma determinada função e, para isso, usa-se a passagem de parâmetros por referência.

5.6.2 Passagem por Referência

A passagem de parâmetros por referência funciona passando para a função chamada *referências* para variáveis ao invés de valores de expressões. Isso permite à função chamada afetar as variáveis usadas como argumento para a função.

Vamos ilustrar como isso funciona demonstrando como criar uma função que troca o valor de duas variáveis e realmente funciona:

Código fonte code/cap5/troca_ref.c

Função para trocar o valor de duas variáveis usando passagem por referência

```
#include <stdio.h>  
  
void troca_variaveis(int *a, int *b) { // ❶  
    int temp = *a;  
    *a = *b; // ❷  
    *b = temp;  
  
    printf("Dentro de troca_variaveis: a = %d, b = %d\n", *a, *b);  
}  
  
int main() {  
    int x = 5;  
    int y = 7;  
  
    printf("Antes da troca: x = %d, y = %d\n", x, y);  
  
    troca_variaveis(&x, &y); // ❸  
  
    printf("Depois da troca: x = %d, y = %d\n", x, y);  
}
```

- ❶ Definição do procedimento. Os parâmetros `a` e `b` são declarados usando `int *` ao invés de simplesmente `int`. Isso indica passagem por referência.
- ❷ Ao usar as variáveis `a` e `b` que foram passadas por referência, é necessário usar `*a` e `*b` para acessar ou modificar seu valor.

- ❸ Na chamada da função `troca_variaveis` é preciso passar referências para as variáveis `x` e `y`, isso é conseguido usando `&x` e `&y`.

Resultado da execução do programa `code/cap5/troca_ref.c`

Antes da troca: `x = 5, y = 7`
Dentro de `troca_variaveis`: `a = 7, b = 5`
Depois da troca: `x = 7, y = 5`

A primeira coisa a notar é que são necessárias algumas mudanças sintáticas para usar parâmetros por referência. A declaração dos parâmetros no início da função agora define os parâmetros `a` e `b` como tendo tipo `int *`. Quando esses parâmetros são usados na função `troca_variaveis`, eles precisam ser precedidos de asterisco (`*a` ao invés de `a`). E para chamar a função, é preciso passar referências para as variáveis `x` e `y` ao invés de passar seu valor, por isso usamos `&x` e `&y` na chamada.

De maneira simplificada, a passagem por referência funciona da seguinte forma: ao escrever um argumento como `&x` para a função `troca_variaveis`, não estamos passando o *valor* de `x` para a função, mas sim uma *referência* para a própria variável `x`. Isso significa que, dentro de `troca_variaveis`, o parâmetro `a` se torna um nome diferente para a mesma variável `x`; desta forma, alterações feitas em `a` (através da sintaxe `*a`) são alterações também no valor da variável original `x`. É por isso que o programa acima funciona, como pode ser visto no resultado da execução: a função `troca_variaveis` recebe referências para as variáveis `x` e `y`, e por isso pode alterar o valor destas variáveis diretamente, trocando o valor das duas.

A passagem de parâmetros por referência é usada quando uma função precisa alterar o valor de uma variável que existe fora da própria função e que não necessariamente é uma variável global. O mesmo efeito de ter uma função alterando uma variável externa poderia ser atingido usando variáveis globais ao invés de passagem por referência, mas com grande perda de flexibilidade. Além disso, o uso desnecessário de variáveis globais não é recomendado, como comentado antes.

5.6.2.1 Realizando troca de valores com variáveis globais

Uma outra forma de trocar os valores de duas variáveis, dentro de uma função, poderia ser elaborada utilizando variáveis globais. Por exemplo, poderíamos trocar os valores das variáveis `x` e `y` no exemplo anterior se ambas fossem alteradas para serem variáveis globais:

Código fonte `code/cap5/troca_glob.c`

Função para trocar o valor de duas variáveis globais

```
#include <stdio.h>

int x = 5; // ❶
int y = 7;

void troca_variaveis() { // ❷
    int temp = x;
    x = y;
    y = temp;

    printf("Dentro de troca_variaveis: x = %d, y = %d\n", x, y);
}
```

```
int main() {  
    printf("Antes da troca: x = %d, y = %d\n", x, y);  
    troca_variaveis();  
    printf("Depois da troca: x = %d, y = %d\n", x, y);  
}
```

- ❶ `x` e `y` agora são variáveis globais.
- ❷ `troca_variaveis` não utiliza parâmetros, já que acessa diretamente as variáveis globais.

Resultado da execução do programa

Antes da troca: `x = 5, y = 7`
Dentro de `troca_variaveis`: `x = 7, y = 5`
Depois da troca: `x = 7, y = 5`

O programa funciona, mas note que agora `troca_variaveis` só altera o valor de duas variáveis específicas, `x` e `y`, enquanto que a versão usando passagem por referência era geral e podia trocar o valor de quaisquer duas variáveis inteiras. Se um programa precisa trocar os valores de vários pares de variáveis, em vários locais diferentes, seria preciso criar uma função de troca para cada par, e fazer todas as variáveis serem globais. Isso acarretaria em muita repetição, e muito uso desnecessário de variáveis globais que tornariam a manutenção do código muito mais difícil. Neste caso, é muito mais recomendado usar a passagem por referência para chegar a um código mais geral, mais fácil de manter e com menos repetição.

Nota



A passagem por referência vem sendo usada neste livro há vários capítulos, pois a função `scanf` (Seção 2.10.2 [28]) usa esse modo de passagem de parâmetros. Em toda chamada a `scanf` passamos referências para as variáveis que vão receber os valores, e não os valores dessas variáveis. Isso faz sentido já que `scanf` precisa alterar o valor das variáveis passadas como parâmetro, e ao mesmo tempo `scanf` não utiliza o valor original dessas variáveis para nada.

Nota



Com a passagem por referência e por valor na linguagem C acontece algo semelhante ao que vimos com os conceitos de procedimento e função: a rigor na linguagem C só existe a passagem por valor, mas a passagem por referência pode ser obtida pelo uso de **ponteiros**, um conceito avançado da linguagem C. Como se trata de um conceito avançado, não vamos detalhar mais sobre eles aqui neste livro.

5.7 Protótipos e Declaração de Funções

Em todos os exemplos que vimos neste capítulo até agora, nós sempre definimos uma função antes de chamá-la em outra função. Nesses exemplos, a função `main` sempre aparece no final do arquivo, já que as chamadas para as outras funções apareciam apenas em `main`.

Mas em muitos casos pode ser necessário chamar uma função que é definida posteriormente no arquivo, sem precisar mudar as funções de lugar. O mesmo ocorre em programas maiores, quando usamos vários arquivos de código-fonte para um programa (mas este caso não será detalhado aqui).

Se tentarmos usar uma função antes de sua definição vamos observar um erro de compilação. No exemplo abaixo, a intenção é definir uma função que imprime a situação de um aluno em uma disciplina cuja média é 7.0. Para isso é necessário passar para a função as três notas do aluno na disciplina. Mas se definirmos a função após `main`, como abaixo:

Código fonte code/cap5/situacao.c

Chamando uma função antes de sua definição

```
#include <stdio.h>

int main() {
    float nota1, nota2, nota3;

    printf("Entre as três notas: ");
    scanf("%f %f %f", &nota1, &nota2, &nota3);

    situacao(nota1, nota2, nota3); // ❶
    return 0;
}

void situacao(float n1, float n2, float n3) { // ❷
    float media = (n1 + n2 + n3) / 3.0;

    printf("Media %f, ", media);
    if (media >= 7.0)
        printf("Aluno aprovado\n");
    else if (media < 4.0)
        printf("Aluno reprovado por media\n");
    else
        printf("Aluno na prova final");
}
```

❶ A função `main` chama `situacao` antes de sua definição.

❷ A definição de `situacao` começa após a função `main`.

Ao tentar compilar esse programa, um erro similar ao seguinte será acusado:

Resultado da execução do programa

```
situacao.c:12:6: error: conflicting types for 'situacao'
void situacao(float n1, float n2, float n3) { // ❶
```

Isso acontece porque o compilador não pode identificar se a função foi realmente definida em algum lugar e, principalmente, se o tipo dos parâmetros e o tipo do retorno da função estão sendo usados corretamente.

Esse problema pode ser resolvido através do uso de **protótipos** para declarar uma função antes que seja definida. Um protótipo de função é uma declaração que especifica as seguintes informações:

1. O tipo de retorno da função;
2. O nome da função;
3. O número de parâmetros;
4. Os tipos de cada um dos parâmetros.

Com essas informações, o compilador pode identificar se a função está sendo usada de maneira correta com relação aos tipos, e evita os erros anteriores.

Para consertar o erro no exemplo anterior basta adicionar uma linha com o protótipo da função:

Código fonte code/cap5/situacao_prot.c

Usando protótipos de funções

```
#include <stdio.h>

// Prototipo da funcao situacao
void situacao(float, float, float); // ❶

int main() {
    float nota1, nota2, nota3;

    printf("Entre as três notas: ");
    scanf("%f %f %f", &nota1, &nota2, &nota3);

    situacao(nota1, nota2, nota3); // ❷
}

void situacao(float n1, float n2, float n3) {
    float media = (n1 + n2 + n3) / 3.0;

    printf("Media %f, ", media);
    if (media >= 7.0)
        printf("Aluno aprovado\n");
    else if (media < 4.0)
        printf("Aluno reprovado por media\n");
    else
        printf("Aluno na prova final");
}
```

- ❶ Protótipo da função `situacao`. Note que o protótipo inclui o tipo de retorno (`void`), o nome da função, que a função aceita três parâmetros, e que todos eles possuem tipo `float`. Não é preciso especificar o nome dos parâmetros em um protótipo, mas é possível especificar os nomes, se desejado; incluir os nomes dos parâmetros pode ser útil como uma forma simples de documentação.

- ❶, ❷ A chamada a `situacao` em `main` agora pode acontecer sem problema.

Resultado da execução do programa

```
Entre as três notas: 6.8 5.2 9.0
Media 7.000000, Aluno aprovado
```

Nesse exemplo, seria possível consertar o problema simplesmente movendo a função `situacao` completa para antes da função `main`, mas em programas maiores o uso de protótipos toma grande importância.

5.8 Funções Recursivas

Uma função pode ser chamada a partir de qualquer outra função no mesmo programa. Dessa forma, podemos pensar em uma função chamando a si mesma. Isso é possível e útil em muitos casos. Quando uma função `f` chama a si mesma em algum ponto, dizemos que `f` é uma **função recursiva**. *Recursividade* se refere a um objeto auto-referente, ou seja, que referencia a si próprio; isso inclui as funções recursivas mas também outros tipos de objetos.

Um exemplo é o cálculo do fatorial de um número. O *fatorial* de um número inteiro positivo N (cuja notação é $N!$) é definido como o produto de todos os números de 1 até N :

$$N! = N \times (N - 1) \times \cdots \times 2 \times 1$$

Por exemplo, o fatorial de 3 é $3! = 3 \times 2 \times 1 = 6$, e o fatorial de 4 é $4! = 4 \times 3 \times 2 \times 1 = 24$, e assim por diante. Para números N maiores que 1, sempre podemos fazer a seguinte relação:

$$N! = N \times (N - 1)!$$

ou seja, $4! = 4 \times 3! = 4 \times 6 = 24$. Isso indica a estrutura recursiva do cálculo do fatorial: o fatorial de um número N é igual a N vezes o fatorial do número anterior a N , $N-1$.

Seguindo essa ideia, podemos escrever uma função recursiva que calcula o valor do fatorial de um número. Como a função chama a si mesma, é muito importante saber quando a cadeia de chamadas da função a ela mesma termina. No caso do fatorial, quando queremos calcular o fatorial do número 1 ou do número 0, podemos responder diretamente: $1! = 0! = 1$, sem usar a recursividade. Esse é o chamado **caso-base** da recursão. Sem isso, a função nunca pararia de chamar a si mesma, e o programa seria finalizado com algum erro dependente do sistema operacional e do compilador utilizados.

O programa com a função que calcula o fatorial é o seguinte:

Código fonte `code/cap5/fatorial.c`

Cálculo do fatorial usando uma função recursiva

```
#include <stdio.h>

// prototipo
int fatorial(int);    // ❶

int main() {
    int n;

    printf("Entre o numero para calcular o fatorial: ");
    scanf("%d", &n);

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

    return 0;
}
```

```
int fatorial(int n) {  
    if (n == 0 || n == 1)  
        return 1; // ❷  
    else  
        return n * fatorial(n - 1); // ❸  
}
```

- ❶ Protótipo da função fatorial.
- ❷ Caso-base na função fatorial: para n igual a 0 ou 1, retorna 1.
- ❸ Caso recursivo na função fatorial: se n for maior que 1, retorne n multiplicado pelo fatorial de $n - 1$.

Resultado de três execuções do programa code/cap5/fatorial.c

```
Entre o numero para calcular o fatorial: 4  
4! = 24
```

```
Entre o numero para calcular o fatorial: 5  
5! = 120
```

```
Entre o numero para calcular o fatorial: 6  
6! = 720
```

O código calcula o fatorial de um número inteiro recursivamente. Se o número for 0 ou 1, a resposta é direta: 1. Se o número for maior do que 0 ou 1, a resposta é obtida pela multiplicação do número pelo fatorial do número anterior a ele. Note que esse processo sempre termina se o parâmetro n da função fatorial for um inteiro positivo, pois em cada chamada recursiva o argumento da função fatorial será um número menor que o anterior, até eventualmente chegar a 1, cuja resposta é direta. Como a função tem um parâmetro declarado como inteiro, o compilador C não permitiria passar como parâmetro um número não-inteiro, então, isso não pode criar problema para essa função recursiva. Entretanto, um erro que pode acontecer neste caso é se o usuário especificar um número negativo:

Cálculo do fatorial usando número negativo, usando o programa code/cap5/fatorial.c

```
Entre o numero para calcular o fatorial: -4  
Segmentation fault: 11
```

A falha de segmentação (*segmentation fault*) mostrada quando se passa um argumento negativo para a função fatorial é o resultado da recursão infinita que ocorre na função: como em cada etapa o número é diminuído de um, um número negativo nunca vai chegar a 1 (ou 0) e portanto nunca vai parar no caso-base. Mesmo a noção matemática do fatorial não está definida para números negativos. Uma forma de consertar o programa acima seria imprimir um erro quando fatorial fosse chamado com um argumento negativo, ou retornar um valor qualquer, sem chamar a função recursivamente. Uma forma de fazer isso seria mudando o teste do caso base:

Código fonte code/cap5/fatorial2.c

Cálculo do fatorial com alteração do teste do caso base

```
#include <stdio.h>

// prototipo
int fatorial(int);

int main() {
    int n;

    printf("Entre o numero para calcular o fatorial: ");
    scanf("%d", &n);

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

    return 0;
}

int fatorial(int n) {
    if (n <= 1) // ❶
        return 1;
    else
        return n * fatorial(n - 1);
}
```

- ❶ Caso-base da função `fatorial`. O teste agora é se o parâmetro é menor ou igual a 1, o que inclui os números negativos. Neste caso, a função retorna 1 sem fazer chamadas recursivas.

Resultado de duas execuções do programa `code/cap5/fatorial2.c`

```
Entre o numero para calcular o fatorial: 5
5! = 120
Sandman:funcoes andrei$ ./fat
Entre o numero para calcular o fatorial: -4
-4! = 1
```

Neste caso, a função sempre retorna 1 para números negativos, o que é um resultado estranho, mas como o fatorial não está definido para números negativos, isso não chega a ser um grande problema.

A recursividade é um conceito de programação considerado avançado, mas se torna bastante importante em programas mais complexos. Alguns paradigmas de programação como a *programação funcional* e a *programação lógica* são bastante baseadas no uso de funções recursivas, e muitos processos se tornam muito mais fáceis de implementar usando funções recursivas. Por isso, apesar de não entrarmos em mais detalhes sobre funções recursivas aqui, é importante ter em mente que esse é um dos conceitos que se tornam importantes para a evolução de um programador.

5.9 Recapitulando

Neste capítulo, vimos uma importante ferramenta para a criação de programas na linguagem C: funções. As funções (e os procedimentos, que são tratados na linguagem C como funções) possibilitam

a reutilização de trechos de código em várias partes de um programa, e permitem isolar determinadas componentes do programa em unidades mais autocontidas, melhorando a modularização do programa. É raro que um programa não-trivial na linguagem C não faça uso de funções.

Vimos exemplos do uso de procedimentos (funções que não retornam valores) com e sem parâmetros. Vimos também como usar parâmetros para funções e como retornar valores a partir de uma função.

As regras que regem a visibilidade de variáveis locais, globais e parâmetros de funções foram apresentadas, assim como os diferentes modos de passagem de parâmetros e como utilizá-los: passagem por valor e por referência.

Também vimos como declarar funções usando protótipos, para poder utilizar essas funções antes de sua definição (ou em arquivos diferentes de onde elas estão definidas). Um exemplo simples de função recursiva foi mostrado, como forma de introdução ao conceito.

Compreender o conteúdo deste capítulo é extremamente importante para aprender a programar bem na linguagem C.

5.10 Exercícios Propostos

1. Escreva uma função que calcula a média final de um aluno que fez prova final em uma disciplina. A função deve receber a média parcial do aluno (média das notas nas provas regulares da disciplina) e a nota obtida na prova final. O cálculo para a média final é

$MF = \frac{6 \times MP + 4 \times PF}{10}$ onde MF é a média final, MP é a média parcial e PF é a nota da prova final. Escreva um programa que utiliza esta função, pedindo os dados necessários ao usuário e imprimindo o valor da média final na tela.

2. Às vezes é útil limpar a "tela" antes de imprimir informações adicionais no console. Uma forma de limpar a tela é imprimir mais de 24 linhas em branco (dependendo do tamanho do console). Em geral, imprimir 30 linhas em branco deve ser suficiente. Escreva uma função (procedimento) que limpe a tela imprimindo 30 linhas em branco.
3. Escreva uma função que recebe dois parâmetros a e b e troca o valor de a com o valor de b se o valor de a for maior do que o de b ; o objetivo é ter, ao final, o menor dos dois valores em a e o maior em b . Por exemplo, se $a = 5$ e $b = 3$, então os valores das duas variáveis devem ser trocados, mas se $a = 2$ e $b = 7$, então a ordem já está correta e não é necessário trocar os valores. Utilize passagem de parâmetros por referência para poder afetar o valor das variáveis. Escreva um programa para testar a função.
4. O que é impresso pelo seguinte programa?

Código fonte code/cap5/exercicio4.c

```
#include <stdio.h>

int x = 5;
int y = 9;

int f(int x) {
    int z = x * x;
    return z * y;
}
```



```
int main() {  
    int y = 3;  
  
    printf("%d \n", x);  
    printf("%d \n", f(y));  
  
    return 0;  
}
```

Feedback sobre o capítulo

Você pode contribuir para melhoria dos nossos livros. Encontrou algum erro? Gostaria de submeter uma sugestão ou crítica?



Acesse <https://github.com/edusantana/introducao-a-programacao-livro/issues/new> para realizar seu feedback. Lembre-se de incluir na mensagem a seção, capítulo (**cap5**) e a versão do livro (**v1.0.2**) alvo de sua contribuição. Você receberá notificações sobre os encaminhamentos que serão dados a partir do seu feedback. Para compreender melhor como feedbacks funcionam consulte o guia do curso.

Capítulo 6

Índice Remissivo

A

argumento, 68
atribuição, 26

B

break, 48

C

Constantes, 22
Constantes simbólicas, 23
continue, 49

D

do-while, 44

E

E, 35
Enquanto, 42
Entrada padrão, 17
escopo, 77
ESCREVA, 33
Expressões lógicas, 34

F

for, 45
função recursiva, 87
Funções, 63
funções, 64

I

if, 34, 37
if-else, 38
incremento, 26
indentação, 24, 39

L

Laço infinito, 46
LEIA, 33

M

módulos, 64
matrizes, 59

N

Não, 35
NULO, 57

O

Ou, 35

P

parâmetro, 68
Passagem de Parâmetros, 80
Passagem por Referência, 82
Passagem por Valor, 81
procedimentos, 63

S

Saída padrão, 17
Se, 33
stdio, 27
strings, 57
switch, 40

V

variáveis globais, 83
vetor, 52

W

while, 42

