

Iguatemi Eduardo da Fonseca
Eduardo de Santana Medeiros Alexandre

Linguagem de Programação I: Programação Estruturada usando C

Editora da UFPB
João Pessoa
2014



UNIVERSIDADE FEDERAL DA PARAÍBA

Reitora	MARGARETH DE FÁTIMA FORMIGA MELO DINIZ
Vice-Reitor	EDUARDO RAMALHO RABENHORST
Pró-reitora de graduação	ARIANE NORMA DE MENESES SÁ
Diretor da UFPB Virtual	JAN EDSON RODRIGUES LEITE
Diretor do CI	GUIDO LEMOS DE SOUZA FILHO



EDITORA DA UFPB

Diretora	IZABEL FRANÇA DE LIMA
Supervisão de Editoração	ALMIR CORREIA DE VASCONCELLOS JÚNIOR
Supervisão de Produção	JOSÉ AUGUSTO DOS SANTOS FILHO

CURSO DE LICENCIATURA EM COMPUTAÇÃO A DISTÂNCIA

Coordenador	LUCIDIO DOS ANJOS FORMIGA CABRAL
Vice-coordenadora	DANIELLE ROUSY DIAS DA SILVA
Conselho Editorial	Prof Dr. Lucídio Cabral (UFPB) Prof Dr. Danielle Rousy (UFPB)

F676l Fonseca, Iguatemi Eduardo da.

Linguagem de Programação I: programação estruturada usando C /
Iguatemi Eduardo da Fonseca, Eduardo de Santana Medeiros Alexandre. - João
Pessoa: Editora da UFPB, 2014.

211. : il. –

ISBN: 978-85-237-0909-9

Curso de Licenciatura em Computação na Modalidade à Distância.
Universidade Federal da Paraíba.

1. Programação de computadores. 2. Estruturas da linguagem C. 2.
Registros. 3. Ponteiros. 4. Alocação Dinâmica. 5. Arquivos. I. Título.

CDU 004.42

Todos os direitos e responsabilidades dos autores.

EDITORA DA UFPB
Caixa Postal 5081 – Cidade Universitária
João Pessoa – Paraíba – Brasil
CEP: 58.051 – 970
<http://www.editora.ufpb.br>

Impresso no Brasil
Printed in Brazil

Linguagem de Programação I

Programação Estruturada usando C

Sumário

I	Conteúdos programáticos	1
1	Revisão	2
1.1	Algoritmos computacionais	2
1.2	A linguagem de programação C	3
1.3	Estruturas de dados e estruturas de controle	4
1.3.1	Declaração de variáveis em C	5
1.3.2	Estruturas de controle sequenciais e de decisão	6
1.3.3	Estruturas de repetição	9
1.3.3.1	Estrutura de repetição para um número definido de repetições (estrutura <code>for</code>)	9
1.3.3.2	Estrutura de repetição para um número indefinido de repetições (estrutura <code>while</code>)	10
1.3.3.3	Estrutura de repetição para um número indefinido de repetições e teste no final da estrutura (estrutura <code>do-while</code>)	10
1.3.4	Arrays em C	11
1.3.4.1	Arrays unidimensionais ou vetores	11
1.3.4.2	Arrays bidimensionais ou multidimensionais	12
1.4	Estrutura geral de um programa em C	14
1.5	Atividades	16
2	Registros	20
2.1	Definição de registro	20
2.2	Sintaxe para criação de registros	22
2.3	Identificadores de registros	22
2.4	Análise para criação de Registros	23
2.4.1	Situação do cálculo das notas de um aluno	23
2.4.2	Situação do cálculo e consulta do IMC de uma pessoa	24
2.4.3	Situação sobre manipulação de pontos no plano cartesiano	25

2.4.4	Situação sobre cadastro de produtos no supermercado	26
2.4.5	Situação sobre gerenciamento de contas bancárias	27
2.5	Exemplos de utilização dos Registros	29
2.5.1	Aluno	29
2.5.2	Produto	30
2.5.3	Pontos	30
2.6	Exercícios resolvidos	31
2.6.1	Programa do cálculo de médias de alunos	32
2.6.2	Problema do cálculo e consulta do IMC de uma pessoa	33
2.6.3	Problema de pontos no plano cartesiano	35
2.6.4	Problema sobre cadastro de produtos no supermercado	37
2.6.5	Problema sobre gerenciamento de contas bancárias	39
2.7	Inicializando registros	42
2.8	Composição de Registros	43
2.8.1	Triângulo	43
2.8.2	Informação Pessoal	44
2.9	Comparação entre Array e Registro	46
2.10	Recapitulando	46
2.11	Atividades	47
3	Ponteiros	50
3.1	Armazenamento de dados no computador	50
3.2	Definição de ponteiros	52
3.3	Ponteiros em funções ou sub-rotinas	56
3.3.1	Contextualização	56
3.3.2	Passagem de parâmetros em funções: passagem por valor vs passagem por referência	58
3.4	Atividades	62
4	Alocação Dinâmica de Memória	66
4.1	Alocação dinâmica de memória	68
4.2	Arrays dinâmicos	71
4.2.1	Vetores dinâmicos	71
4.2.2	Matrizes dinâmicas	73
4.2.3	Registros (estruturas) dinâmicas	74
4.3	Atividades	78

5	Arquivos	80
5.1	Os tipos de arquivos	81
5.2	Arquivos e fluxos	81
5.2.1	Fluxo de dados	81
5.2.2	Arquivos	82
5.2.3	Arquivos em C	82
5.3	Cabeçalho <code>stdio.h</code>	82
5.4	Abrindo e Fechando arquivos em C	82
5.4.1	Abrindo um arquivo com <code>fopen</code>	83
5.4.2	Entrada e saídas padrões	85
5.4.3	Fechando um arquivo	85
5.5	Indicadores de erro e final de arquivo	86
5.5.1	Erro na leitura/escrita	86
5.5.2	Atingiu o final do arquivo	87
5.6	Lendo e escrevendo um carácter por vez	87
5.6.1	Lendo um carácter com <code>fgetc</code> ou <code>getc</code>	87
5.6.2	Escrevendo um carácter por vez com <code>fputc</code> e <code>putchar</code>	87
5.6.3	Exemplo completo de leitura e escrita	88
5.7	Lendo e escrevendo uma linha por vez com <code>fgets</code> e <code>fputs</code>	89
5.7.1	Lendo uma linha com <code>fgets</code>	89
5.7.2	Escrevendo um string com <code>fputs</code>	89
5.7.3	Exemplo completo de leitura e escrita	90
5.8	Lendo e escrevendo dados binários	91
5.8.1	Lendo uma sequencia de elementos de um arquivo com <code>fread</code>	91
5.8.2	Escrevendo uma sequencia de elementos de um arquivo com <code>fwrite</code>	91
5.8.3	Exemplo completo de leitura e escrita binária	91
5.9	Descarregando o buffer com <code>fflush</code>	93
5.10	Escrevendo e lendo dados formatados	94
5.10.1	Ler dados formatados	94
5.10.2	Escrever dados formatados	94
5.11	Movendo o indicador de posição	94
5.11.1	Retrocede para o início do arquivo com <code>rewind</code>	94
5.11.2	Lendo a posição atual	94
5.11.3	Indo para o final do arquivo	95
5.12	Recapitulando	95
5.13	Atividades	95

II	Projeto	98
6	Lingua do i	99
6.1	Etapa 1: Estrutura inicial do projeto	99
6.1.1	Criando um main inocente	100
6.1.2	Iniciando um teste	101
6.1.3	Elaborando o Makefile	102
6.1.4	Criando um main e compilando o teste	103
6.1.5	Implementando a função verificaConteudosSaoIguais	104
6.1.6	Implementação inocente de lerConteudoDoArquivo	105
6.1.7	Implementando o cabeçalho de core	106
6.1.8	Arquivo texto para o teste	107
6.1.9	Verificando o teste falhar	107
6.1.10	Código da etapa	108
6.2	Etapa 2: Utilizando <code>assert</code> em vez de <code>exit</code>	108
6.2.1	Utilizando <code>assert</code> no teste	108
6.2.2	Compilação e execução do teste	109
6.2.3	Código da etapa	109
6.3	Etapa 3: Fazendo o teste passar	109
6.3.1	Implementação para fazer o teste passar	110
6.3.2	Código da etapa	110
6.4	Etapa 4: Lendo do arquivo	111
6.4.1	Abrindo e fechando arquivo para leitura	111
6.4.2	Verificando se o arquivo foi aberto com sucesso	112
6.4.3	Lendo conteúdo do arquivo aberto	112
6.4.4	Descobrimo o tamanho do arquivo	113
6.4.5	Executando o teste	113
6.4.6	Código da etapa	113
6.5	Etapa 5: Trocando as vogais do string por i	114
6.5.1	Inclusão de teste de tradução	114
6.5.2	Criando a função traduzParaLingaDoI	115
6.5.3	Código da etapa	115
6.6	Etapa 6: Fazendo o teste passar	115
6.6.1	Fazendo o teste passar com esforço mínimo	116
6.6.2	Código da etapa	116
6.7	Etapa 7: Implementando troca das vogais	116

6.7.1	Implementando visão geral da tradução	117
6.7.2	Construindo função para tradução de caracteres	117
6.7.3	Código da etapa	118
6.8	Etapa 8: Depurando a aplicação	119
6.8.1	Depuração do programa	119
6.8.2	Correção realizada	119
6.8.3	Código da etapa	119
6.9	Etapa 9: Inclusão de novos testes	120
6.9.1	Adição de testes	120
6.9.2	Atualizar a implementação para fazer o teste passar	121
6.9.3	Código da etapa	122
6.10	Etapa 10: Tratando texto com acentos	122
6.10.1	Desfazer teste com acentos	123
6.10.2	Desfazer implementação de core que tratava acentos	123
6.10.3	Mantendo registro de novas funcionalidades no TODO	124
6.10.4	Código da etapa	124
6.11	Etapa 11: Salvando conteúdo em arquivo	124
6.11.1	Adicionar teste de salvamento de conteúdo	125
6.11.2	Função para criar arquivo temporário	125
6.11.3	Verificando conteúdo salvo no arquivo	126
6.11.4	Atualização do arquivo de cabeçalho do core	126
6.11.5	Implementação vazia para possibilitar teste falhar	126
6.11.6	Verificando o teste falhando	127
6.11.7	Fazendo o teste passar	127
6.11.8	Código da etapa	128
6.12	Etapa 12: Determinando entrada do aplicativo	128
6.12.1	Testes para especificar a entrada	128
6.12.2	Fazendo os testes falharem	130
6.12.3	Certificando-se que todos os novos testes estão falhando	132
6.12.4	Código da etapa	134
6.13	Etapa 13: Fazendo os testes de entrada passarem	134
6.13.1	Fazendo os testes passarem	134
6.13.2	Código da etapa	135
6.14	Etapa 14: Implementando aplicação da lingua-do-i	136
6.14.1	Construindo o <code>main</code>	136
6.14.2	Executando lingua-do-i	137

6.14.3	Código da etapa	137
6.15	Etapa 15: Processando entrada por fluxo	138
6.15.1	Incluindo teste para tradução de fluxo	138
6.15.2	Salvando a letra da música e sua tradução	139
6.15.3	Fazendo o teste falhar	139
6.15.4	Fazendo o teste passar	140
6.15.5	Implementando tradução por fluxo	141
6.15.5.1	Lendo conteúdo da entrada com <code>fgets</code>	141
6.15.5.2	Sobre as funções <code>fgets</code> e <code>feof</code>	142
6.15.5.3	Condição de parada tentouLerAposFinalDoArquivo	142
6.15.5.4	Salvando o conteúdo com a função <code>fwrite</code>	143
6.15.6	Verificando que os testes continuam passando	143
6.15.7	Atualizando aplicação para processar fluxos	143
6.15.8	Código da etapa	144
6.16	Etapa 16: Tratando acentos	144
6.16.1	Fazendo o teste de tradução de vogais acentuadas falhar	144
6.16.2	Entendendo o que é necessário para fazer o teste passar	145
6.16.3	Fazendo o primeiro teste de tradução acentuada passar	146
6.16.4	Criando teste de tradução específico para vogais acentuadas	148
6.16.5	Fazendo o teste específico das vogais acentuadas passar	149
6.16.6	Fazendo o teste de tradução da música passar com os acentos	151
6.16.7	Código da etapa	153
A	Depuração	155
A.1	Requisitos da depuração	155
A.2	Depurando o programa lingua do i	155
A.2.1	Checando o pré-requisito para depuração	155
A.2.2	Última execução do programa	156
A.2.3	Escolha do ponto de parada	157
A.2.4	Realizando a depuração com o <code>gdb</code>	158
A.2.5	Resultado da depuração	160
B	Conteúdo extra: Arquivos especiais	161
B.1	Arquivos especiais que representam dispositivos	161
B.1.1	No Linux	161
B.1.2	No Windows	162
B.1.2.1	Testando arquivos de dispositivos no Windows	162
B.2	Arquivos especiais que criam um Pipe em memória	163
B.3	Arquivos especiais devido ao conteúdo	163

C	Experimento com mkfifo para demonstrar arquivos especiais	165
C.1	Imprimindo na saída padrão	165
C.2	Criando um arquivo especial com mkfifo	166
C.3	Utilizando o arquivo especial	166
D	Bibliotecas	169
D.1	Padrão C1X da Linguagem C	169
D.2	main()	169
D.3	assert.h	169
D.3.1	assert()	170
D.4	stdlib.h	170
D.4.1	exit()	170
D.4.2	EXIT_SUCCESS	170
D.4.3	EXIT_FAILURE	170
D.4.4	Gerência de memória	170
D.4.4.1	malloc()	170
D.4.4.2	calloc()	170
D.4.4.3	free()	170
D.4.4.4	realloc()	171
D.5	string.h	171
D.5.1	strlen()	171
D.5.2	strcmp()	171
D.5.3	strncmp()	171
D.5.4	strcpy()	171
D.5.5	strncpy()	171
D.6	stdbool.h	172
D.7	stdio.h	172
D.7.1	fopen()	172
D.7.2	fclose()	173
D.7.3	fgetc()	173
D.7.4	getchar()	173
D.7.5	fputc()	173
D.7.6	putchar()	173
D.7.7	fgets	174
D.7.8	fputs	174
D.7.9	fread()	174

D.7.10	fwrite()	174
D.7.11	fflush()	175
D.7.12	fseek()	175
D.7.13	ftell()	175
D.7.14	rewind()	175
D.7.15	fscanf()	175
D.7.16	scanf()	175
D.7.17	printf()	176
D.7.18	fprintf	176
D.7.18.1	Flags	176
D.7.18.2	Modificadores de tamanho	176
D.7.18.3	Especificadores de Conversão de tipos	176
D.7.18.4	Resumo e utilização do formato	177
D.7.19	feof	180
D.8	math.h	180
D.8.1	M_PI	180
D.8.2	fabs()	180
D.8.3	pow()	181
D.8.4	sqrt()	181
E	Respostas das atividades	182
E.1	Capítulo 1	182
E.2	Capítulo 2	183
7	Índice Remissivo	185

Prefácio

BAIXANDO A VERSÃO MAIS NOVA DESTE LIVRO

Acesse <https://github.com/edusantana/linguagem-de-programacao-i-livro/releases> para verificar se há uma versão mais o Histórico de revisões, na 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 continuar o estudo da Linguagem C, iniciado em disciplina anterior de Introdução a Programação.

A primeira parte deste livro contém:

Capítulo 1

Será realizada uma revisão dos conteúdos introdutórios da Linguagem C, que são pré-requisitos para os demais capítulos do livro.

Capítulo 2

Será apresentado o conceito de Registro e como utilizá-lo na Linguagem C.

Capítulo 3

Veremos a utilização de Ponteiros e os tipos de passagem de parâmetros.

Capítulo 4

Aprenderemos o que é Alocação Dinâmica de Memória e como realizá-la.

Capítulo 5

Conheceremos os diferentes tipos de Arquivos e aprenderemos como manipulá-los em C.

A segunda parte deste livro contém:

Capítulo 6

Apresentação de todas as etapas do desenvolvimento do projeto *lingua do i*, demonstrando a utilização de arquivos e testes para guiar o desenvolvimento.

Apêndice A

Demonstração da depuração de uma aplicação com o `gdb`.

Apêndice B

Experimento com `mkfifo` para demonstrar a existência de arquivos especiais.

Apêndice C

Documentação das bibliotecas em C, utilizadas no livro.

Apêndice D

Contém respostas de algumas atividades.

Público alvo

O público alvo desse livro são os alunos de Licenciatura em Computação, na modalidade à distância¹. Ele foi concebido para ser utilizado numa disciplina de *Linguagem de Programação I*, no segundo 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.



Importante

Esta caixa é utilizada para chamar atenção sobre algo importante.

¹Embora 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.



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 [x]

Referências a seções

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

Referências a imagens

Figura 2 [xiii]



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?

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>.



Figura 2: Exemplo de contribuição

Parte I

Conteúdos programáticos

Capítulo 1

Revisão

OBJETIVOS DO CAPÍTULO

Ao final deste capítulo você deverá ser capaz de:

- Entender e relembrar os principais conceitos vistos na disciplina Introdução à Programação;
- Ser capaz de utilizar em programas em linguagem C estruturas de controle (estruturas de seleção ou decisão, estruturas de repetição);
- Ser capaz de elaborar programas em linguagem C que utilizem arrays (vetores, matrizes, etc.) e funções ou sub-rotinas.

A Informática é uma área que se caracteriza por sofrer transformações em um curto espaço de tempo. Isso, principalmente, em decorrência do surgimento e aprimoramento de novas tecnologias de hardware, de novas técnicas e paradigmas de desenvolvimento de software e do contínuo desenvolvimento de ferramentas e aplicativos para a Internet. Entretanto, mesmo com todo o avanço e rápidas mudanças, ainda é preciso ter profissionais com conhecimento em programação de computadores, tanto nos paradigmas mais tradicionais e em programação estruturada e orientada a objeto, quanto nos paradigmas mais modernos como o de programação concorrente. A porta de entrada para o mundo da programação de computadores ainda continua sendo o aprendizado da construção de **algoritmos**, pois aprender programação de computadores não significa aprender apenas a sintaxe de uma linguagem de programação, mas, principalmente, saber modelar um problema a ser resolvido em uma linguagem computacional. O ciclo de aprendizado se completa quando se usa uma linguagem de programação para exercitar e representar o problema modelado previamente e executá-lo em um computador.

Neste capítulo serão revisados os principais conceitos vistos na disciplina *Introdução à Programação*, no entanto, não é objetivo rever a definição de algoritmo e aspectos básicos de programação, já que entende-se que estes tópicos já foram abordados na referida disciplina.

1.1 Algoritmos computacionais

Na resolução de problemas através de computadores, a tarefa desempenhada por eles é apenas parte do processo de solução. Há outras etapas que não são realizadas pelos computadores. As etapas na solução de problemas são:

- i. Entendimento do problema (realizada por pessoas);

- ii. Criação de uma sequência de operações (ou ações) que, quando executadas, produzem a solução para o problema e a tradução para uma linguagem de programação (realizada por pessoas);
- iii. Execução dessa sequência de operações (realizada pelo computador);
- iv. Verificação da adequação da solução (realizada por pessoas).

A etapa (i) é fundamental para que o programador saiba exatamente o que deve ser resolvido, quais são os requisitos do problema, necessidades do cliente interessado na solução, dentre outras coisas.

Na etapa (ii) é feita a modelagem do problema em uma linguagem computacional, ou seja, é desenvolvido o algoritmo, e, em seguida, é feita a tradução do algoritmo para uma linguagem de programação que seja mais apropriada ao problema. Neste ponto, pode-se perguntar qual a melhor linguagem de programação? A resposta para essa pergunta é direta. Não existe uma linguagem de programação que seja a melhor para todos os tipos de problemas. A escolha da linguagem depende da natureza do problema, logo, pode-se usar C ou C++ em uma situação e Java em outra, e assim por diante. Com o tempo, o aluno vai ganhando conhecimento e maturidade e será capaz de saber qual paradigma de programação (estruturado, orientado a objeto, recorrente, ...) é mais adequado e qual linguagem de programação pertencente ao paradigma escolhido é mais apropriada. Nas disciplinas *Introdução à Programação* e *Linguagem de Programação I* será estudado o paradigma de programação estruturada. Exemplos de linguagens de programação pertencentes a este paradigma são C, Fortran e Pascal.

Na etapa (iii) o algoritmo e/ou programa de computador desenvolvido na etapa anterior é executado em um computador gerando uma resposta ou solução para o problema.

A resolução do problema não acaba após a execução do programa pelo computador. É preciso avaliar e analisar com calma se a solução encontrada é viável ou correta. É muito comum a ocorrência de erros na modelagem/entendimento do problema na etapa (i) ou na construção do algoritmo ou programa na etapa (ii). Esses erros gerarão como consequência uma solução que não está correta. Caso isso aconteça, é preciso rever as etapas (i) e (ii) com cuidado para que os erros sejam encontrados até que a solução obtida na etapa (iii) passe a ser correta.

Finalmente, perceba que das quatro etapas, apenas um delas é executada por um computador. Note também que a resolução de problemas através de computadores não inicia com a construção direta do programa ou código em uma linguagem de programação. É preciso antes entender e modelar o problema.

1.2 A linguagem de programação C

O C nasceu na década de 1970 e teve como inventor Dennis Ritchie no AT&T Bell Labs. Ele é derivado de uma outra linguagem: o B, criado por Ken Thompson. O B, por sua vez, veio da linguagem BCPL, inventada por Martin Richards. C é uma linguagem de programação genérica que é utilizada para a criação de programas diversos como processadores de texto, planilhas eletrônicas, programas para a automação industrial, sistemas operacionais, implementação de protocolos de comunicação, gerenciadores de bancos de dados, programas de projeto assistido por computador, programas para a solução de problemas da Engenharia, Física, Química e outras Ciências.

C é uma linguagem *Case Sensitive*, isto é, *letras maiúsculas e minúsculas fazem diferença*. Se declararmos uma variável com o nome `soma` ela será diferente de `Soma`, `SOMA`, `SoMa` ou `sOmA`. Da mesma maneira, os comandos do C `if` e `for`, por exemplo, só podem ser escritos em minúsculas, caso contrário, o compilador não irá interpretá-los como sendo comandos, mas sim como variáveis ou outra estrutura do programa.

Existe um conjunto de palavras que são reservadas da linguagem e usadas para representar comandos de controle do programa, operadores e diretivas ou bibliotecas. A Tabela 1.1 [4] mostra palavras que são reservadas na linguagem C.

Tabela 1.1: Palavras reservadas em C

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Programas em C têm estrutura flexível. É possível escrever programas desorganizados que funcionam, porém, esses programas são ilegíveis e de difícil correção. Por esse motivo, é fundamental aprender e utilizar os conceitos e potencialidades da programação estruturada que são abordados nesta disciplina e na disciplina *Introdução à Programação*.

1.3 Estruturas de dados e estruturas de controle

Um programa de computador contém basicamente dois tipos de estruturas, a saber, as estruturas de dados e as estruturas de controle.

As estruturas de dados podem ser entendidas como estruturas que são utilizadas para armazenar e/ou representar as informações (dados) que são úteis para a resolução ou entendimento do problema e serão utilizados nos programas e algoritmos. Basicamente, há dois tipos de estruturas de dados:

Estrutura de dados estática

São estrutura que utilizam **alocação estática de memória**, o *espaço de memória* necessário para armazenar a estrutura do dado é sempre o mesmo, e é conhecido no momento da sua declaração. Exemplos de estruturas estáticas são: variáveis, *arrays* unidimensionais (também conhecidos como vetores), *arrays* multidimensionais (também conhecidos como matrizes de duas, três ou mais dimensões), registros (também conhecidos como estruturas ou tipos estruturados) e arquivos.

Estrutura de dados dinâmica

São estrutura que utilizam **alocação dinâmica de memória**, o *espaço de memória* necessário para armazenar a estrutura do dado não é constante, podendo aumentar ou diminuir sempre que um novo dado é adicionado ou removido da estrutura. Exemplos de estruturas dinâmicas são: filas, pilhas, listas encadeadas, árvores e tabelas de dispersão.



Nota

As estruturas de dados estáticas são abordadas neste livro, enquanto que as estruturas de dados dinâmicas serão abordadas na disciplina *Estrutura de Dados*.

As **estruturas de controle** são utilizadas para manipular e controlar as estruturas de dados, ou seja, controlar o fluxo de informação, e também para controlar as tomadas de decisão dentro de um programa ou algoritmo.

1.3.1 Declaração de variáveis em C

A linguagem C possui cinco tipos básicos que podem ser usados na declaração das variáveis, a saber, `int`, `float`, `double`, `void` e `char`. A partir desses tipos são utilizadas palavras-chaves para qualificar outras variáveis. São elas:

- i. `short` ou `long`, que se referem ao tamanho ou espaço de representação numérica da variável;
- ii. `signed` ou `unsigned`, que se referem ao tipo definido com ou sem sinal, respectivamente.

A Tabela 1.2 [5] mostra os **Tipos de dados** com suas variantes, a **Faixa de valores** numéricos que pode representar, o **Formato** para uso nas funções `printf` e `scanf` (utilizadas para impressão e leitura de dados respectivamente), e o **Tamanho em bits** necessário para armazenamento na memória. Os bytes são armazenados de forma contínua na memória. O código a seguir ajuda a calcular os s bits usados como tamanho das variáveis que cada computador utiliza para armazenar na memória. Para o inteiro, por exemplo, dependendo do computador pode-se usar 16 ou 32 bits (ou $8*s$ na Tabela 1.2 [5]).

Tabela 1.2: Tipos de dados em C

Tipo	Faixa de valores	Formato	Tamanho em bits (aproximado)
<code>char</code>	-128 a +127	<code>%c</code>	8
<code>unsigned char</code>	0 a +255	<code>%c</code>	8
<code>signed char</code>	-128 a + 127	<code>%c</code>	8
<code>int</code>	-2^{8s-1} a $+2^{8s-1}$	<code>%d</code>	$8*s$
<code>unsigned int</code>	0 a $(2^{8s} - 1)$	<code>%d</code>	$8*s$
<code>signed int</code>	-2^{8s-1} a $+2^{8s-1}$	<code>%d</code>	$8*s$
<code>short int</code>	-32.768 a +32.767	<code>%hd</code>	16
<code>unsigned short int</code>	0 a 65.535	<code>%hu</code>	16
<code>signed short int</code>	-32.768 a +32.767	<code>%hd</code>	16
<code>long int</code>	-2.147.483.648 a +2.147.483.647	<code>%ld</code>	32
<code>signed long int</code>	-2.147.483.648 a +2.147.483.647	<code>%lu</code>	32
<code>unsigned long int</code>	0 a + 4.294.967.295	<code>%lu</code>	32
<code>float</code>	3,4E-38 a + 3,4E+38	<code>%f</code>	32
<code>double</code>	1,7E-308 a 1,7E+308	<code>%lf</code>	64
<code>long double</code>	3,4E-4932 a 3,4E+4932	<code>%lf</code>	80

Exemplo 1.1 Programa para demonstrar os tamanhos dos tipos em bits

Código fonte /tamanho_dos_tipos.c[[code/cap1/tamanho_dos_tipos.c](#)]

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

int main(void) {
    printf("sizeof (unsigned int) = %d bits\n", sizeof (unsigned int)*8);
    printf("sizeof (signed int) = %d bits\n", sizeof (signed int)*8);
    printf("sizeof (short int) = %d bits\n", sizeof (short int)*8);
    printf("sizeof (short int) = %d bits\n", sizeof (long long)*8);
    //system ("pause");
    return EXIT_SUCCESS;
}
```

Saída da execução do programa

```
sizeof (unsigned int) = 32 bits
sizeof (signed int) = 32 bits
sizeof (short int) = 16 bits
sizeof (short int) = 64 bits
```

1.3.2 Estruturas de controle sequenciais e de decisão

As estruturas sequenciais são as mais simples e se caracterizam por serem sequências ordenadas de comandos que são executados da maneira como aparecem e sem desvios de execução, a não ser que exista algum comando específico que cause um desvio de execução. O Exemplo 1.2 [6] apresenta a sintaxe geral de estruturas sequenciais.

Exemplo 1.2 Sintaxe geral da estrutura de controle sequencial

Estrutura de controle sequencial:

```
// comando1;
// comando2;
// comando3;
```

Código fonte /estrutura_sequencial.c[[code/cap1/estrutura_sequencial.c](#)]

```
float nota1 = 6;
float nota2 = 8;
float media = (nota1 + nota2)/2;
printf("Media = %f\n", media);
```

De acordo com o problema em análise, talvez seja preciso seguir caminhos diferentes dependendo do teste de uma condição. As estruturas de seleção ou decisão permitem que um grupo de comandos seja executado de acordo com a aceitação ou não de certas condições. Uma condição lógica (booleana) é testada e, dependendo do resultado (**verdadeiro** ou **falso**), um determinado caminho é seguido dentro do programa. Pode-se ter três tipos de estrutura de decisão, a saber, **decisão simples**, **composta** e **múltipla**. O Exemplo 1.3 [7] apresenta a sintaxe geral da estrutura de controle de seleção simples. Caso condição seja **verdadeira**, o bloco de comandos 1 é executado. Caso seja **falsa**, serão executados os comandos após o fechamento da chave.

Exemplo 1.3 Estrutura de decisão simples

Sintaxe geral da estrutura de controle de decisão simples

```
if (condição) {  
    // bloco de comandos 1;  
} // fechamento da chave
```

Código fonte

```
printf("Digite duas notas não negativas de um(a) aluno(a): ");  
scanf("%f %f", &nota1, &nota2);  
  
if (nota1 < 0 || nota2 < 0) {  
    // Bloco de decisão simples  
    printf("Notas não podem ser negativas.\n");  
    exit(EXIT_FAILURE);  
}
```

**Importante**

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

Há problemas em que é preciso testar várias condições (como pode ser visto no Exemplo 1.4 [7]). O funcionamento da estrutura composta é simular ao da seleção simples. Caso condição1 seja verdadeira, executa-se o **bloco de comandos 1**; **senão** (else) a condição 2 é testada e, se for verdadeira, executa-se o **bloco de comandos 2**; e assim por diante, até ser testada a última condição.

Exemplo 1.4 Estrutura de decisão composta

Sintaxe geral da estrutura de decisão composta

```
// Seleção composta com apenas um else  
if (condição1) {  
    // bloco de comandos 1;  
} else {  
    // bloco de comandos 2;  
}  
  
// seleção composta com else if  
if (condição1) {  
    // bloco de comandos 1;  
} else if (condição2) {  
    // bloco de comandos 2;  
} else if (condição3) {  
    // bloco de comandos 3;  
} else if (condiçãoN) {  
    // bloco de comandos N;  
} else {  
    // bloco de comando default;  
}
```

Código fonte

```
// decisão composta
if (nota1 < 0 || nota1 >10 || nota2 < 0 || nota2 >10){//condição1
    // bloco de comandos 1
    printf("Alguma nota foi inválida: %f, %f\n", nota1, nota2);
    exit(EXIT_FAILURE);
} else if (media>=7.0 && media<=10){//condição2
    // bloco de comandos 2
    printf("Situação do aluno(a): APROVADO(A)\n");
} else if (media>=4 && media<7){//condição3
    // bloco de comandos 3
    printf("Situação do aluno(a): PRECISA FAZER PROVA FINAL\n");
} else {
    // bloco de comandos default
    printf("Situação do aluno(a): REPROVADO(A)\n");
}
```

A estrutura de decisão múltipla permite realizar múltiplas comparações de valores de uma variável, através do comando `switch`, como pode ser visto no Exemplo 1.4 [7]. Neste caso, variável é comparada com `VALOR1`, se forem iguais as instruções 1 e 2 são executadas. Em seguida variável é comparada com `VALOR2`, se forem iguais então as instruções 3 e 4 são executadas. Por fim, caso variável não foi igual a nenhum dos valores, então as instruções em default são executadas.

Exemplo 1.5 Estrutura de decisão múltipla

Sintaxe geral da estrutura de decisão múltipla

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

Código fonte

```
switch (caracter) {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
        printf("Você digitou uma vogal minúscula\n");
        break;
    case 'A':
    case 'E':
    case 'I':
```

```
case 'O':
case 'U':
    printf("Você digitou uma vogal MAIÚSCULA\n");
    break;
default:
    printf("Você não digitou uma vogal\n");
}
```

**Importante**

O bloco de comandos `default` é opcional, pode ser utilizado, por exemplo, para dar uma mensagem de alerta ao usuário dizendo que todas as condições foram testadas e nenhuma delas foi **verdadeira**.

1.3.3 Estruturas de repetição

Uma estrutura de repetição é utilizada quando se deseja repetir um trecho do programa ou até mesmo o programa inteiro. O número de repetições pode ser um número fixo ou estar relacionado a uma condição definida pelo programador. Dessa maneira, há três tipos de estruturas de repetição que podem ser utilizadas para cada situação.

1.3.3.1 Estrutura de repetição para um número definido de repetições (estrutura `for`)

Esta estrutura é utilizada quando se sabe o número de vezes que um trecho do programa deve ser repetido. Esta estrutura é chamada em linguagem C de `for`. A sintaxe geral do comando `for` pode ser vista a seguir:

Sintaxe geral da estrutura de repetição `for`

```
for(i=valor inicial; condição de parada; incremento/decremento de i){
    // bloco de comandos
}
```

A primeira parte atribui um valor inicial à variável `i`, que é chamada de contador e tem a função de controlar o número necessário de repetições. A *condição de parada* é utilizada como critério para se saber quando deve-se parar o comando de repetição. Quando essa condição assumir o valor **falso**, o comando `for` será encerrado. A terceira parte é o *incremento* ou *decremento* do contador `i` e tem a função de alterar o valor do contador `i` até que a condição de parada assuma valor **falso**.

Exemplo do comando `for`

```
for (j=0; j<10; j++){
    // bloco de comandos;
}
```

Nesse exemplo é inicialmente atribuído o valor 0 ao contador, variável `j`, e depois vai sendo **incrementado** em uma unidade. A cada valor de `j`, o *bloco de comandos* é executado. Este processo se repete até que o valor do contador `j` se torne maior ou igual a 10, fazendo com que a condição `j<10` assumo o valor **falso**. Perceba que nesse exemplo o *bloco de comandos* será executado 10 vezes.

1.3.3.2 Estrutura de repetição para um número indefinido de repetições (estrutura `while`)

A estrutura de repetição `while` é utilizada principalmente quando o número de repetições não é conhecido pelo programador. A sintaxe geral da estrutura `while` é apresentada a seguir:

Sintaxe geral da estrutura de repetição `while`

```
while (condição) {  
    // bloco de comandos;  
}
```

Nessa estrutura, o *bloco de comando* é repetido enquanto a *condição* for verdadeira e a *condição de parada* é testada logo no início do comando `while`, assim, caso ela seja falsa, o bloco de comandos não é executado. O `while` pode também ser utilizado em situações em que se conhece o número de repetições, como demonstrado no Exemplo 1.6 [10].

Exemplo 1.6 Uso do `while`

Código fonte

```
int z=0;  
while (z<=3) {  
    printf("Continuo repetindo o comando.\n");  
    z++; ❶  
}
```

- ❶ O incremento do contador (`z++`) é último comando a ser executado no `while`.
-

1.3.3.3 Estrutura de repetição para um número indefinido de repetições e teste no final da estrutura (estrutura `do-while`)

Assim como a estrutura `while`, a estrutura `do-while` é utilizada principalmente quando o número de repetições não é conhecido pelo programador, a diferença é que a *condição de parada* é testada no final da estrutura, portanto o *bloco de comandos* é executado pelo menos uma vez, confira a sintaxe geral a seguir:

Sintaxe geral da estrutura de repetição `do-while`

```
do {  
    // bloco de comandos  
} while (condição);
```

O Exemplo 1.7 [10] demonstra o uso da estrutura `do-while`. Nele, o menu é exibido na tela a primeira vez e, caso o usuário digite uma opção diferente de 1, 2 ou 3, o comando é repetido. Perceba que nesse caso o `do-while` terminará somente quando o usuário escolher uma das opções 1, 2 ou 3.

Exemplo 1.7 Uso do `do-while`

Código fonte

Exemplo do `do-while`

```
do{  
    printf ("\n Escolha a fruta pelo numero: \n");  
    printf ("\t(1)...Mamão\n");
```

```
printf ("\t(2)...Abacaxi\n");  
printf ("\t(3)...Laranja\n");  
scanf("%d", &i);  
} while ((i<1) || (i>3));
```

1.3.4 Arrays em C

Um array em C pode ser entendido como uma estrutura de dados composta homogênea, ou seja, é capaz de armazenar um conjunto de dados do mesmo tipo (inteiros ou reais), que possuem o mesmo identificador (nome) e são alocados de forma contínua na memória. Podem ser de dois tipos, unidimensionais (também chamados de vetores) ou multidimensionais (também chamados de matrizes de n-dimensões).

1.3.4.1 Arrays unidimensionais ou vetores

Para se declarar um vetor em C, usa-se colchetes logo após o nome dado ao vetor e, dentro dos colchetes, a quantidade de posições do vetor. A Figura 1.1 [11] mostra um exemplo de definição de vetor. Note que é preciso também definir o tipo de dado que o vetor irá armazenar (`float`, `double`, `int`, etc.). Neste exemplo foi definido um vetor de `float` com nome `vet` e com tamanho de oito posições, é dado também um exemplo de um vetor preenchido na Figura 1.2 [11]. Serão então alocados na memória RAM um espaço com $8 * \text{sizeof}(\text{float})$ bytes para armazenamento deste vetor.

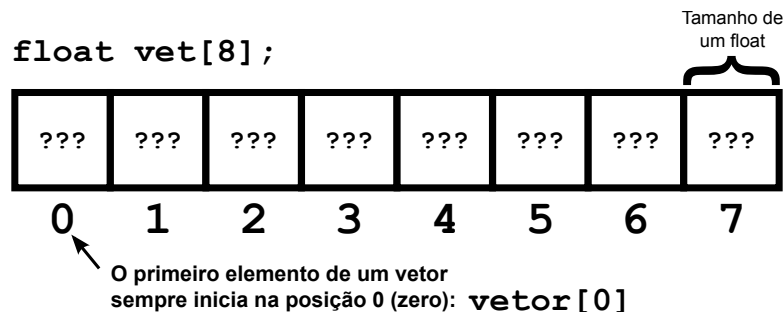


Figura 1.1: Vetor de float com tamanho 8, durante sua definição. Seu conteúdo é indeterminado e cada posição ocupa o tamanho de um float.

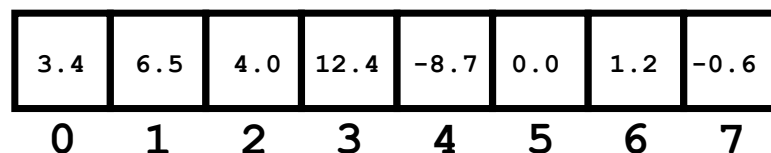


Figura 1.2: Exemplo do mesmo vetor preenchido com alguns valores.

As operações básicas que se pode fazer com vetores são: atribuir valores a posições dos vetores, preencher por completo o vetor e mostrar ou imprimir os valores armazenados no vetor.

Para atribuir um valor a uma posição do vetor basta fazer:

```
vet[0]= 3.4; // atribui o valor 3.4 a primeira posição do vetor vet
vet[6]= 0.0; // atribui o valor 0.0 a sétima posição do vetor vet
soma[4]= 28.9; // atribui o valor 28.9 a quinta posição do vetor soma
```

O preenchimento ou impressão completa de um vetor é sempre feito usando-se um comando de repetição para acessá-lo. Em geral, usa-se o comando `for` já que se conhece o tamanho do vetor e, portanto, quantas vezes os comandos serão repetidos. O preenchimento de um vetor como o da Figura 1.1 [11] pode ser feito com um código como:

```
for(i=0; i<7; i++){
    printf("Digite o valor a ser preenchido em vet[%d]. \n", i+1);
    scanf("%f", &vet[i]); // grava o valor digitado na i-ésima posição
}
```

A impressão dos valores armazenados pode ser feita com um código como:

```
printf("Os valores armazenados no vetor vet são: \n");
for(i=0; i<7; i++){
    printf("%0.2f ", vet[i]);
}
```

É possível também definir e preencher vetores diretamente da seguinte forma:

```
float notas[6] = {1.3, 4.5, 2.7, 4.1, 0.0, 100.1};
```

1.3.4.2 Arrays bidimensionais ou multidimensionais

Um array multidimensional pode ser definido de maneira similar a um vetor, a diferença é que deve-se utilizar um índice para cada dimensão. No caso de vetores foi utilizado apenas um índice por que o vetor possui apenas uma dimensão. A sintaxe geral para um array multidimensional é:

```
tipo-de-dado  nome_do_array [dimensão1][dimensão2]...[dimensãoD];
```

Em que:

tipo-de-dado

corresponde ao tipo de dado (`int`, `float`, `double`, ...) a ser armazenado na matriz;

nome_do_array

é o identificador ou nome dado à matriz;

[dimensão1]

é o tamanho da primeira dimensão da matriz;

[dimensão2]

é o tamanho da segunda dimensão da matriz;

[dimensãoD]

é o tamanho da D-ésima dimensão da matriz;

Em geral, usam-se matrizes com duas dimensões, as quais recebem os nomes de linhas e colunas da matriz. A Figura 1.3 [13] mostra um exemplo de como definir uma matriz com 3 linhas e 4 colunas para armazenar números do tipo `double`.

`double matriz[3][4];`

	0	1	2	3
0	0.5	-9.6	3.4	4.5
1	-0.9	-1.5	9.6	-7.4
2	3.3	23.9	30	104

Diagram illustrating a 3x4 matrix. The rows are indexed 0, 1, and 2, and the columns are indexed 0, 1, 2, and 3. A bracket on the right indicates 3 rows, and a bracket at the bottom indicates 4 columns.

Figura 1.3: Exemplo de definição de um array bidimensional ou matriz

As operações básicas que se pode fazer com matrizes são: atribuir valores a posições das matrizes, preencher por completo a matriz e mostrar ou imprimir os valores armazenados na matriz.

Para atribuir um valor a uma posição da matriz basta fazer:

```
// atribui o valor 0.5 a primeira posição de matriz
matriz[0][0]= 0.5;
// atribui o valor 104 a posição linha 2, coluna 3 de matriz
matriz[2][3]= 104;
```

O preenchimento ou impressão completa de um array é sempre feito usando-se comandos de repetição para acessar as dimensões do array. Para cada dimensão usa-se um comando de repetição que, em geral, é o comando `for`. O preenchimento de uma matriz de duas dimensões, como a da Figura 1.3 [13], pode ser feito como no Exemplo 1.8 [13].

Exemplo 1.8 Preenchimento de uma matriz de duas dimensões

Código fonte /matriz.c[code/cap1/matriz.c]

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

int main(void) {

    double matriz[3][4];

    // Preenchimento de matriz
    for(int i=0; i<3; i++){
        for(int j=0; j<4; j++){
            printf("Digite valor a ser preenchido em matriz[%d,%d]: ", i, j);
            scanf("%lf", &matriz[i][j]); // grava na linha i coluna j
        }
    }
}
```

```
// Impressão de matriz
printf("Os valores armazenados em matriz são: \n");
for (int i=0; i<3; i++){
    for (int j=0; j<4; j++){
        printf ("%lf ", matriz[i][j]);
    }
    printf ("\n");
}

return EXIT_SUCCESS;
}
```

Saída da execução do programa

```
Digite valor a ser preenchido em matriz[0,0]: 0
Digite valor a ser preenchido em matriz[0,1]: 1
Digite valor a ser preenchido em matriz[0,2]: 2
Digite valor a ser preenchido em matriz[0,3]: 3
Digite valor a ser preenchido em matriz[1,0]: 10
Digite valor a ser preenchido em matriz[1,1]: 11
Digite valor a ser preenchido em matriz[1,2]: 12
Digite valor a ser preenchido em matriz[1,3]: 13
Digite valor a ser preenchido em matriz[2,0]: 20
Digite valor a ser preenchido em matriz[2,1]: 21
Digite valor a ser preenchido em matriz[2,2]: 22
Digite valor a ser preenchido em matriz[2,3]: 23
Os valores armazenados em matriz são:
0.000000  1.000000  2.000000  3.000000
10.000000 11.000000 12.000000 13.000000
20.000000 21.000000 22.000000 23.000000
```

Observe que nos códigos do preenchimento e impressão da matriz `matriz` foram utilizados dois comandos de repetição, já que ela é bidimensional.

1.4 Estrutura geral de um programa em C

Um programa em C pode ter uma estrutura geral como aparece a seguir:

```
1  /* Comentários gerais explicando o que o programa faz . */
2  Diretivas de pré-compilação (include, define, uso de < > e " ")
3  Declarações globais
4  Definição de protótipo de funções ou sub-rotinas usadas no programa
5
6  int main() {
7      comandos contidos na função main; // comentários
8      return 0;
9  } // fim da função main
10
11 Implementação das funções ou sub-rotinas usadas no programa
```

A linha 2 representa a inclusão das bibliotecas da linguagem, como `stdio.h`, `stdlib.h`, `math.h`, dentre outras, ou bibliotecas criadas pelo próprio programador. A linha 3 representa a definição de declarações globais, como por exemplo, variáveis globais do programa. Em seguida, podem ser incluídos os protótipos das funções ou sub-rotinas criadas pelo programador e que serão utilizadas no programa. A linha 6 representa o início da função `main` do programa C com os seus comandos delimitados pelas chaves `{ ... }`. Finalmente, após a função `main`, são incluídos os códigos das funções ou sub-rotinas definidas na linha 4 e utilizadas no programa.

Embora seja simples, essa estrutura geral de um programa em C pode ser utilizada pelo estudante ou programador iniciante para não esquecer partes fundamentais que um programa em C deve conter. Pode-se acrescentar ainda no início do programa (linha 1) um comentário geral explicando o que ele faz, qual seu objetivo, que tipo de entrada precisa, que tipo de saída irá gerar, como pode ser visto no Exemplo 1.9 [15].

Exemplo 1.9 Programa com estrutura completa em C

Código fonte /programa_completo.c[[code/cap1/programa_completo.c](#)]

```
/* O propósito deste programa é demonstrar a estrutura geral
 * de um programa na linguagem C.
 * Ele possui uma variável global 'NUMERO_DE_ITERACOES', que determina
 * quantas vezes serão invocadas a função 'imprime_mensagem'.
 */

// Diretivas de pré-compilação
#include <stdio.h>
#include <stdlib.h>

// Declarações globais
int NUMERO_DE_ITERACOES = 3;

// Definição de protótipo de funções ou sub-rotinas usadas no programa
void imprime_mensagem(); // protótipo

int main() { // início do main
    for (int i=0; i < NUMERO_DE_ITERACOES; i++) {
        imprime_mensagem();
    }
    return EXIT_SUCCESS;
} // final do main

void imprime_mensagem(){ // implementação da função
    printf("Não sei, só sei que foi assim.\n");
}
```

Saída da execução do programa

```
Não sei, só sei que foi assim.
Não sei, só sei que foi assim.
Não sei, só sei que foi assim.
```

1.5 Atividades


1. Faça um algoritmo que receba um número positivo e maior que zero, calcule e mostre: o número digitado ao quadrado, o número digitado ao cubo e a raiz quadrada do número digitado.
2. Analise o algoritmo abaixo e explique o que ele faz. Quais são os valores de LUCROMAX, PREÇOMAX e INGRESSOSMAX que serão impressos no final da execução do algoritmo?

```
1 Algoritmo
2   declare PREÇO, //preço do ingresso em Reais
3   INGRESSOS, //número provável de ingressos vendidos
4   DESPESAS, //despesas com o espetáculo
5   LUCRO, //lucro provável
6   LUCROMAX, //lucro máximo provável
7   PREÇOMAX, //preço máximo
8   INGRESSOSMAX: numérico; //número máximo de ingressos vendidos
9   LUCROMAX ← 0;
10  PREÇO ← 5;
11  INGRESSOS ← 120;
12  DESPESAS ← 200;
13  Enquanto (PREÇO > 1) {
14    LUCRO ← INGRESSOS*PREÇO - DESPESAS;
15    imprima LUCRO, PREÇO;
16    se (LUCRO > LUCROMAX) {
17      LUCROMAX ← LUCRO;
18      PREÇOMAX ← PREÇO;
19      INGRESSOSMAX ← INGRESSOS;
20    }
21    INGRESSOS ← INGRESSOS + 26;
22    PREÇO ← PREÇO - 0,5;
23  }
24  imprima LUCROMAX, PREÇOMAX, INGRESSOSMAX;
25 fim algoritmo
```

3. Analise o código abaixo e diga qual é o valor que a variável S terá ao final da execução do algoritmo.

```
Algoritmo
declare fim, i, j, x, exp, num_ter, den, D, fat, S: numérico;
num_ter = 3;
x = 3;
S = 0;
D = 1;
para (i=1; i<= num_ter; i=i+1) {
  fim = D;
  fat = 1;
  para (j=1; j<= fim; j=j+1) {
    fat = fat*j;
  }
  exp = i+1;
  se (RESTO(exp/2) == 0) {
    S = S - (x^(exp))/fat;
  }
}
```

```
senão{
    S = S + (x^(exp))/fat;
}
se (D == 4) {
    den = -1;
}
se (D==1) {
    den = 1;
}
se (den == 1) {
    D = D +1;
}
senão{
    D = D - 1;
}
}
imprima S;
fim algoritmo
```

4. Faça um programa que leia um CPF, salve-o numa variável do tipo `long long` e em seguida imprima o CPF lido. 
5. Faça um programa em C que receba o ano de nascimento de uma pessoa e o ano atual, calcule e mostre: (a) a idade da pessoa; e (b) quantos anos ela terá em 2045. O programa deve ser capaz de receber somente valores positivos e maiores do que zero para a idade da pessoa.
6. Um supermercado deseja reajustar os preços de seus produtos usando o seguinte critério: o produto poderá ter seu preço aumentado ou diminuído. Para o preço ser alterado (ou seja, ser aumentado ou diminuído), o produto deve atender **PELO MENOS** uma das linhas da tabela a seguir:

Venda média mensal (unidades vendidas)	Preço atual (R\$)	% de aumento	% de diminuição
Menor que 500	< R\$ 30,00	10 %	-
Entre 500 (inclusive) e 1200	> = R\$ 30,00 e < R\$ 80,00	15 %	-
Igual ou maior que 1200	> = R\$ 80,00	-	20 %

Faça um programa em C que receba o preço atual e a venda média mensal do produto, calcule e mostre o novo preço.

7. João Papo-de-Pescador está organizando uma corrida de automóveis e precisa de um método para computar os tempos das voltas no circuito. Faça um programa em C que tenha como entrada: i) o número de pilotos que disputarão a corrida; ii) os seus respectivos nomes; iii) o número de voltas da corrida; iv) e o tempo de todas as voltas para cada piloto. O programa deve imprimir no final o tempo médio da volta para cada piloto e o tempo da volta mais rápida da corrida (ou seja, a volta que foi feita no menor tempo) e qual foi o piloto que a fez.
8. Em uma competição de ginástica olímpica, a nota é determinada por um painel de seis juízes. Cada um dos juízes atribui uma nota entre zero e dez para o desempenho do atleta. Para calcular a nota final, a nota mais alta e a nota mais baixa são descartadas e é calculada a média das quatro

notas restantes. Escreva um programa que leia seis notas que devem estar entre zero e dez e, em seguida, calcule a média após o descarte da maior e da menor nota. Faça duas versões desse programa em C: i) usando somente estruturas de seleção e de repetição e sem o uso de vetores; ii) com o uso de vetores e estruturas de seleção e repetição.

9. Uma empresa decidiu dar uma gratificação de Natal a seus funcionários baseada no número de horas extras e no número de horas que o funcionário faltou ao trabalho. Faça um programa em C que calcule o valor da gratificação de Natal. O valor do prêmio é obtido pela consulta à tabela que se segue, na qual a variável SH é calculada da seguinte forma: $SH = \text{número de horas extras} - 0,5 * (\text{número de horas faltadas})$.

SH (em minutos)	Gratificação (em R\$)
Maior ou igual a 2200	600,00
Entre 1700 (inclusive) e 2200	500,00
Entre 1000 (inclusive) e 1700	400,00
Entre 500 (inclusive) e 1000	300,00
Menor que 500	200,00

10. Analise o código abaixo e diga qual é o valor que a variável E terá ao final da execução do programa.

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

int main(void) {
    double fat, E;
    int i, j, n;
    n = 4;
    for(i=1; i<=n; i++){
        fat=1;
        for(j=1; j<=n; j++){
            fat=fat*j;
        }
        E=E+(1/fat);
    }
    printf ("O valor da variável E é %lf ", E);
    return 0;
}
```

11. Faça um programa em C que receba o salário de um funcionário e, usando a tabela a seguir, calcule e mostre o seu novo salário. O programa deve aceitar somente valores positivos e maiores do que zero para o salário do funcionário.

Faixa Salarial	% de aumento
Até R\$ 500,00	55 %
Entre R\$ 500,00 e 700,00 (inclusive)	45 %
Entre R\$ 700,00 e 900,00 (inclusive)	35 %
Entre R\$ 900,00 e 1100,00 (inclusive)	25 %
Entre R\$ 1100,00 e 1300,00 (inclusive)	15%
Acima de R\$ 1300,00	5%

12. Uma agência bancária possui vários clientes que podem fazer investimentos com rendimentos mensais conforme a tabela a seguir:

Código do tipo de investimento	Nome	Rendimento mensal %
1	Poupança	0,6 %
2	Poupança plus	1,1 %
3	Fundos de renda fixa	1,8 %
4	Fundos de renda variável	2,5 %

Faça um programa em C que tenha como dados de entrada o código do cliente, o código do tipo de investimento e o valor investido. O programa deve calcular e imprimir o rendimento mensal de acordo com o tipo de investimento do cliente. No final, deverá imprimir o total investido por todos os clientes consultados e o somatório do rendimento mensal pago a todos os clientes consultados. A leitura de clientes pelo programa terminará quando o código do cliente digitado for menor ou igual a 0 (zero).

**Feedback sobre o capítulo**

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

Para compreender melhor como feedbacks funcionam consulte o guia do curso.

Capítulo 2

Registros

OBJETIVOS DO CAPÍTULO

Ao final deste capítulo você deverá ser capaz de:

- Criar registros em C;
- Analisar problemas e reconhecer os campos necessários para utilizar nos registros;
- Reconhecer campos identificadores de registros;
- Criar e manipular listas de registros;

Neste capítulo nós iremos estudar sobre Registros. Vamos conhecer a sua utilidade e como declará-los em C. Depois vamos analisar diversas situações para aprender como é o processo de criação de um registro.

Serão apresentados alguns programas demonstrando a utilização dos registros e por fim, vamos aprender como compor registros a partir de outros registros.

Mas, o que são Registros?

2.1 Definição de registro

Definição de Registro

Um Registro é um **tipo de dado** criado pelo usuário, através da **composição** de outros tipos de dados.

Nós utilizamos registros quando desejamos criar um tipo de dado para reunir informações sobre o que desejamos representar. No registro, as informações são organizadas em **campos**.

Uma analogia de registro pode ser vista quando preenchemos um formulário. Na Tabela 2.1 [21] nós temos um exemplo de formulário para cadastrar **Clientes**. Os **campos** do formulário são preenchidos com os dados do cliente que nos interessa registrar.

Sabendo as informações que desejamos registrar sobre um Cliente, nós podemos esquematizar um registro, informando os tipos de dado de cada campo, conforme descrito na Tabela 2.2 [21].

Tabela 2.1: Formulário para cadastro de Cliente

Nome:	
Data de Nascimento:	Telefone para contato:
CPF:	RG:

Tabela 2.2: Representação de um registro Cliente.

Novo tipo	Campo	Tipo do campo
Cliente	Nome	Textual
	Data de Nascimento	Numérico
	Telefone para contato	Textual
	CPF ★	Numérico
	RG	Numérico

Relembrando

Em nossos programas nós utilizamos **variáveis** para manter as informações que desejamos manipular.

No momento da criação de uma variável precisamos especificar o **tipo de dado** que desejamos que ela mantenha, através da **declaração da variável**. Vamos relembrar como declaramos variáveis:



Em pseudocódigo

```
DECLARE nome_da_variavel: TEXTUAL  
DECLARE var1,var2,var3: NUMÉRICO
```

Em C

```
char nome_da_variavel[];  
double var1,var2,var3;
```

Quando especificamos mais de uma variável separadas por vírgula, assumimos que todas elas possuem o mesmo tipo.

Na próxima seção, veremos como é a sintaxe para criação de registros, em pseudocódigo e em C.

Importante



Embora, na prática, o uso de registro geralmente esteja associado a persistência de dados, sempre que mencionarmos **cadastrar** neste capítulo, estamos nos referindo a manter os dados **em memória** para consulta posterior.

Em um sistema real, geralmente existe alguma forma de persistência dos dados através de arquivos ou banco de dados — caso contrário, os dados seriam perdidos.

2.2 Sintaxe para criação de registros

Agora que temos o entendimento que um registro é um **tipo de dado**, vamos conhecer a sintaxe para especificá-lo:

Sintaxe em pseudocódigo para criar registro

```
REGISTRO nome_do_registro
    // Declarações dos campos
REGISTRO_FIM
```

Sintaxe em C para criar registro

```
typedef struct {
    // Declarações dos campos
} nome_do_registro;
```

Quando criamos um novo tipo de dado precisamos nomeá-lo, para podermos referenciá-lo mais tarde. Nestas notações, `nome_do_registro` é o nome do tipo de dado registro que será criado.

As Declarações dos campos definem os campos que **compõem** o registro. Esta **composição** ficará evidente nas próximas seções, onde iremos criar e manipular vários registros.

Após a definição do novo tipo de dado registro, uma declaração de variável com este tipo é realizada da forma usual:

Declaração de variável do tipo criado em pseudocódigo

```
DECLARE variavel_nome: nome_do_registro
```

Declaração de variável do tipo criado em C

```
nome_do_registro variavel_nome;
```

2.3 Identificadores de registros

Antes de começarmos a especificar os registros, vamos primeiro entender a necessidade de identificar unicamente um *registro*.

Importante

A palavra **registro** pode ser empregada em dois contextos diferentes.

Tipo de dado

É o tipo de dado, conforme apresentado na definição de Registro.



Instância do registro

Utilizando a analogia do formulário, equivaleria às fichas dos clientes. Cada ficha preenchida equivale a uma instância ou um *registro* daquele tipo.

Por conveniência, sempre que utilizarmos a palavra *registro* para indicar instância do tipo, ela será grafada em *itálico*.

Identificadores de *registros* são campos nos registros que os **identificam e diferenciam** um *registro* de qualquer outro.

No registro indicado na tabela Tabela 2.2 [21], como podemos **diferenciar** um cliente cadastrado de outro? Qual campo **identifica** um cliente? Seria o nome? Data de Nascimento? Telefone? CPF ou RG? Neste caso, preferimos utilizar o CPF, pois sabemos que duas pessoas diferentes não podem possuir o mesmo número de CPF.

Em nosso livro, os campos **identificadores** estão marcados com ★.

**Nota**

Os identificadores costumam ser do tipo **inteiro**, pois a comparação de inteiros é mais rápida do que a comparação textual.

Na próxima seção, faremos análises em algumas situações, caso você não tenha compreendido o que são **campos identificadores** terá outra oportunidade.

2.4 Análise para criação de Registros

Nesta seção mostramos o processo de criação de um Registro em diversas situações diferentes.

Em cada situação apresentada faremos a seguinte **análise**:

- Determinar o **tipo de registro** que vamos criar;
- Especificar quais serão os **campos** do registro, com os seus respectivos **tipos**;
- Indicar qual o **campo identificador** (★), caso exista;
- Apresentar o **código de criação** do Registro em Pseudocódigo e em C.

2.4.1 Situação do cálculo das notas de um aluno

Em uma disciplina na qual os alunos possuem **duas notas**, e precisamos registrar e **calcular as médias** de todos eles, como seria um registro para representar esta situação?

Nome do Registro

Aluno

Campos

Obviamente vamos precisar guardar **duas notas** para cada aluno. *Vamos precisar guardar a média também?* Não, uma vez que temos as duas notas registradas, sempre que desejarmos consultar a média poderemos calculá-las. *O nome do aluno seria uma informação útil?* Sem dúvidas! Será importante registrar o **nome** do aluno pois poderíamos imprimir uma lista com os nomes, notas e médias de cada aluno. *A matrícula do aluno é importante também?* Nós poderíamos suprimir a matrícula do aluno, mas qual seria a consequência disso? Por exemplo, na lista de notas poderia conter apenas os nomes, notas e médias. Mas o que aconteceria se tivéssemos dois alunos com o mesmo nome? Nós precisamos de uma informação extra para **identificar** e diferenciar um aluno do outro. Com este intuito, vamos optar por registrar a **matrícula** também. *O nome da disciplina é importante?* Neste caso não, pois estamos nos limitando aos alunos e suas notas.

Novo tipo	Campo	Tipo do campo
Aluno	matricula ★	Numérico
	nome	Textual
	nota1	Numérico
	nota2	Numérico

Registro em Pseudocódigo

```
REGISTRO Aluno
  matricula: NUMÉRICO
  nome: TEXTO
  nota1, nota2: NUMÉRICO
FIM_REGISTRO
```

Código fonte /reg_aluno.c[code/cap2/reg_aluno.c]

Registro de Aluno em C

```
typedef struct {
    int matricula;
    char nome[100];
    float nota1;
    float nota2;
} Aluno;
```



Nota

Até agora você teve dificuldade para entender esta análise? Você compreendeu a necessidade da utilização de *matricula* como campo identificador? Concordou com os tipos de dados utilizados para cada variável?

2.4.2 Situação do cálculo e consulta do IMC de uma pessoa

Nesta situação desejamos criar um sistema para **cadastrar** pessoas e em seguida **consultar** o IMC delas.

Nome do Registro

Pessoa

Campos

Para o cálculo do IMC são necessárias duas informações: a **altura** e o **peso**. Novamente, o **nome** da pessoa é uma informação relevante, pois vamos imprimir o IMC calculado junto com o nome. *Mas como realizar a consulta? Após o cadastro realizado de algumas pessoas, qual o parâmetro de busca que iremos utilizar para encontrar a pessoa certa?* Poderíamos utilizar o nome completo da pessoa para encontrá-la. Mas digitar o nome todo é enfadonho. Poderíamos utilizar apenas o primeiro nome para busca, mas então teríamos que apresentar um lista com todas as pessoas com aquele primeiro nome e selecionar a pessoa correta entre elas.¹ Se cadastrarmos o **CPF** da pessoa poderíamos consultá-la mais tarde informando apenas ele, simplificando a busca. Por último, como algumas tabelas do IMC apresentam os dados categorizados por **sexo**, vamos registrá-lo também.

¹A opção de utilizar o primeiro nome iria complicar o algoritmo da busca.

Novo tipo	Campo	Tipo do campo
Pessoa	nome	Textual
	peso	Numérico
	altura	Numérico
	cpf ★	Numérico
	sexo	Textual

Registro em Pseudocódigo

```
REGISTRO Pessoa
    nome, sexo: TEXTO
    peso, altura, cpf: NUMÉRICO
FIM_REGISTRO
```

Código fonte /reg_pessoa.c[[code/cap2/reg_pessoa.c](#)]

Registro de Pessoa em C

```
typedef struct{
    char nome[100];
    char sexo; // 'm': masculino, 'f': feminino
    float peso;
    float altura;
    long long cpf;
} Pessoa;
```

Nota

Mais uma vez, embora nosso problema não tenha indicado os campos que necessita, fomos capazes de deduzir alguns. Aqui não há certo ou errado, cada um pode realizar sua análise e chegar a resultados diferentes.

Você concorda com os tipos de dados apresentados aqui? Não achou estranho `cpf` ser do tipo `long long`? Você declararia `sexo` com outro tipo, diferente de **char**?



cpf

Declaramos ele como `long long` pois os tipos **long** e **int** não armazenam números na ordem de **11** dígitos.

sexo

Optamos por utilizar o tipo **char** para simplificar comparações, caso sejam necessárias. Poderíamos declará-lo do tipo **int**, fazendo uma correspondência de valores: 1=Feminino e 2=Masculino. Ou ainda poderíamos utilizar **char[]** e registrar o texto completo: Feminino ou Masculino.

2.4.3 Situação sobre manipulação de pontos no plano cartesiano

Nesta situação desejamos criar um sistema matemático para manipular pontos no plano cartesiano.

Nome do Registro

Ponto

Campos

Quais as informações que precisamos para registrar um ponto no plano cartesiano? Apenas suas coordenadas (x,y) . Existe mais alguma informação que desejamos registrar? O ponto possui um lugar associado a ele, como um região geográfica? **Não!** Os pontos serão plotados com colorações específicas? **Não!** Para identificar os campos identificadores, nos perguntamos: se dois pontos possuem as mesmas coordenadas, eles são os mesmos? **Sim!** Concluimos que, neste caso, as duas coordenadas juntas identificam o ponto.

Novo tipo	Campo	Tipo do campo
Ponto	x ★	Numérico
	y ★	Numérico

Registro em Pseudocódigo

```
REGISTRO Ponto
  x, y: NUMÉRICO
FIM_REGISTRO
```

Código fonte /reg_ponto.c[[code/cap2/reg_ponto.c](#)]

Registro de Ponto em C

```
typedef struct {
    int x;
    int y;
} Ponto;
```

Nota

Neste registro nós temos uma novidade: estamos utilizando dois campos como **identificadores** simultaneamente. As vezes um único campo não é suficiente para identificar um *registro*.



Neste caso, fica evidente que dois pontos são iguais se, e somente se, eles possuírem os mesmo valores para o par (x,y) .

E em relação aos tipos de dados das coordenadas? Você teria utilizado outro tipo, diferente de `int`, como `float` ou `double`? Mais uma vez, aqui não há certo ou errado, nós optamos por `int` apenas por ser mais simples fornecer coordenadas em inteiro.

2.4.4 Situação sobre cadastro de produtos no supermercado

Nesta situação desejamos criar um sistema, para um supermercado, que cadastre produtos e seus preços.

Nome do Registro

Produto

Campos

Para registrar um produto vamos precisar do seu **nome** e o seu **preço**. *Mas como identificar um produto cadastrado?* Quando vamos no supermercado e compramos alguma mercadoria

no peso, o caixa do supermercado precisa fornecer um código para cadastrar o produto pesado. Geralmente ele utiliza uma tabela, na qual há o nome do produto e o seu código. Para a nossa aplicação vamos utilizar este mesmo **código** para identificar unicamente cada produto.

Novo tipo	Campo	Tipo do campo
Produto	nome	Textual
	preco	Numérico
	codigo ★	Numérico

Registro em Pseudocódigo

```
REGISTRO Produto
  codigo: NUMÉRICO
  nome: TEXTUAL
  preco: NUMÉRICO
FIM_REGISTRO
```

Código fonte /reg_produto.c[code/cap2/reg_produto.c]

Registro de Produto em C

```
typedef struct {
    long    codigo;
    char    nome[100];
    float   preco;
} Produto;
```

Nota



Neste registro tivemos contato com um provável campo **identificador** universal, o `codigo`. Geralmente, quando nos deparamos com um campo código, ele será utilizado como o identificador.^a

^aExceto quando o código for utilizado para designar uma senha.

2.4.5 Situação sobre gerenciamento de contas bancárias

Nesta situação desejamos criar um sistema bancário para gerenciar clientes e suas contas bancárias.

Nomes dos Registros

Cliente e Conta.

Campos

O **nome** do cliente é uma informação relevante. O **CPF** poderá ser utilizado para **diferenciar** clientes com o mesmo nome. *Como identificar a conta do cliente?* Cada conta poderia ter um **número de conta único**, que serviria para identificar a conta do cliente. Cada conta terá um **saldo**, que será gerenciada pelo sistema. Como cada cliente pode possuir mais de uma conta bancária, junto com a conta deveremos registrar qual cliente é o dono dela. Vamos utilizar o **CPF do cliente na conta** para identificar o seu **dono**.

Novo tipo	Campo	Tipo do campo
Conta	numero_da_conta ★	Numérico
	saldo	Numérico
	cpf_do_cliente	Numérico

Registro em Pseudocódigo

```
REGISTRO Conta
    numero_da_conta, cpf_do_cliente, saldo: NUMÉRICO
FIM_REGISTRO
```

Código fonte /reg_conta.c[code/cap2/reg_conta.c]

Registro de Conta em C

```
typedef struct {
    long    numero_da_conta;
    long long cpf_do_cliente;
    double  saldo;
} Conta;
```

Novo tipo	Campo	Tipo do campo
Cliente	nome	Textual
	cpf ★	Numérico

Registro em Pseudocódigo

```
REGISTRO Cliente
    cpf: NUMÉRICO
    nome: TEXTUAL
FIM_REGISTRO
```

Código fonte /reg_cliente.c[code/cap2/reg_cliente.c]

Registro de Cliente em C

```
typedef struct {
    char nome[256];
    long long cpf;
} Cliente;
```

Nota

Nesta situação temos outras novidades: a criação de **dois** Registros e utilização de um campo para registrar o **relacionamento** entre os dois *registros*.^a



Percebam que `cpf` é o campo **identificador** de Cliente. Para identificar que uma conta é de um determinado cliente, utilizamos o campo identificador de cliente na conta.

Esta é uma estratégia muito importante para especificar **relacionamento** entre registros, certifique-se que compreendeu-a antes de prosseguir.

^aRelacionamento entre registros é um assunto que está fora do escopo de uma disciplina de Introdução à Programação, você estudará este tópico numa disciplina de Banco de Dados.

2.5 Exemplos de utilização dos Registros

Nesta seção veremos alguns exemplos que demonstram a utilização de registros. Nestes exemplos você irá aprender:

1. Como atribuir e acessar valores aos campos do registro;
2. Como atribuir valores de texto aos campos do registro;
3. Como ler valores da entrada e atribuí-los aos campos;
4. Como declarar um arranjo de registros;
5. Como acessar um campo num arranjo de registros.

2.5.1 Aluno

Exemplo de utilização do registro Aluno.

Código fonte /reg_aluno_exemplo.c[[code/cap2/reg_aluno_exemplo.c](#)]

Exemplo de utilização do registro Aluno

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

typedef struct {
    int matricula;
    char nome[100];
    float nota1;
    float nota2;
} Aluno;

int main(){
    Aluno aluno;
    aluno.matricula = 201328; //❶
    strncpy(aluno.nome, "Maria Bonita", sizeof(aluno.nome)); //❷
    aluno.nota1 = 8.0; //❸
    aluno.nota2 = 9.0; //❹

    printf("\n%d %s %1.2f %1.2f", aluno.matricula, aluno.nome, //❺
           aluno.nota1, aluno.nota2); //❻

    getchar();
    return 0;
}
```

❶, ❸, ❹ Como atribuir valores aos campos do registro.

❷ Como atribuir valores de texto aos campos do registro. Você já estudou a função `strncpy` antes.

❺, ❻ Como acessar valores atribuídos aos campos do registro.

Resultado ao simular a execução do programa

201328 Maria Bonita 8.00 9.00

2.5.2 Produto

Exemplo de utilização do registro Produto.

Exemplo 2.1 Utilização do registro Produto em C

Código fonte /reg_produto_exemplo.c[code/cap2/reg_produto_exemplo.c]

```
#include <stdio.h>

typedef struct {
    long    codigo;
    char    nome[100];
    float   preco;
} Produto;

int main(){
    Produto p;
    scanf("%ld %s %f", &p.codigo, p.nome, &p.preco); //❶

    if (p.preco < 4)
        printf("\nProduto em promocao: %s R$ %1.2f", p.nome, p.preco);
    else
        printf("\nProduto cadastrado.");

    getchar();
    return 0;
}
```

- ❶ Como ler da entrada os valores e atribuí-los aos campos. Consulte a documentação de `scanf` (ou `fscanf`) para conhecer a sintaxe de leitura e conversão dos dados. Percebam a ausência de `&` antes do campo `nome`.

Resultado ao simular a execução do programa

**Atenção**

Percebam que quando atribuímos um valor de texto aos campos do tipo `char[]`, nós **suprimimos** o `&`. Isto correu com o campo `aluno.nome` em `strncpy` e `p.nome` no `scanf`.

2.5.3 Pontos

Exemplo de utilização do registro Ponto com Arranjo.

Exemplo 2.2 Utilização do Registro Ponto

Código fonte /reg_ponto_exemplo.c[code/cap2/reg_ponto_exemplo.c]

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

typedef struct{
    int x;
    int y;
} Ponto;

#define QUANTIDADE_DE_PONTOS 3 // ❶

int main(){
    Ponto pontos[QUANTIDADE_DE_PONTOS]; // ❷

    pontos[0].x = -4; pontos[0].y = 7; // ❸
    pontos[1].x = 2; pontos[1].y = -9; // ❹
    pontos[2].x = 5; pontos[2].y = 3; // ❺

    for (int i = 0; i < QUANTIDADE_DE_PONTOS ; i++){
        if(pontos[i].y > 0)
            printf("\nPonto acima da reta: (%d,%d)",
                pontos[i].x, pontos[i].y);
    }

    getchar();
    return 0;
}
```

- ❶ Declaração de **constante** que definirá o tamanho do arranjo.
- ❷ Como declarar um arranjo de registros do tipo Ponto, com o tamanho definido pela constante QUANTIDADE_DE_PONTOS.
- ❸, ❹, ❺ Como acessar um campo em arranjo de registros. Cada posição, do arranjo contém um registro. Você pode acessar as posições do arranjo com a mesma sintaxe: [índice].

Resultado ao simular a execução do programa

```
Ponto acima da reta: (-4,7)
Ponto acima da reta: (5,3)
```

2.6 Exercícios resolvidos

Nesta seção teremos a especificação de diversos problemas. Para cada um deles iremos escrever um pseudocódigo que resolva o problema descrito, utilizando o recurso de Registros. Em seguida, iremos implementar um programa em C.

2.6.1 Programa do cálculo de médias de alunos

Escrever um programa que cadastre o nome, a matrícula e duas notas de vários alunos. Em seguida imprima a matrícula, o nome e a média de cada um deles.

Pseudocódigo do programa

```
REGISTRO Aluno
    matricula: NUMÉRICO
    nome: TEXTO
    nota1, nota2: NUMÉRICO
FIM_REGISTRO

QUANTIDADE_DE_ALUNOS = 3
DECLARA alunos: Aluno[QUANTIDADE_DE_ALUNOS]

PARA i=0 ATÉ QUANTIDADE_DE_ALUNOS FAÇA
    LEIA alunos[i].nome
    LEIA alunos[i].matricula
    LEIA alunos[i].nota1
    LEIA alunos[i].nota2
FIM_PARA

PARA i=0 ATÉ QUANTIDADE_DE_ALUNOS FAÇA
    ESCREVA alunos[i].matricula
    ESCREVA alunos[i].nome
    ESCREVA (alunos[i].nota1 + alunos[i].nota2)/2 ❶
FIM_PARA
```

❶ Imprime a média calculada.

Exemplo 2.3 Implementação de calculo_das_medias.c

Código fonte /calculo_das_medias.c[[code/cap2/calculo_das_medias.c](#)]

```
#include <stdio.h>

typedef struct {
    int matricula;
    char nome[100];
    float nota1;
    float nota2;
} Aluno;

#define QUANTIDADE_DE_ALUNOS 3

int main(){
    Aluno alunos[QUANTIDADE_DE_ALUNOS];

    printf("Dados: nome(sem espacos), matricula, nota1, nota2\n");
    for(int i=0; (i < QUANTIDADE_DE_ALUNOS); i++){
        printf("\nInforme os dados do aluno(%i): ",i+1);
        scanf("%s %i %f %f",alunos[i].nome, &alunos[i].matricula,
```

```
        &alunos[i].nota1, &alunos[i].nota2);
    }

    printf("\nMatricula\tNome\tMedia\n");
    for(int i=0; (i < QUANTIDADE_DE_ALUNOS); i++){
        printf("%i\t%s\t%.2f\n",alunos[i].matricula,alunos[i].nome,
            (alunos[i].nota1 + alunos[i].nota2)/2);
    }

    getchar();
    return 0;
}
```

Resultado ao simular a execução do programa

Dados do aluno: nome(sem espacos), matricula, nota1, nota2

```
Informe os dados do aluno(1): Jesuíno 2887399 6.0 7.5
Informe os dados do aluno(2): Maria 2887398 7.0 9.0
Informe os dados do aluno(3): Virgulino 2887400 10.0 8.0
Matricula   Nome      Media
2887399 Jesuíno      6.75
2887398 Maria        8.00
2887400 Virgulino    9.00
```

2.6.2 Problema do cálculo e consulta do IMC de uma pessoa

Escrever um programa que cadastre o nome, a altura, o peso, o cpf e sexo de algumas pessoas. Com os dados cadastrados, em seguida localizar uma pessoa através do seu CPF e imprimir o seu IMC.

Pseudocódigo do programa

```
REGISTRO Pessoa
    nome, sexo: TEXTO
    peso, altura, cpf: NUMÉRICO
FIM_REGISTRO

QUANTIDADE_DE_PESSOAS = 3

PARA i=0 ATÉ QUANTIDADE_DE_PESSOAS FAÇA
    LEIA pessoas[i].nome
    LEIA pessoas[i].altura
    LEIA pessoas[i].peso
    LEIA pessoas[i].cpf
    LEIA pessoas[i].sexo
FIM-PARA

DECLARA cpf_localizador: NUMÉRICO
LEIA cpf_localizador ❶

PARA i=0 ATÉ QUANTIDADE_DE_PESSOAS FAÇA ❷
    SE pessoas[i].cpf == cpf_localizador ENTÃO ❸
```



```
    ESCRIVE pessoas[i].nome
    ESCRIVE pessoas[i].sexo
    // IMC = peso / (altura * altura)
    ESCRIVE pessoas[i].peso / (pessoas[i].altura * pessoas[i].altura)
FIM-PARA
```

- ❶ O ler o campo **identificador** de Pessoa (CPF).
- ❷, ❸ Pesquisa pelo registro Pessoa identificado pelo CPF lido.

Exemplo 2.4 Implementação de `imc_calculo.c`

Código fonte `/imc_calculo.c`[\[code/cap2/imc_calculo.c\]](#)

```
#include <stdio.h>

typedef struct{
    char nome[100];
    char sexo; // 'm': masculino, 'f': feminino
    float peso;
    float altura;
    long long cpf;
} Pessoa;

#define QUANTIDADE_DE_PESSOAS 3

int main(){
    Pessoa pessoas[QUANTIDADE_DE_PESSOAS];

    printf("Campos: nome, altura, peso, cpf, sexo\n");
    for(int i=0; (i < QUANTIDADE_DE_PESSOAS); i++){
        printf("\nInforme os dados da pessoa(%i): ", i+1);
        scanf("%s %f %f %lld %c", &pessoas[i].nome, &pessoas[i].altura,
            &pessoas[i].peso, &pessoas[i].cpf, &pessoas[i].sexo);
    }

    long long cpf_localizador;
    printf("\nInforme o CPF da pessoa: ");
    scanf("%lld", &cpf_localizador); // ❶

    printf("\nSexo\tNome\tIMC");
    for(int i=0; (i < QUANTIDADE_DE_PESSOAS); i++){ // ❷
        if (cpf_localizador == pessoas[i].cpf){ // ❸
            float imc = pessoas[i].peso / (pessoas[i].altura *
                pessoas[i].altura);
            printf("\n%c\t%s\t%.2f\n", pessoas[i].sexo,
                pessoas[i].nome, imc);
            break; // ❹
        }
    }

    getchar();
    return 0;
```

```
}
```

- ❶ O ler o campo **identificador** de Pessoa (cpf).
- ❷, ❸ Pesquisa pelo registro Pessoa identificado pelo CPF lido.
- ❹ Realiza quebra na execução.

Resultado ao simular a execução do programa

Campos: nome, altura, peso, cpf, sexo

```
Informe os dados da pessoa(1): Jesuíno 1.82 79 48755891748 m
Informe os dados da pessoa(2): Maria 1.66 52 72779162201 f
Informe os dados da pessoa(3): Virgulino 1.75 80 71443626406 m
Informe o CPF da pessoa: 72779162201
Sexo      Nome      IMC
f   Maria   18.87
```

2.6.3 Problema de pontos no plano cartesiano

Escrever um programa que leia 5 pontos. Em seguida imprima qual o ponto mais próximo do primeiro ponto lido.

Pseudocódigo do programa

```
REGISTRO Ponto
  x, y: NUMÉRICO
FIM_REGISTRO

QUANTIDADE_DE_PONTOS = 5

PARA i=0 ATÉ QUANTIDADE_DE_PONTOS FAÇA
  LEIA p[i].x
  LEIA p[i].y
FIM_PARA

menor_distancia_ao_quadrado = MAIOR_INTEIRO ❶
ponto_mais_proximo = 1 ❷

PARA i=1 ATÉ QUANTIDADE_DE_PONTOS FAÇA
  distancia_ao_quadrado = (pontos[i].x-pontos[0].x)*
    (pontos[i].x-pontos[0].x)+(pontos[i].y-pontos[0].y)*
    (pontos[i].y-pontos[0].y) ❸
  SE distancia_ao_quadrado < menor_distancia_ao_quadrado ENTÃO ❹
    ponto_mais_proximo = i ❺
    menor_distancia_ao_quadrado = distancia_ao_quadrado ❻
FIM_PARA

ESCREVA p[ponto_mais_proximo].x,p[ponto_mais_proximo].y
```

- ❶, ❷, ❸ `MAIOR_INTEIRO` representa o maior número inteiro que podemos armazenar numa variável. Geralmente atribuímos o **maior** inteiro quando procuramos por **um menor** valor. No código, comparamos `menor_distancia_ao_quadrado` com `distancia_ao_quadrado` e salvamos o **menor** deles. Se executarmos isso sucessivamente, ao final, `menor_distancia_ao_quadrado` conterá o **menor** valor comparado.²
- ❷, ❹ Esta variável irá guardar a posição do ponto mais próximo. Ela é atualizada, sempre que encontramos outro ponto com menor distância.
- ❸ Cálculo para encontrar a distância entre dois pontos. Na realidade, a distância entre os dois pontos seria a raiz de `distancia_ao_quadrado`. Mas não há diferença em comparar a distância ao quadrado. Sabemos, por exemplo, que a **raiz** de x é **menor do que a raiz** de y se x for **menor** do que y .

Exemplo 2.5 Implementação de `ponto_proximo.c`

Código fonte /`ponto_proximo.c`[[code/cap2/ponto_proximo.c](#)]

```
#include <stdio.h>
#include <limits.h> // contém definição de INT_MAX

typedef struct{
    int x;
    int y;
} Ponto;

#define QUANTIDADE_DE_PONTOS 5

int main(){
    Ponto pontos[QUANTIDADE_DE_PONTOS];

    printf("Campos: x, y\n");
    for(int i=0; (i < QUANTIDADE_DE_PONTOS); i++){
        printf("\nInforme as coordenadas do ponto(%i): ", i+1);
        scanf("%d %d", &pontos[i].x, &pontos[i].y);
    }

    int menor_distancia_ao_quadrado = INT_MAX; // maior inteiro
    int ponto_mais_proximo = 1;

    for(int i=1; (i < QUANTIDADE_DE_PONTOS); i++){
        int distancia_ao_quadrado = (pontos[i].x-pontos[0].x)*
            (pontos[i].x-pontos[0].x)+(pontos[i].y-pontos[0].y)*
            (pontos[i].y-pontos[0].y);
        if(distancia_ao_quadrado < menor_distancia_ao_quadrado){
            ponto_mais_proximo = i;
            menor_distancia_ao_quadrado = distancia_ao_quadrado;
        }
    }

    printf("\nPonto mais proximo: (%d,%d)\n",
```

²Caso tivéssemos inicializado a variável `menor_distancia_ao_quadrado` com 0, ao compará-lo com outro número, ele seria o **menor**, impossibilitando encontrar a **menor** distância.

```
    pontos[ponto_mais_proximo].x, pontos[ponto_mais_proximo].y);

    getchar();
    return 0;
}
```

Resultado ao simular a execução do programa

Campos: x, y

```
Informe as coordenadas do ponto(1): 0 0
Informe as coordenadas do ponto(2): 4 6
Informe as coordenadas do ponto(3): 6 1
Informe as coordenadas do ponto(4): 5 3
Informe as coordenadas do ponto(5): 7 2
Ponto mais proximo: (5,3)
```

2.6.4 Problema sobre cadastro de produtos no supermercado

Escrever um programa que cadastre vários produtos. Em seguida, imprima uma lista com o código e nome de cada produto. Por último, consulte o preço de um produto através de seu código.

Pseudocódigo do programa

```
REGISTRO Produto
    codigo: NUMÉRICO
    nome: TEXTUAL
    preco: NUMÉRICO
FIM_REGISTRO

QUANTIDADE_DE_PRODUTOS = 5
DECLARA produtos: Produto[QUANTIDADE_DE_PRODUTOS]

PARA i=0 ATÉ QUANTIDADE_DE_PRODUTOS FAÇA
    LEIA produtos[i].codigo
    LEIA produtos[i].nome
    LEIA produtos[i].preco
FIM_PARA

PARA i=0 ATÉ QUANTIDADE_DE_PRODUTOS FAÇA
    ESCRIVA produtos[i].codigo
    ESCRIVA produtos[i].nome
FIM_PARA

DECLARA codigo_digitado: NUMÉRICO
LEIA codigo_digitado

PARA i=0 ATÉ QUANTIDADE_DE_PRODUTOS FAÇA
    SE produtos[i].codigo == codigo_digitado ENTÃO
        ESCRIVA produtos[i].preco
FIM_PARA
```

Exemplo 2.6 Implementação de supermercado.c**Código fonte** /supermercado.c[[code/cap2/supermercado.c](#)]

```
#include <stdio.h>

typedef struct {
    long   codigo;
    char   nome[100];
    float  preco;
} Produto;

#define QUANTIDADE_DE_PRODUTOS 5

int main(){
    Produto produtos[QUANTIDADE_DE_PRODUTOS];

    printf("Campos: codigo-do-produto nome preco\n");
    for(int i=0; (i < QUANTIDADE_DE_PRODUTOS); i++){
        printf("\nInforme os dados do produto(%i): ",i+1);
        scanf("%ld %s %f",&produtos[i].codigo,produtos[i].nome,
            &produtos[i].preco);
    }

    for(int i=0; (i < QUANTIDADE_DE_PRODUTOS); i++){
        printf("\n%ld\t%s R$ %1.2f", produtos[i].codigo,
            produtos[i].nome,produtos[i].preco);
    }

    long codigo_digitado;
    printf("\nInforme o codigo do produto: ");
    scanf("%ld", &codigo_digitado);

    for(int i=1; (i < QUANTIDADE_DE_PRODUTOS); i++){
        if (produtos[i].codigo == codigo_digitado) {
            printf("\nPreço: R$ %1.2f\n", produtos[i].preco);
        }
    }

    getchar();
    return 0;
}
```

Resultado ao simular a execução do programa

Campos: codigo-do-produto nome preco

```
Informe os dados do produto(1): 1 laranja 1.4
Informe os dados do produto(2): 2 rosquinha 3
Informe os dados do produto(3): 3 leite-moca 4.5
Informe os dados do produto(4): 4 farinha-de-trigo 2.7
Informe os dados do produto(5): 5 coxinha 1.5
1   laranja R$ 1.40
2   rosquinha R$ 3.00
```

```
3    leite-moca R$ 4.50
4    farinha-de-trigo R$ 2.70
5    coxinha R$ 1.50
Informe o código do produto: 4
Preço: R$ 2.70
```

2.6.5 Problema sobre gerenciamento de contas bancárias

Escreva um programa que simule contas bancárias, com as seguintes especificações:

- Ao iniciar o programa vamos criar contas bancárias para três clientes.
 - Cada conta terá o nome e o CPF do cliente associado a ela.
 - No ato da criação da conta o cliente precisará fazer um depósito inicial.
- Após as contas serem criadas, o sistema deverá possibilitar realizações de saques ou depósitos nas contas.
 - Sempre que uma operação de saque ou depósito seja realizada, o sistema deverá imprimir o nome do titular e o saldo final da conta.

Pseudocódigo do programa

```
REGISTRO Conta
    numero_da_conta, cpf_do_cliente, saldo: NUMÉRICO
FIM_REGISTRO

REGISTRO Cliente
    cpf: NUMÉRICO
    nome: TEXTUAL
FIM_REGISTRO

QUANTIDADE_DE_CLIENTES = 3
DECLARA clientes: Cliente[QUANTIDADE_DE_CLIENTES]
DECLARA contas: Conta[QUANTIDADE_DE_CLIENTES]

PARA i=0 ATÉ QUANTIDADE_DE_CLIENTES FAÇA
    LEIA clientes[i].cpf
    LEIA clientes[i].nome
    LEIA contas[i].saldo // depósito inicial

    clientes[i].codigo = i
    contas[i].numero_da_conta = i
    contas[i].codigo_do_cliente = clientes[i].codigo
FIM_PARA

DECLARA operacao: TEXTUAL
DECLARA num_conta, valor, sair=0: NUMÉRICO

ENQUANTO sair == 0 FAÇA
```

```
LEIA operacao

SE operacao == "saque" OU operacao == "deposito" ENTÃO
    LEIA num_conta, valor
    PARA i=0 ATÉ QUANTIDADE_DE_CLIENTES FAÇA
        SE contas[i].numero_da_conta == num_conta ENTÃO
            SE operacao == "saque" ENTÃO
                contas[i].saldo = contas[i].saldo - valor
            SE operacao == "deposito" ENTÃO
                contas[i].saldo = contas[i].saldo + valor
        PARA j=0 ATÉ QUANTIDADE_DE_CLIENTES FAÇA
            SE clientes[j].codigo == contas[i].codigo_do_cliente ENTÃO
                ESCRIBE clientes[j].nome, contas[i].saldo
        FIM_PARA
    FIM_PARA
SENÃO operacao == "sair" ENTÃO
    sair = 1
FIM_ENQUANTO
```

Exemplo 2.7 Implementação de conta_bancaria.c

Código fonte /conta_bancaria.c[[code/cap2/conta_bancaria.c](#)]

```
#include <stdio.h>

typedef struct {
    char nome[256];
    long long cpf;
} Cliente;

typedef struct {
    long numero_da_conta;
    long long cpf_do_cliente;
    double saldo;
} Conta;

#define QUANTIDADE_DE_CLIENTES 3
#define OPERACAO_SAQUE 1
#define OPERACAO_DEPOSITO 2

int main(){
    Cliente clientes[QUANTIDADE_DE_CLIENTES];
    Conta contas[QUANTIDADE_DE_CLIENTES];

    printf("Campos: cpf nome deposito-inicial\n");
    for(long i=0; (i < QUANTIDADE_DE_CLIENTES); i++){
        printf("\nDados para abertura da conta(%ld): ", i+1);
        scanf("%lld %s %lf", &clientes[i].cpf, clientes[i].nome,
            &contas[i].saldo);

        contas[i].numero_da_conta = i;
        contas[i].cpf_do_cliente = clientes[i].cpf;

        printf("\nCliente: %s Conta: %ld Saldo inicial: %1.2lf\n",
```

```
        clientes[i].nome, contas[i].numero_da_conta, contas[i].saldo);
    }

    int operacao; // como ainda não aprendemos a comparar strings,
                  // vamos usar 'operação' como numérico.
    long num_conta;
    double valor;
    int sair=0; // FALSE

    while (!sair){ // ❶
        printf("\nInforme a operação: 1-Saque 2-Deposito 3-Sair: ");
        scanf("%d", &operacao);

        if (operacao == OPERACAO_SAQUE || operacao == OPERACAO_DEPOSITO){
            printf("\nInforme numero-da-conta e valor: ");
            scanf("%ld %lf", &num_conta, &valor);
            for(int i=0; (i < QUANTIDADE_DE_CLIENTES); i++){
                if (contas[i].numero_da_conta == num_conta) {
                    if (operacao == OPERACAO_SAQUE){
                        contas[i].saldo -= valor;
                        printf("\nSAQUE: %1.2lf", valor);
                    }
                    if (operacao == OPERACAO_DEPOSITO){
                        contas[i].saldo += valor;
                        printf("\nDEPOSITO: %1.2lf", valor);
                    }
                    for(int j=0; j < QUANTIDADE_DE_CLIENTES; j++){
                        if (clientes[j].cpf == contas[i].cpf_do_cliente)
                            printf("\nCliente: %s Saldo atual: %1.2lf",
                                    clientes[j].nome, contas[i].saldo);
                    }
                }
            }
        }else{
            sair = 1; // TRUE
        }
    }

    getchar();
    return 0;
}
```

Resultado ao simular a execução do programa

Campos: cpf nome deposito-inicial

Dados para abertura da conta(1): 48755891748 Jesuíno 1500
Cliente: Jesuíno Conta: 0 Saldo inicial: 1500.00

Dados para abertura da conta(2): 72779162201 Maria 200
Cliente: Maria Conta: 1 Saldo inicial: 200.00

Dados para abertura da conta(3): 71443626406 Virgulino 600

Cliente: Virgulino Conta: 2 Saldo inicial: 600.00

Informe a operação: 1-Saque 2-Deposito 3-Sair: 1

Informe numero-da-conta e valor: 0 300

SAQUE: 300.00

Cliente: Jesuíno Saldo atual: 1200.00

Informe a operação: 1-Saque 2-Deposito 3-Sair: 2

Informe numero-da-conta e valor: 2 400

DEPOSITO: 400.00

Cliente: Virgulino Saldo atual: 1000.00

Informe a operação: 1-Saque 2-Deposito 3-Sair: 3

Após todos estes programas, agora vamos ver uma técnica não utilizada ainda, a inicialização de *registro* com valores pré-definidos.

2.7 Inicializando registros

Quando declaramos uma variável do tipo registro, também podemos realizar uma atribuição aos valores dos seus campos. O programa a seguir ilustra esta atribuição.



Atenção

Para a atribuição poder ocorrer, os campos precisam ser inseridos **na ordem que foram declarados** no tipo do registro.

Exemplo 2.8 Programa em C

Código fonte /reg_atribuicao.c[[code/cap2/reg_atribuicao.c](#)]

```
#include <stdio.h>

typedef struct {
    int matricula; // ❶
    char nome[100]; // ❷
    float nota1; // ❸
    float nota2; // ❹
} Aluno;

typedef struct {
    char nome[256]; // ❺
    long long cpf; // ❻
} Cliente;

int main() {
    Aluno a = {15, "Virgulino da Silva", 9.0f, 10.0f}; // ❼
    Cliente c = {"Maria Bonita", 72779162201}; // ❽

    printf("Aluno: %s Mat.: %d Nota1: %1.2f Nota2: %1.2f\n",
        a.nome, a.matricula, a.nota1, a.nota2);
    printf("Cliente: %s CPF: %11lld\n", c.nome, c.cpf);
```

```
    return 0;
}
```

❶, ❶, ❷, ❸, ❹, ❺ Seguindo a ordem da declaração do registro, `matricula` recebe 15, `nome` recebe “Virgulino da Silva”, `nota1` recebe 9 e `nota2` recebe 10.

❻, ❻, ❸ Seguindo a ordem da declaração do registro, `nome` recebe “Maria Bonita” e `cpf` recebe 72779162201.

Resultado ao simular a execução do programa

Aluno: Virgulino da Silva Mat.: 15 Nota1: 9.00 Nota2: 10.00
Cliente: Maria Bonita CPF: 72779162201



Nota

O **Registro** é um tipo de dado composto por campos com outros tipos. *Mas será que é possível declarar um campo do tipo Registro? Veremos a resposta na próxima seção.*

2.8 Composição de Registros

Na definição de registros (Seção 2.1 [20]), vimos que um Registro é criado pela **composição** de outros tipos de dado. Agora veremos que podemos compor um Registro utilizando outros Registros **previamente definidos**.



Cuidado

Ao realizar composição de registros, a definição do registro que será utilizado na composição precisa **aparecer antes** (no código fonte) da definição do novo registro. Caso contrário, você poderá ter erros de compilação.

2.8.1 Triângulo

Nesta seção vamos definir um Registro triângulo que contém **3** campos do tipo `Ponto`.

Composição de registro em Pseudocódigo

```
REGISTRO Ponto
    x, y: NUMÉRICO
FIM_REGISTRO

REGISTRO Triangulo
    p1, p2, p3: Ponto
FIM_REGISTRO
```

Exemplo 2.9 Composição de registro em C - Triângulo

Código fonte /reg_triangulo.c[code/cap2/reg_triangulo.c]

```
#include <stdio.h>

typedef struct {
    int x;
    int y;
} Ponto ;

typedef struct {
    Ponto p1;
    Ponto p2;
    Ponto p3;
} Triangulo ;

int main() {
    Triangulo t;
    t.p1.x= 1; t.p1.y=0;
    t.p2.x=-1; t.p2.y=0;
    t.p3.x= 0; t.p3.y=1;

    printf("Triangulo: (%d, %d), (%d, %d), (%d, %d).\n",
        t.p1.x, t.p1.y, t.p2.x, t.p2.y, t.p3.x, t.p3.y);

    return 0;
}
```

Nota

Neste exemplo, o registro do tipo `Triangulo` foi criado com campos do tipo `Ponto`, os três campos foram: `p1`, `p2` e `p3`. Para acessar a coordenada `x` do primeiro ponto do `Triangulo` `t`, chamamos: `t.p1.x`.

Foram dispostas duas atribuições de coordenadas numa mesma linha apenas para ficar melhor visualmente, não há necessidade de serem assim.

2.8.2 Informação Pessoal

Nesta seção vamos definir um Registro `InformacaoPessoal` e utilizá-lo no Registro `Aluno` e `Cliente`.

Composição de registro em Pseudocódigo

```
REGISTRO InformacaoPessoal
    cep: NUMÉRICO
    estado_civil: TEXTO
FIM_REGISTRO

REGISTRO Aluno
    matricula: NUMÉRICO
```

```
nome: TEXTO
nota1, nota2: NUMÉRICO
info_pessoal: InformacaoPessoal
FIM_REGISTRO
```

```
REGISTRO Cliente
  cpf: NUMÉRICO
  nome: TEXTUAL
  info_pessoal: InformacaoPessoal
FIM_REGISTRO
```

Exemplo 2.10 Composição de registro em C - Informação Pessoal

Código fonte /reg_infopessoal.c[[code](#)/cap2/reg_infopessoal.c]

```
#include <stdio.h>

typedef struct {
    long long cep;
    int estado_civil; // 1:Solteiro 2:Casado 3:Viuvo 4:Divorciado
} InformacaoPessoal;

typedef struct {
    int matricula;
    char nome[100];
    float nota1;
    float nota2;
    InformacaoPessoal info_pessoal;
} Aluno;

typedef struct {
    char nome[256];
    long long cpf;
    InformacaoPessoal info_pessoal;
} Cliente;

int main() {
    Aluno a = {15, "Virgulino da Silva", 9.0f, 10.0f, {58051400, 1}};
    Cliente c = {"Maria Bonita", 72779162201, {58051400, 2}};

    printf("Aluno: %s %8lld %d.\n", a.nome, a.info_pessoal.cep,
           a.matricula);
    printf("Cliente: %s %8lld %11lld.\n", c.nome, c.info_pessoal.cep,
           c.cpf);

    return 0;
}
```

Nota

A composição de Registro utiliza a sintaxe usual de declaração de campos. Uma vez que definimos um novo tipo, basta utilizar o tipo na declaração normal do campo.

O acesso aos campos internos do registro passam pelo campo definido no registro externo, por exemplo, para acessar o interno `cep`, primeiro precisamos referenciar o campo externo `info_pessoal`, portanto o acesso fica: `a.info_pessoal.cep`.

Para finalizar nossos estudos sobre Registro, na seção seguinte vamos compará-lo com Arranjo.

2.9 Comparação entre Array e Registro

A tabela a seguir mostra uma comparação entre Arranjos e Registros.

Arranjo (ou array)	Registro
<ul style="list-style-type: none">• Estrutura de dados homogênea<ul style="list-style-type: none">– Arranjo de variáveis referenciadas por um mesmo nome e indexada por um inteiro. Ex: <code>notas[i]</code>.• Armazena vários valores, mas todos do mesmo tipo.	<ul style="list-style-type: none">• Estrutura de dados heterogênea<ul style="list-style-type: none">– Coleção de variáveis referenciadas por um mesmo nome• Armazena vários valores e podem ser de diferentes tipos• Cada valor é armazenado num campo com um tipo próprio

2.10 Recapitulando

Iniciamos este capítulo conhecendo a definição de **Registro** e sua utilidade.

Em seguida aprendemos a sua **sintaxe** de criação. Vimos o que é um campo **identificador** e como ele é utilizado para diferenciar um registro de outro.

Realizamos **análises** em 5 situações demonstrando como criamos registros em cada uma delas.

Na Seção 2.6 [31] vimos como implementar diversos programas em pseudocódigo e em C.


Por fim, aprendemos como um registro pode ser inicializado (Seção 2.7 [42]), comparamos os registros com os arranjos (Seção 2.9 [46]) e aprendemos como criar um registro através da composição de outro (Seção 2.8 [43]).

2.11 Atividades



Dica

Na sala de aula, nos fóruns e mensagens recomendamos a utilização do site <https://gist.github.com> para compartilhar códigos e tirar dúvidas sobre eles.

1. **Entendo a necessidade dos registros.** Nós poderíamos escrever os programas sem utilizar registros. Qual a utilidade de se utilizar registros nos programas?
2. O que é um campo **identificador**? Dê exemplos **não** contidos neste capítulo.
3. Na Seção 2.4 [23] analisamos diversas situações buscando os campos necessários para criação de Registros. Agora chegou a sua vez de fazer o mesmo. Para cada situação a seguir:
 - Defina o(s) **nome(s)** do tipo de registro que você criará
 - Especifique os **campos** com seus respectivos **tipos**
 - Indique quais são os campos **identificadores**, caso exista
 - Escreva as **declarações** do(s) Registro(s) em C
 - a. Um programa para registrar os animais e os clientes de um Petshop.
 - b. Um programa para registrar e consultar filmes.
 - c. Um programa para uma biblioteca registrar os seus livros.
 - d. Um programa para agendar e consultar compromissos. 
4. **Pratique o uso de registros.** Utilizando os registros definidos no capítulo, faça pequenos programas e teste com as entradas indicadas.³

- a. Utilizando o Registro de Aluno em C [24], faça um programa que indique a situação de cada Aluno: **Aprovado**, se média das notas for maior ou igual 5; **Reprovado** se a média for inferior a 5; **Aprovado com menção de honra** se média for superior ou igual a 9.

Entrada: nome matrícula nota1 nota2

```
Maria 12887398 7.0 9.0
Jesuino 12887399 3.5 6.0
Virgulino 12887400 10.0 8.0
```

- b. Utilizando o Registro de Pessoa em C [25], escreva um programa que imprima o IMC das mulheres, depois o dos homens. Quando imprimir o IMC, exiba uma mensagem indicando a condição em que a pessoa se encontra, de acordo com a tabela a seguir.

IMC	Classificação
abaixo de 18,5	Subnutrido ou abaixo do peso
entre 18,6 e 24,9	Peso ideal (parabéns)
entre 25,0 e 29,9	Levemente acima do peso
entre 30,0 e 34,9	Primeiro grau de obesidade
entre 35,0 e 39,9	Segundo grau de obesidade
acima de 40	Obesidade mórbida

³Testar com valores pré-definidos facilita o desenvolvimento dos programas, faça disso um hábito. Veremos mais adiante como redirecionar a entrada do programa, facilitando ainda mais os testes.

Entrada: nome altura peso cpf sexo

```
Jesuino 1.82 79 48755891748 m
Maria 1.66 52 72779162201 f
Virgulino 1.75 80 71443626406 m
```

- c. Utilizando o Registro de Ponto em C [26], escreva um programa que leia alguns pontos, indicando em qual quadrante eles estão no plano cartesiano.

Entrada: p1.x p1.y p2.x p2.y p3.x p3.y

```
1 3
-2 4
-3 -3
2 -5
4 0
```

- d. Utilizando o Registro de Produto em C [27], escreva um programa que cadastra uma lista de produtos. Em seguida imprima os produtos ordenadamente pelo menor preço.

Entrada: codigo nome valor

```
11 laranja 1.4
12 rosquinha 3
13 leite-moca 4.5
14 farinha-de-trigo 2.7
15 coxinha 1.5
```

5. **Criando novos Registros.** Agora que você já praticou a utilização de Registro está no momento de criar os seus próprios Registros.

Importante

As questões a seguir **não** especificam os programas minuciosamente, elas foram elaboradas assim para permitir que você expresse a sua criatividade. No entanto, você deve:



- Resolver as questões utilizando os conhecimentos adquiridos neste capítulo
 - Utilizar composição de registros quando for apropriado
 - Preparar valores de entradas fixos para o seu programa, de forma a testá-lo eficientemente.
-

- a. Faça um programa para um Petshop, para cadastrar os clientes da loja e seus animais. O programa deve possibilitar pesquisa pelo cliente ou pelo seu animal.
- b. Faça um programa para gerenciar os gastos pessoais. O programa deve poder registrar os gastos por categoria e emitir um relatório para proporcionar um controle financeiro.
- c. Faça um programa para registrar os filmes que você assistiu ou quer assistir. Os filmes devem ser cadastrados por categorias. O programa deve emitir listas de filmes com base em dois critérios à sua escolha.
- d. Faça um programa para auxiliar a Policia Federal acompanhar as explosões de caixas eletrônicos ao longo do tempo. Após cadastrar as explosões, o sistema deve informar as regiões críticas.

- e. Faça um programa para simular um dicionário. Ele deve cadastrar algumas palavras e possibilitar alguma forma de navegação. Consulte um dicionário real para verificar que além do significado da palavra, outras informações diferentes também são cadastradas.



Feedback sobre o capítulo

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

Para compreender melhor como feedbacks funcionam consulte o guia do curso.

Capítulo 3

Ponteiros

OBJETIVOS DO CAPÍTULO

Ao final deste capítulo você deverá ser capaz de:

- Entender o conceito de ponteiros e utilizá-los em funções, vetores, registros ou estruturas, dentre outras;
- Aprofundar o conhecimento sobre armazenamento e manipulação de dados no computador;

No início do Capítulo 1, foi discutido que o conhecimento de linguagens de programação específicas, por si só, não habilita ou não é suficiente para programadores, é imprescindível saber usá-las de maneira eficiente. Foi visto também que o projeto de um algoritmo ou programa engloba, entre outras, a fase da modelagem computacional de um problema, a qual por sua vez possui uma etapa de identificação das propriedades dos dados (ou informações contidas no problema) e suas características funcionais. Dessa forma, uma representação adequada dos dados ou informações do problema, em vista das funcionalidades que devem ser atendidas, constitui uma etapa fundamental para a obtenção de programas eficientes.

Neste capítulo serão abordados os conceitos de ponteiros e a sua aplicação para alocação dinâmica de memória. Será visto o uso de ponteiros em funções ou sub-rotinas, vetores, registros ou estruturas, vetores de registros, dentre outros.

3.1 Armazenamento de dados no computador

Para armazenar dados no computador um programa gerencia fundamentalmente três parâmetros, a saber, onde a informação está armazenada, que tipo de informação é armazenada e que valor é mantido lá. O primeiro refere-se ao endereço de memória onde os dados são gravados. O segundo refere-se ao tipo de dado, ou seja, qual o tipo de estrutura (variável, array, registros, dentre outras) e qual a natureza da informação armazenada (real, inteiro, caracteres, número sem sinal, dentre outras), essas informações são importantes para se calcular o espaço de memória que será necessário para o armazenamento desses dados. O último parâmetro refere-se aos valores que são atribuídos pelo usuário ou calculados no próprio programa e que serão armazenados nas estruturas de dados. O exemplo mais simples para ilustrar o que foi colocado, é o processo de declaração de uma variável num programa.

Como ilustrado na Figura 3.1 [51], nesse caso, são realizados basicamente dois passos, a declaração da variável em si e a atribuição de um valor a essa variável. Note na Figura 3.1 [51] que no momento em que a variável `soma` é declarada, é alocado na memória RAM do computador um espaço de memória para essa variável (posições `0xCB22` e `0xCB23`). Esse espaço de memória é preenchido quando um valor é atribuído a variável, como por exemplo, o valor 3126.

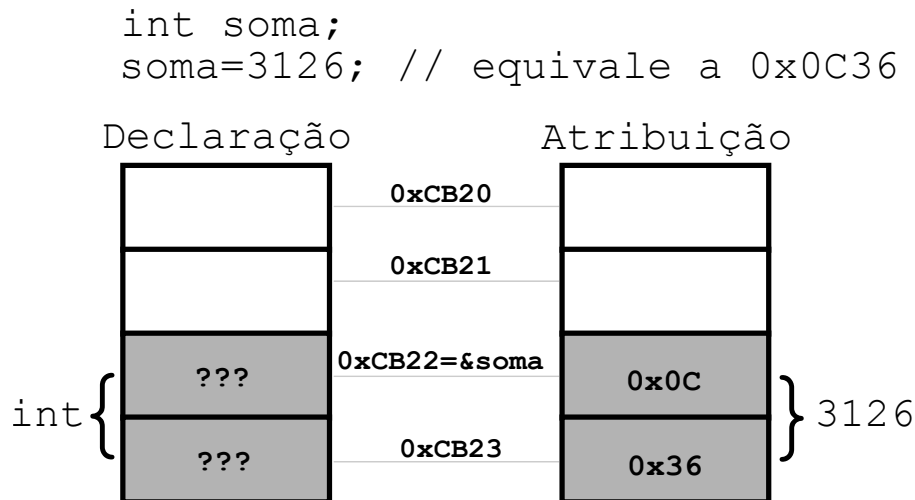


Figura 3.1: Ilustração de uso da memória por uma variável em um programa em C.

Um cuidado importante que todo programador deve ter é inicializar as variáveis antes do seu uso. Veja no Exemplo 3.1 [51] que, quando não se inicializa uma variável, o valor que ela assume é aleatório e depende do “lixo” no espaço de memória onde foi criada. No momento em que esse código foi criado, compilado e executado o valor de `conta` foi 624.756.000,00!¹ Ou seja, um valor totalmente aleatório e que muda a cada vez que a memória RAM do computador é inicializada. O correto é inicializar a variável antes de utilizá-la. Como exercício, compile e execute o código a seguir com e sem o comentário indicado. Certamente, com a linha comentada, o valor impresso na tela será diferente em cada computador em que esse código for executado.

Exemplo 3.1 Programa para demonstrar valor de variável não inicializada

Código fonte `/variavel_nao_inicializada.c`[code/cap3/variavel_nao_inicializada.c]

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

int main() {
    float conta; // representa o valor da conta de luz da sua casa
    //conta=65.90; // ❶ Descomente este linha
    printf("O valor da conta esse mês é %0.2f. \n", conta);
    //system("pause");
    return EXIT_SUCCESS;
}
```

Saída do programa

O valor da conta esse mês é 624756000.00.

¹Você iria trabalhar a vida inteira para pagar essa conta de luz!

Dessa forma, todo nome de variável está associado a um endereço na memória. O **operador de endereço** & pode ser usado para obter a localização de uma variável na memória, ou seja, o endereço de memória. O Exemplo 3.2 [52] mostra como esse operador pode ser utilizado para se obter o endereço de memória onde uma variável foi criada. Note que no `printf`, o `%p` foi utilizado como formato para endereço de memória.

Exemplo 3.2 Programa para demonstrar uso do Operador de Endereço

Código fonte /operador_de_endereco.c[code/cap3/operador_de_endereco.c]

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

int main() {
    int pg=120;
    double livro=145.60;
    printf("Valor de pg = %d. \n", pg);
    printf("Endereço de pg = %p. \n", &pg);
    printf("Valor de livro = %lf. \n", livro);
    printf("Endereço de livro = %p. \n", &livro);
    //system ("pause");
    return EXIT_SUCCESS;
}
```

Saída da execução do programa

```
Valor de pg = 120.
Endereço de pg = 0xbfda0dac.
Valor de livro = 145.600000.
Endereço de livro = 0xbfda0da0.
```

3.2 Definição de ponteiros

Outra estratégia para armazenar e manipular dados no computador é:

- Alocar memória manualmente;
- Guardar o endereço de memória em um ponteiro;
- Usar o ponteiro para acessar e modificar os dados.

Ou seja, é necessário definir e utilizar ponteiros no programa. Um ponteiro é um tipo especial (“variável especial”) que guarda valores que são endereços de memória. Em outras palavras, um ponteiro é uma variável que é utilizada para se armazenar endereços de memória. Em geral, um ponteiro é associado a uma variável, ou seja, o ponteiro contém um endereço de uma variável que contém um valor específico. A referência de um valor por meio de um ponteiro é chamada de **indireção**.

A declaração de um ponteiro segue o seguinte padrão:

```
float *ptr; // Lê-se "ptr é um ponteiro para um float"
```

Em que `float` especifica o tipo de elemento apontado pelo ponteiro, `ptr` é o nome que está sendo dado ao ponteiro, `*` é o **operador de indireção** e é usado para se definir um ponteiro.

Como o ponteiro contém um endereço de memória, diz-se que ele aponta para aquela posição de memória. A Figura 3.2 [53] ilustra a definição de um ponteiro com nome `ptr`, que é utilizado para apontar para a variável `float soma`. Note que na linha 4 `ptr` é declarado. Até esse momento não existe nenhuma relação entre o ponteiro `ptr` e a variável `soma`. No final, `ptr` é associado a variável `soma` por meio do comando `ptr = &soma`, ou seja, `ptr` recebe o endereço de memória da variável `soma` usando o **operador de endereço** `&`. A partir desse comando, o programador pode utilizar o ponteiro `ptr` para manipular o conteúdo armazenado na variável `soma`. A seguir será visto como isso é feito.

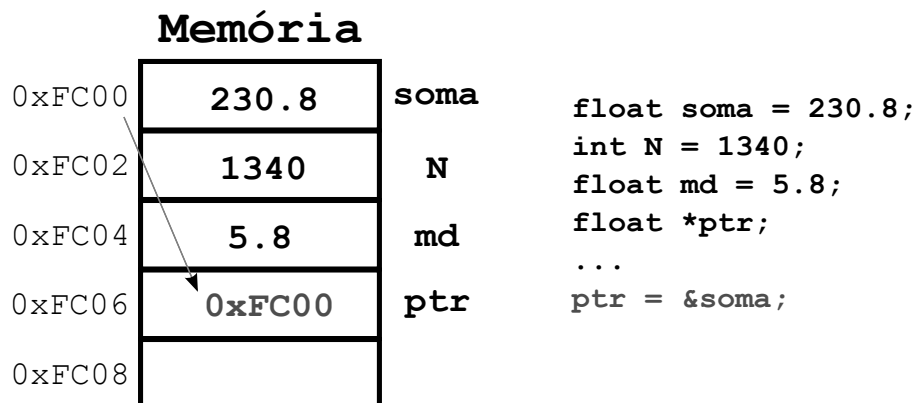


Figura 3.2: Ilustração de declaração e inicialização de um ponteiro na memória

O nome de um ponteiro representa uma localização na memória. Outra função do **operador de indireção** `*` é ser usado para se obter o valor armazenado na memória. Ou seja, é a partir do operador `*` que o programador pode manipular o conteúdo da variável para qual o ponteiro está apontando.



Importante

Sempre inicialize um ponteiro que você declarou em um programa. Um ponteiro só terá utilidade em um programa quando for inicializado, ou seja, for associado a algum elemento.

Um ponteiro pode ser inicializado com `NULL`, 0 (zero) ou um endereço de uma variável usando o operador de endereço `&`. Um ponteiro com valor `NULL` não aponta para nada. `NULL` é uma constante simbólica definida na biblioteca `<stddef.h>` e também em `<stdio.h>`. A inicialização do ponteiro com o valor 0 (zero) é equivalente a inicializar com `NULL`, todavia, em geral, usa-se a inicialização com `NULL`.



Nota

Outras linguagens de programação utilizam `nil` ou `null` com o mesmo significado.

Outra maneira de se inicializar ponteiros é associando-o diretamente a uma variável. Para isso o operador de endereço `&` deve ser utilizado. A Figura 3.3 [54](a) ilustra que, ao declarar um ponteiro, o computador não aloca automaticamente memória para guardar o valor apontado e, nesse caso,

o comando `*pt = 230.8` não faz sentido. É preciso inicializar o ponteiro como mostrado na Figura 3.3 [54](b), ou seja, o ponteiro `pt` é declarado e associado a variável `N1`. Em seguida, pode-se utilizar o ponteiro para armazenar o valor 230.8 na variável `N1`.

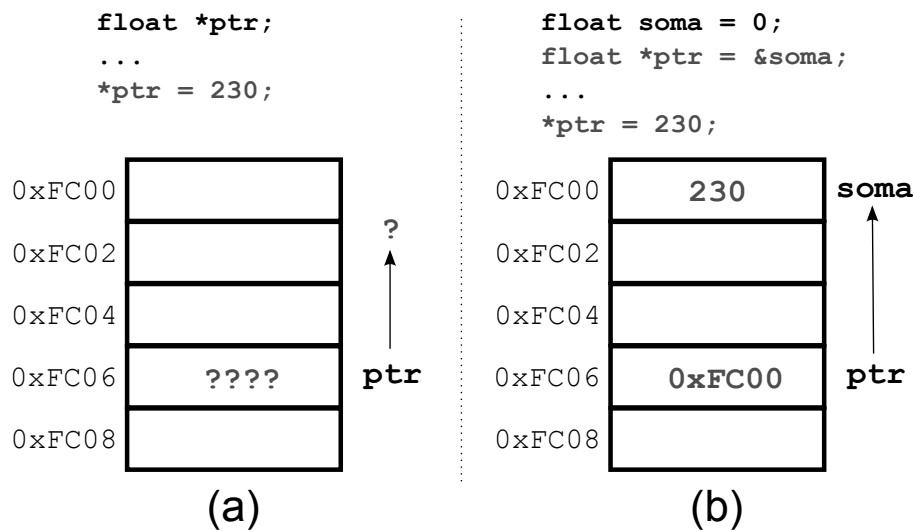


Figura 3.3: Utilização de ponteiro.



Atenção

Na Figura 3.3 [54](a) é demonstrada uma forma errada da utilização de ponteiro, ao atribuir um valor a um ponteiro não inicializado. Em b) é apresentada a forma correta, com atribuição de valor após ponteiro inicializado.



Nota

Na declaração de um ponteiro o uso de espaços ao redor do `*` é opcional.

```
int *pt; // enfatiza que *pt é um int
int* pt; // enfatiza que pt é um ponteiro para int
int * pt; // estilo neutro
```

Cuidado

Cuidado com declarações múltiplas.



```
char *p1, p2; // p1 é um ponteiro para char,
               // p2 é uma variável do tipo char
```

```
char *p1, *p2; // p1 e p2 são ponteiros para char
```

Com o intuito de sedimentar o conceito de ponteiros, o Exemplo 3.3 [55] mostra como um ponteiro é usado para manipular o conteúdo da variável para qual está apontando. Execute esse programa no seu computador e faça modificações à vontade para você melhor absorver o conceito de ponteiros.

Exemplo 3.3 Usando ponteiro para alterar variável**Código fonte** /usando_ponteiro.c[[code/cap3/usando_ponteiro.c](#)]

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

int main() {
    int total = 18;
    int *pt; // ❶
    pt = &total; // ❷

    printf("Valor de total = %d\n", total); // ❸
    printf("Endereço de total = %p\n", &total); // ❹
    printf("Valor de pt = %p\n", pt); // ❺
    printf("Valor de *pt = %d\n", *pt); // ❻

    *pt = *pt + 12; // ❼

    printf("Apos soma usando ponteiro: total=%d\n", total); // ❽
    //system("pause");
    return EXIT_SUCCESS;
}
```

- ❶, ❶ o ponteiro `pt` é criado
- ❷ `pt` é inicializado com o endereço de `total`
- ❸, ❹ Impressão do **valor** de `total` e seu **endereço**
- ❺ Impressão do **endereço** apontado por `pt`
- ❻ Impressão do **valor** endereçado por `pt`
- ❼ operador de indireção `*` é utilizado para alterar o valor da variável `total` por meio do ponteiro
- ❽ Impressão do **valor** da variável `total`, que teve seu valor alterado pela utilização de ponteiro.

Saída da execução do programa

```
Valor de total = 18
Endereço de total = 0xbfd286a8
Valor de pt = 0xbfd286a8
Valor de *pt = 18
Apos soma usando ponteiro: total=30
```

Finalmente, recapitulando, note que há dois operadores que são utilizados para a manipulação e/ou definição de ponteiros, a saber, o operador `&`, conhecido como **operador de endereço**, e o operador `*`, normalmente chamado de **operador de indireção**.

O operador de endereço

é utilizado para se obter o endereço de seu operando ou variável a que é aplicado.

O operador de indireção

pode ser utilizado de duas formas, a primeira para definição de um ponteiro; já a segunda função é retornar o valor ou conteúdo da variável para a qual um ponteiro está apontando.

3.3 Ponteiros em funções ou sub-rotinas

3.3.1 Contextualização

Para entendermos a utilidade do uso de ponteiros em funções é preciso antes analisar aspectos da execução de uma função em um programa em C, como, por exemplo, a **pilha de execução** de uma **função** e transferência de dados para o programa principal (função `main`).

As funções em um programa em C são independentes entre si. Como consequência, as **variáveis locais** definidas dentro do **escopo de uma função**, inclusive os parâmetros de entrada da função, não existem fora dela. Isso significa que cada vez que uma função é executada, as suas variáveis locais são criadas, e após a sua execução, essas variáveis locais deixam de existir.

Outra característica básica no conceito de funções é que a transferência de dados entre funções é feita com o uso de parâmetros e do valor de retorno da função chamada. Ou seja, uma função pode retornar um valor para a função que a chamou por meio do comando `return`.

Para entendermos melhor a transferência de dados entre funções e o fluxo de dados dentro do escopo de uma função, um exemplo detalhado é dado e sua execução é analisada usando o esquema ilustrado na Figura 3.4 [56]. Essa figura esboça o conceito de **pilha de execução** na memória RAM do computador, na qual os nomes das variáveis e/ou das funções são mostrados à direita de cada célula/linha, o valor/conteúdo que uma variável assume no momento da execução da função é mostrado em cada célula/linha da coluna e, finalmente, o endereço de memória onde cada variável está gravada é mostrado à esquerda de cada célula/linha.

Endereço	Memória	Variáveis
0x0101		
0x0100	35.20	iac
0x0011	'M'	sexo
0x0010	120	quadril
0x0001	172	altura
0x0000	94.0	peso

Figura 3.4: Ilustração da pilha de execução na memória RAM do computador.

O Exemplo 3.4 [57] mostra um programa que calcula a soma dos números pares entre 0 e 200. A etapa de cálculo da soma é feita na função `f_soma`, a qual é chamada na linha 10 dentro da *função principal* (função `main` ou função chamadora). Como dito anteriormente, quando uma função é executada, as suas *variáveis locais* são criadas, e após a sua execução, essas variáveis locais deixam de existir. Note no código exemplo que, após a execução da função `f_soma`, a variável `PAR` dentro da função `f_soma` terá valor 0 (zero). Ou seja, diferente do valor 200 que `PAR` possui após a execução da função `main`, já que seu valor não foi alterado dentro da função principal. O que acontece é

que PAR (parâmetro da função `f_soma`) é uma *variável local* inicializada com o valor passado na chamada da função. Sendo assim, PAR dentro da função `f_soma` não representa a PAR dentro da função `main`; o fato delas terem o mesmo nome é indiferente.

Exemplo 3.4 Calcula soma dos números pares entre 0 e 200

Código fonte `/f_soma.c`[code/cap3/f_soma.c]

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

float f_soma(int); // protótipo da função f_soma
int main() {
    float soma; // declara uma variável do tipo float
    int PAR; // declara uma variável do tipo int
    soma=0; // inicializa a variável soma em 0 (zero)
    PAR = 200; // inicializa a variável PAR em 200
    soma = f_soma(PAR);
    printf ("A soma dos números pares em 0 e 200 é %0.1f ", soma);
    system("pause");
    return 0;
}
// Corpo da função f_soma. Soma os números pares entre 200 e zero
float f_soma(int PAR) {
    float s=0; // declara e inicializa em 0 uma variável do tipo float
    while (PAR!=0) {
        s = s + PAR;
        PAR= PAR - 2;
    }
    return s;
}
```

Dica

Como exercício para casa e para visualizar o que foi dito, inclua, usando o comando `printf`, a impressão da variável PAR dentro da função `main` e após o `while` dentro da função `f_soma`. Veja que dentro da função `main` será impresso o valor 200 e após o `while` da função `f_soma` será impresso o valor 0 (zero).

Continuando o debate, será mostrada agora a *pilha de execução* na memória do programa em questão. Será utilizada a notação descrita na Figura 3.4 [56]. Resumidamente, a pilha de execução funciona da seguinte forma. Cada variável local de uma função é colocada na pilha de execução. As chamar uma função, os parâmetros da função são copiados para a pilha e tratados como se fossem variáveis locais da função chamada. Quando a execução da função é concluída, a parte da pilha correspondente àquela função é liberada. Por isso, não se pode acessar as variáveis locais de fora da função em que foram definidas. Perceba na Figura 3.5 [58](a) que quando o programa inicia, a função `main` inicia com a pilha vazia. Seguindo na execução do programa (linhas 6 a 9 do programa), a Figura 3.5 [58](b) mostra a definição e inicialização das variáveis `soma` e `PAR` na pilha. Com a chamada da função (linha 10), a Figura 3.5 [58](c) apresenta a variável `PAR` da função `f_soma` na pilha de execução (observe que `PAR` inicia com valor 200). A Figura 3.5 [58](d) ilustra a criação/inicialização da variável local `s` dentro da função `f_soma`. A Figura 3.5 [58](e) mostra a situação da pilha após

o laço `while` (linha 22), note que agora `PAR` possui valor 0 (zero) e `s` 10100.0 (que é a soma dos números pares entre 0 e 200). Nesse caso, veja que a variável `PAR` dentro da função `main` continua com seu valor inicial, ou seja, 200. A Figura 3.5 [58](f) mostra a situação após o retorno da função (linha 23), ou seja, o valor retornado pela função `f_soma` é armazenado na variável `soma` dentro da função `main` (linha 10).

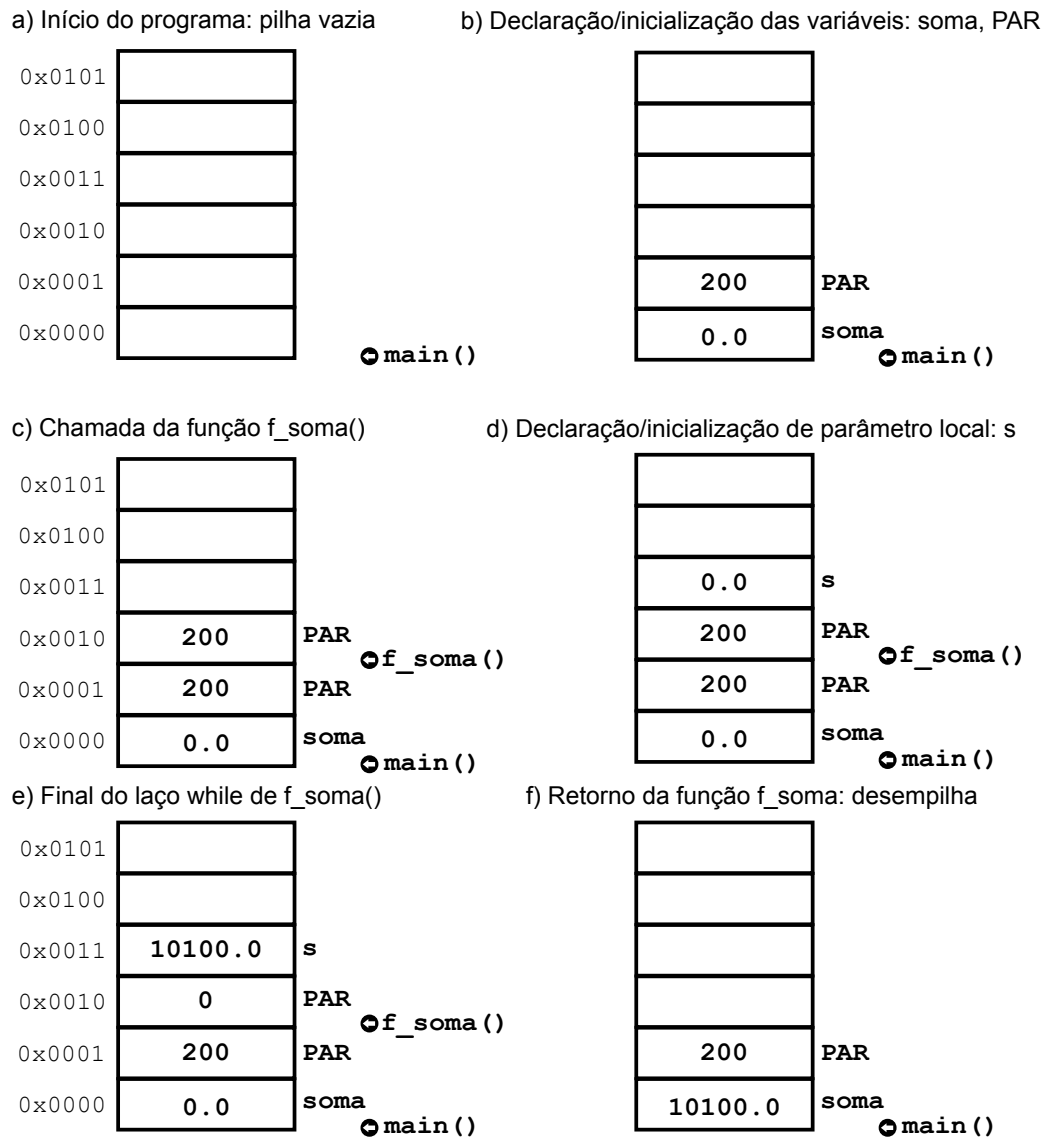


Figura 3.5: Ilustração da pilha de execução na memória RAM de um programa em execução.

3.3.2 Passagem de parâmetros em funções: passagem por valor vs passagem por referência

Na subseção anterior foi vista a pilha de execução de um programa com funções em C e concluiu-se que não se pode modificar valores das variáveis locais pertencentes a uma função fora do escopo de tal função. Esse tipo de passagem de parâmetros que foi vista anteriormente é chamada **passagem por valor**. Nesse caso, uma *cópia* dos parâmetros da função é feita e passada para a função chamada e, portanto, as mudanças nessa *cópia* não afetam o valor original da variável chamadora.

Importante

Todas as chamadas em C são feitas por valor. Cabe ao programador identificar as situações em que se deseja empregar a passagem por referência.

A chamada por valor deverá ser usada sempre que não se desejar/precisar modificar o valor da variável original que foi chamada.

Todavia, muitas funções exigem a necessidade de modificar uma ou mais variáveis na função chamadora ou passar um ponteiro para um objeto com grande quantidade de dados (por exemplo, um array), para evitar a sobrecarga de passar o objeto inteiro por valor, ou seja, isso implicaria na sobrecarga de ter de fazer uma cópia do objeto inteiro na pilha de execução. Uma outra limitação do uso de **passagem por parâmetro** é que nesse tipo de implementação a função só pode retornar **um único valor** para a função chamadora. Quando se deseja retornar mais de um valor, o esquema de passagem por valor não poder ser utilizado. Para resolver esses problemas, usa-se em C² a passagem de **parâmetros por referência** para as funções e é necessário o uso de ponteiros para esse fim.

Para melhor entender esse debate, o Exemplo 3.5 [59] mostra um programa para se calcular a soma e subtração de dois números inteiros. Vamos iniciar com um programa que usa uma função para calcular a subtração de dois números inteiros:

Exemplo 3.5 Calcular a soma e subtração de dois números inteiros

Código fonte /subtracao-passagem-por-valor.c[code/cap3/subtracao-passagem-por-valor.c]

```
/* programa que usa uma função para calcular a
subtração de dois números inteiros*/

#include <stdio.h>
#include <stdlib.h>

int calculadora(int, int); // protótipo da função calculadora

int main() {
    int sub=0; // declara/inicializa uma variável do tipo int
    sub = calculadora(10, 7); //❶
    printf ("A subtração entre 10 e 7 é %d.\n", sub);
    //system("pause");
    return EXIT_SUCCESS;
}

// Corpo da função calculadora
int calculadora(int x, int y){
    return (x - y); // retorno explícito
}
```

- ❶ Salvando na variável `sub` o retorno da função `calculadora`.
-

Nesse exemplo não há um problema explícito, pois o resultado da subtração é retornado pela função `calculadora` explicitamente e armazenado na variável `sub`. Um dos problemas da **passagem de parâmetros por valor** acontece quando se deseja retornar em uma função mais de um valor para a

²Outras linguagens com Lua e Ruby permitem funções retornar vários valores, que podem ser atribuídos a mais de uma variável.

função chamadora. Vamos supor que agora se deseje retornar o valor da soma e subtração de dois números inteiros. Usando a ideia da passagem de parâmetros por valor uma forma incorreta de se implementar esse programa é mostrada no Exemplo 3.6 [60].

Exemplo 3.6 Implementação incorreta de passagem de parâmetros

Código fonte /passagem_incorreta.c[code/cap3/passagem_incorreta.c]

```
/* programa que usa uma função para calcular a soma e
 * subtração de dois números inteiros. FORMA INCORRETA*/
#include <stdio.h>
#include <stdlib.h>

void calculadora(int, int, int, int); // protótipo da função

int main() {
    int sub, soma; // declara duas variáveis do tipo int
    calculadora(10, 7, sub, soma);
    printf ("A subtração e soma entre 10 e 7 são %d e %d. \n",
            sub, soma);
    //system("pause");
    return EXIT_SUCCESS;
}

// Corpo da função calculadora
void calculadora(int x, int y, int sub, int soma){
    sub = x - y;
    soma = x + y;
}
```

Saída da execução do programa

A subtração e soma entre 10 e 7 são 134513769 e -1217228812.

Como se sabe, e já foi alertado no texto, *esse código não funciona*. Serão impressos na tela valores incorretos e que dependem da execução em cada computador, os valores das variáveis `sub` e `soma` calculados na função `calculadora` não possuem relação com as variáveis `sub` e `soma` impressos na função chamadora (nesse caso a função `main`) na linha 10.

Para resolver esse problema é preciso usar a **passagem de valores por referência**, ou seja, passar ponteiros para a função. Como já se sabe, com o uso de ponteiros é possível alterar indiretamente valores de variáveis. Dessa forma, se for passado para uma função os valores dos endereços de memória em que suas variáveis estão armazenadas (i.e., os ponteiros que apontam para essas variáveis), essa função pode alterar *indiretamente* os valores das variáveis na função chamadora. Usando-se essa estratégia, é possível resolver o problema do Exemplo 3.6 [60]. Isso é feito no código do Exemplo 3.7 [60].

Exemplo 3.7 Passagem de valores por referência

Código fonte /passagem_por_referencia.c[code/cap3/passagem_por_referencia.c]

```
/* programa que usa uma função para calcular a soma e
subtração de dois números inteiros. FORMA CORRETA*/

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

```
void calculadora(int, int, int *, int *); // ❶

int main() {
    int sub, soma; // declara duas variáveis do tipo int
    calculadora(10, 7, &sub, &soma); // ❷
    printf ("A subtração e soma entre 10 e 7 são %d e %d. \n",
            sub, soma);
    // system("pause");
    return EXIT_SUCCESS;
}

// Corpo da função calculadora
void calculadora(int x, int y, int *ptr1, int *ptr2){ // ❸
    *ptr1 = x - y; // ❹
    *ptr2 = x + y; // ❺
}
```

- ❶ Protótipo da função cabeçalho definindo receber dois números inteiros e dois ponteiros como para inteiros como parâmetros.
- ❷ Chamada da função calculadora, similar ao programa anterior, mas agora passa o endereço das variáveis sub e soma.
- ❸ Implementação da função com os ponteiros, similar ao cabeçalho; ptr1 recebe o endereço de memória de sub e ptr2 o de soma.
- ❹, ❺ Atualização das variáveis utilizando os ponteiros e o operador de indireção *.

Saída da execução do programa

A subtração e soma entre 10 e 7 são 3 e 17.

O **protótipo da função** calculadora é definido com duas variáveis do tipo inteiro e dois ponteiros para inteiros como parâmetros de entrada. Compare com o exemplo anterior (forma incorreta).

A função calculadora continua sendo uma função sem retorno (como no código anterior). Entretanto, são também passados como parâmetros de entrada os endereços de memória das variáveis sub e soma (i.e., &sub e &soma). Compare esse trecho do código com o exemplo anterior (forma incorreta).

A implementação da função calculadora utiliza dois inteiros (x e y) e dois ponteiros para inteiros como parâmetros de entrada (i.e., ptr1 e ptr2), conforme foi definido no protótipo da função. Nesse cenário, quando a função calculadora é chamada, os endereços de memória das variáveis sub e soma (i.e., &sub e &soma) são armazenados nos ponteiros ptr1 e ptr2. A partir desse momento, é possível atualizar o conteúdo das variáveis sub e soma usando ptr1 e ptr2, utilizando o operador de indireção * (Seção 3.2 [52]).

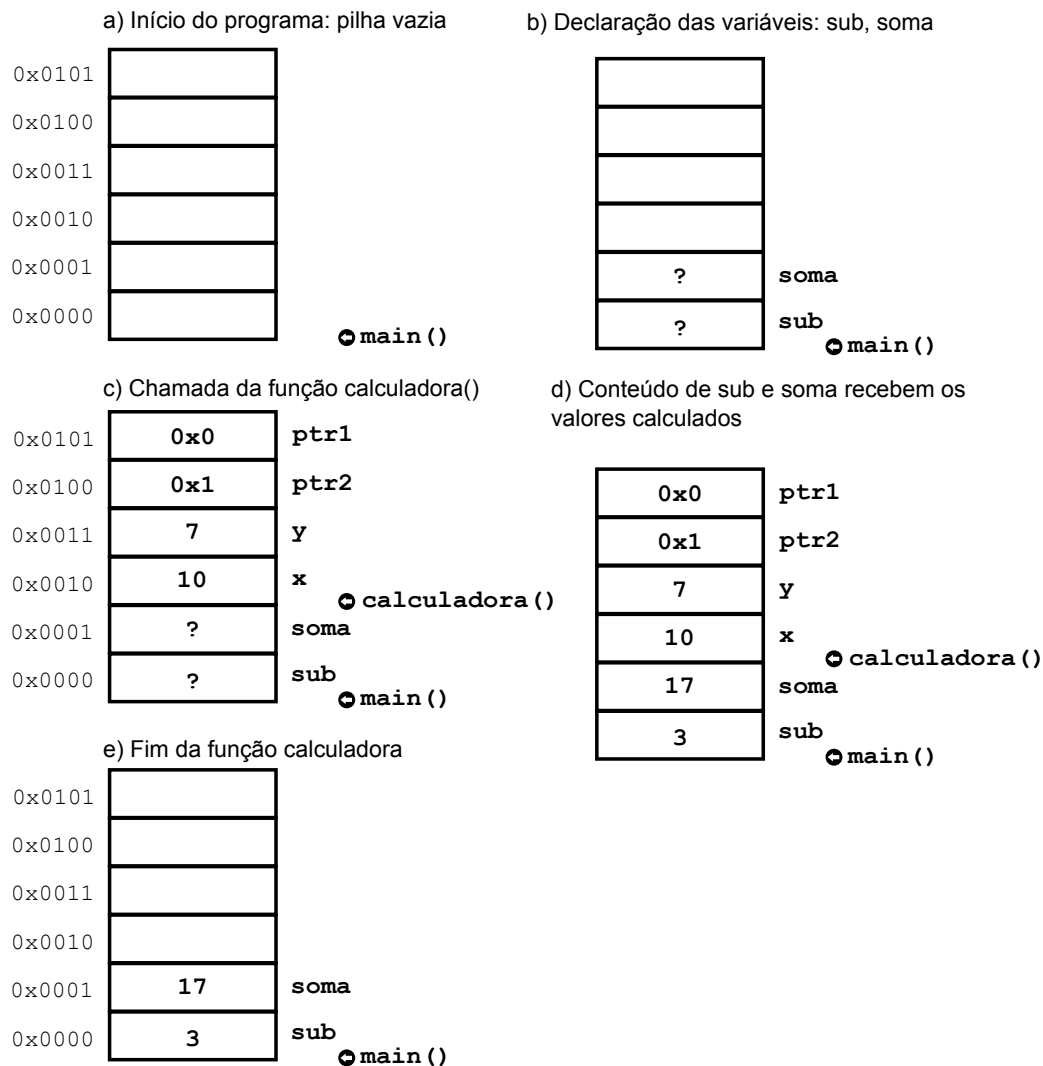


Figura 3.6: Ilustração da execução do programa para o cálculo da subtração e soma de dois números inteiros.

A Figura 3.6 [62] ilustra a pilha de execução da forma correta do programa usando ponteiros. Note que na Figura 3.6 [62](c) que os ponteiros `ptr1` e `ptr2` recebem os endereços de memória das variáveis `sub` e `soma`, respectivamente. Com isso, agora é possível alterar o conteúdo de `sub` e `soma` usando esses ponteiros, mesmo dentro de uma função em que `sub` e `soma` não são definidas. Isso é mostrado na Figura 3.6 [62](d), observe que os ponteiros são utilizados para se computar a subtração e soma. A Figura 3.6 [62](e) apresenta a situação da pilha após a função `calculadora` ser executada. Veja que as variáveis locais `x`, `y`, `*ptr1` e `*ptr2` são desempilhadas.

3.4 Atividades

1. Defina ponteiros e explique usando exemplos a função dos operadores de indireção `*` e de endereço `&`.
2. Considere o código a seguir. Quais valores serão impressos na tela?

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

```
int main ()
{
    int tot = 20;
    int *ptr;
    ptr = &tot;
    printf (" %d.", tot);
    printf (" %d", *ptr);
    *ptr = *ptr + tot;
    printf (" %d", tot);
    *ptr = tot + 16;
    printf (" %d\n", tot);
    //system ("pause");
    return EXIT_SUCCESS;
}
```

3. Escreva os comandos em linguagem C que realizam as tarefas dadas nos itens (b), (c), (d) e (e). Considere que a variável V1 é do tipo double. O item (a) está resolvido para que o aluno melhor entenda o que se pede na questão.

- a. Defina um ponteiro para double;
- b. Atribua o endereço da variável V1 ao ponteiro para double definido na letra (a);
- c. Imprima o endereço de memória da variável V1;
- d. Imprima o endereço de memória armazenado no ponteiro da letra (b);
- e. Inicialize a variável V1 em 12,5 e depois modifique o seu valor para 30,4 usando o ponteiro da letra (b).

(a) Solução:
double *ptr;

4. Faça um programa em C para calcular a área de um retângulo. O programa deve solicitar ao usuário os dois lados do retângulo. Você deve fazer duas versões desse programa:

- a. usando uma função com passagem por valor para calcular a área do retângulo;
- b. usando uma função com passagem por referência para calcular a área do retângulo.

Para cada caso faça/desenhe a pilha de execução do programa.

5. Elabore um programa em C que receba três valores (obrigatoriamente maiores do que zero), representando as medidas dos três lados de um triângulo. Além da função main, o programa deve conter outra função para: i) determinar se esses lados formam um triângulo; ii) determinar e mostrar o tipo de triângulo (equilátero, isósceles ou escaleno), caso as medidas formem um triângulo. Na solução deve-se usar passagem de parâmetros por referência.

Dica



Sabe-se que para ser triângulo, a medida de um lado qualquer deve ser inferior ou igual à soma das medidas dos outros dois lados. Triângulo equilátero possui todos os lados iguais; triângulo isósceles possui pelo menos dois lados de medidas iguais; triângulo escaleno possui as medidas dos três lados diferentes.

6. Considere uma equação de movimento para calcular a posição (s) e velocidade (v) de uma partícula em um determinado instante t, dado sua aceleração a, posição s0 e velocidade v0. O programa abaixo ilustra o uso dessa equação, imprimindo os valores 55.5 e 25.0, que representam a posição e velocidade calculadas para os valores utilizados pelo programa.

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
    float s0 = 3.0, v0 = 10.0, a = 5.0, t = 3.0;
    float s, v;
    movimento (s0, v0, a, t, &s, &v);
    printf ("%f %f\n", s, v);
    //system ("pause");
    return EXIT_SUCCESS;
}
```

Implemente a função movimento de tal forma que o exemplo acima funcione adequadamente. A solução deve funcionar para quaisquer valores de t, a, s0 e v0, não podendo ser particularizada para o programa acima. Para o cálculo da velocidade (v) use a equação $v=v_0 + a.t$ e da posição (s) use $s= s_0 + v_0.t + (a.t^2)/2$.

7. Considere o programa abaixo que possui uma função calculadora que serve para calcular soma (s) e o produto (p) de dois números reais (c e d). O programa abaixo ilustra o uso dessa função para quando os valores 2.5 e 3.9 são utilizados pelo programa. Implemente a função calculadora de tal forma que o código abaixo funcione adequadamente. A solução deve funcionar para quaisquer valores de c e d, não podendo ser particularizada para os valores usados no programa abaixo, i.e., $c= 2.5$ e $d=3.9$. Após montar o código, faça/desenhe a pilha de execução do programa para o caso quando $c= 2.5$ e $d=3.9$.

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
    float c = 2.5, d = 3.9, s, p;
    calculadora (c, d, &s, &p);
    printf ("%f %f\n", s, p);
    //system ("pause");
    return EXIT_SUCCESS;
}
```

8. Implemente uma função chamada calc_esfera, que calcule o volume e a área da superfície de uma esfera de raio r. Essa função deve obedecer ao protótipo:

```
void calc_esfera(float r, float *area, float *volume).
```

Em seguida, monte um programa que peça o raio da esfera ao usuário e calcule os valores da área e volume da esfera. Dica: a área e volume da esfera são calculados como $area=4.\pi.r^2$, $volume=(4.\pi.r^3)/3$.



Feedback sobre o capítulo

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

Para compreender melhor como feedbacks funcionam consulte o guia do curso.

Capítulo 4

Alocação Dinâmica de Memória

OBJETIVOS DO CAPÍTULO

Ao final deste capítulo você deverá ser capaz de:

- Fazer alocação dinâmica de memória em programas em linguagem C e entender as suas vantagens com relação à alocação estática de memória;

Até agora foi visto que ponteiros podem ser usados para guardar endereços de variáveis já existentes em um programa em C. As variáveis podem ser entendidas como um espaço de memória que é rotulado durante o processo de compilação do programa em C. Ou seja, é alocado um espaço de memória **estático** para as variáveis. Nesse caso, foi visto que os ponteiros fornecem apenas uma segunda forma de acesso às variáveis, pois recebem os endereços de memória onde essas variáveis foram criadas.

Um ponto importante a ser notado é que *durante a execução do programa em C*, uma vez criadas, essas variáveis não podem ser mais apagadas da memória. Ou seja, os espaços de memória utilizados pelas variáveis, não podem ser liberados para uso por outras variáveis ou tipos de dados durante a execução desse programa. Esse tipo de alocação de memória é chamada **alocação estática**. Ou seja, esse espaço de memória alocado estaticamente fica reservado para a variável (ou outro tipo/estrutura de dados, como arrays, estruturas ou registros, dentre outros...) durante todo o tempo de execução do programa.

Outro ponto importante é que na alocação estática o programador deve saber a quantidade de memória que é preciso ser definida no programa. Todavia, há casos em que o programador não sabe, no momento em que está montando o programa, a quantidade de memória (ou de tipo/estrutura de dados) que o programa irá precisar. Ou seja, há casos em que a quantidade de memória necessária varia de acordo com a execução do próprio programa. Um exemplo é um programa que permite múltiplos documentos abertos ao mesmo tempo, como muitos processadores de texto. Nesse caso, o programa não estima quantos documentos o usuário irá abrir nem qual o tamanho de cada documento aberto. Faz-se então necessário estudar um mecanismo que permita a alocação de memória de acordo com a necessidade e durante a execução do próprio programa. Isso é o que se chama de **alocação dinâmica** de memória.

Nesse cenário, os ponteiros são nossos aliados e o seu verdadeiro poder está em apontar para a **memória não rotulada ou dinâmica**, alocada durante a execução do programa, i.e., alocada dinamicamente. Para melhor entender o conceito de alocação dinâmica, são discutidas três formas para reservar espaço de memória para o armazenamento de informações em um programa em C:

Uso de variáveis globais e estáticas

Sabe-se que o espaço reservado de memória para a variável global existe enquanto o programa estiver em execução e só pode ser utilizado pela própria variável.

Uso de variáveis locais em funções

Como já foi visto, o espaço de memória reservado para a variável só existe enquanto a função que a declarou está sendo executada. Quando a execução da função for concluída, esse espaço de memória está liberado para outros usos e, portanto, as informações armazenadas na variável são perdidas e não podem ser mais recuperadas no escopo do programa.

Uso de alocação dinâmica

A terceira opção é requisitar ao sistema, durante a execução do programa, um espaço de memória de tamanho determinado. Após concedido, esse espaço alocado dinamicamente permanece reservado até que seja liberado pelo programa, usando uma função específica que será visto adiante. Com isso, pode-se alocar dinamicamente um espaço de memória em uma função e acessá-lo em outra. Quando o espaço de memória é liberado, ele fica disponível para alocações futuras, inclusive para outros processos, não devendo ser acessado novamente após sua liberação. Como já foi discutida, uma vantagem dessa abordagem é que, em problemas nos quais o programador *não sabe* a dimensão ou espaço necessário dos tipos de dados que irá utilizar no programa, pode-se usar a alocação dinâmica de memória para alocar e liberar memória ao longo da execução de um programa.

A Figura 4.1 [67] ilustra um esquema didático do uso da memória RAM do computador pelo sistema operacional durante a execução de um programa. Quando um programa é executado, o seu código em linguagem de máquina é carregado na memória RAM. Há espaços reservados para gravação de variáveis globais e estáticas que possam existir nesse programa. O restante da memória livre pode ser utilizado pelas variáveis locais e pelas variáveis alocadas dinamicamente. Toda vez que uma função for chamada, é reservado um espaço na pilha de execução para o armazenamento das variáveis locais da função. Lembre-se que, conforme debatido anteriormente, esse espaço é liberado quando a função acaba a sua execução. A parte da memória não utilizada pela pilha de execução pode ser usada para alocar memória dinamicamente. Caso a pilha de execução cresça além do espaço de memória livre ou a quantidade de memória a ser alocada dinamicamente seja maior que a memória livre, o programa para de funcionar. Mais adiante, será visto como se deve fazer para abordar o programa na situação de alocação de memória além do espaço disponível.

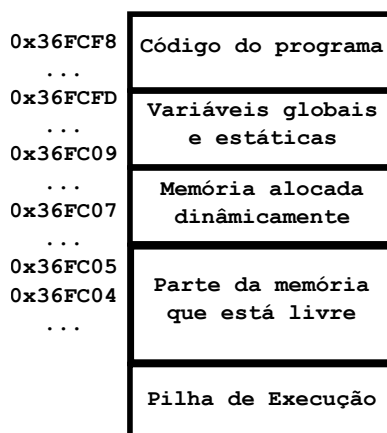


Figura 4.1: Ilustração do uso da memória RAM do computador durante a execução de um programa.

4.1 Alocação dinâmica de memória

A alocação dinâmica de memória é feita usando-se funções da biblioteca padrão `stdlib.h` [170]. As funções básicas para alocar e liberar memória são a **malloc** e **free**, respectivamente. Como será visto adiante, é possível fazer também realocação dinâmica de memória usando o **realloc**, ou seja, dado que existe um espaço de memória alocado dinamicamente é possível realocar esse espaço de memória liberando parte do espaço de memória que não vai se utilizar no programa.

A função `malloc()` [170] recebe como parâmetro de entrada o número de bytes que se deseja alocar e retorna o endereço de memória onde inicia o bloco de memória alocada dinamicamente. O leitor já deve ter percebido que para se armazenar o endereço de memória retornado pela função `malloc` deve-se usar uma variável do tipo ponteiro. Dessa forma, o esquema de alocação de memória em C é uma relação entre o uso da função `malloc` e de ponteiros, além da função `free` como será visto adiante.

Como exemplo, considere o caso que deseja alocar dinamicamente um vetor com 20 elementos do tipo `float`. Nesse caso, os comandos em linguagem C seriam

```
float *ptr;  
ptr = (float*) malloc (20*sizeof(float));
```

Em que `ptr` é um ponteiro que irá armazenar o valor retornado pela função `malloc`, note que é um ponteiro para um tipo `float` já que se trata de alocação de um vetor de tipos `float`. Conforme foi visto no Capítulo 1, o comando `sizeof` é utilizado para se computar a quantidade de bytes que um tipo de dado necessita para ser gravado na memória. Nesse caso, será alocado dinamicamente um espaço de memória equivalente a 20 variáveis do tipo `float`. Um último detalhe é que a parte `(float*)` é necessária porque a função `malloc` retorna ou não um ponteiro genérico para um tipo qualquer, sendo representado por `void*`. O `(float*)` é usado para transforma o ponteiro em ponteiro para `float`, visto que está se alocando memória para um tipo `float`. Algumas bibliotecas mais recentes de C não necessitam desse comando, fazendo a conversão do ponteiro automaticamente.

A Figura 4.2 [69] ilustra como é que ocorre na memória a execução desse trecho de código. Observe na Figura 4.2 [69](a) que após a execução do comando que cria o ponteiro, um espaço é alocado na pilha de execução para `ptr`. Na Figura 4.2 [69](b) , considerando que um `float` ocupa 8 bytes, é ilustrada a situação da memória após o comando `ptr = (float*) malloc (20*sizeof(float))`. Perceba também que 160 bytes são alocados dinamicamente na memória e que `ptr` recebe o endereço de memória onde inicia o bloco de 160 bytes recém alocados, ou seja, `*ptr=0x36FC07`.

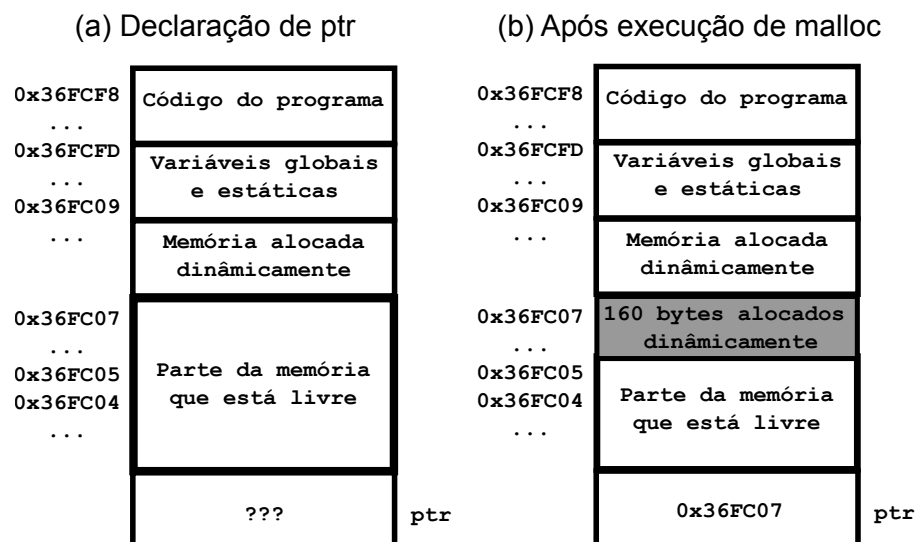


Figura 4.2: Ilustração de alocação dinâmica de um vetor com 20 elementos do tipo float.

Pode acontecer a situação que não haja espaço suficiente de memória para a alocação que se deseja fazer. Nesse caso, a função `malloc()` [170] retorna um endereço nulo para o ponteiro, ou seja, é retornado `NULL`. Um código simples pode ser adicionado no programa para verificar se essa situação ocorre e abortar a execução do programa. Veja no exemplo a seguir que a função `exit()` [170], a qual pertence à biblioteca `stdlib.h` [170], é utilizada para abortar o programa.

```
...
float *ptr;
ptr = (float*) malloc(20*sizeof(float));
if(ptr==NULL) {
    printf ("Não há memória suficiente, o programa será encerrado.");
    exit(EXIT_FAILURE); // aborta o programa retornando erro
}
...
```

Para liberar o espaço de memória alocado dinamicamente a função `free()` [170] é utilizada. Essa função recebe como parâmetro de entrada o ponteiro que indica o início do bloco de memória a ser liberada. No exemplo em questão, seria o ponteiro `ptr`. Dessa forma, para liberar o espaço de memória do vetor de 20 elementos `float` usa-se o comando:

```
free(ptr);
```

Vale a pena mencionar que o espaço de memória que foi liberado, não pode ser mais acessado pelo programa. Porém, pode ser reutilizado para armazenar outras informações, inclusive para se fazer uma nova alocação dinâmica de memória. Outro ponto importante é que o sistema operacional do computador é quem faz o trabalho de alocação e gerenciamento de memória. O programador apenas informa por meio dos comandos no programa que deseja fazer uma alocação de memória.

Dica



A função `free()` [170] não destrói o ponteiro que recebe como entrada. Portanto, esse ponteiro pode ser reaproveitado dentro do escopo do mesmo programa. A função `free()` apenas libera o espaço de memória que foi alocado dinamicamente no programa para que possa ser utilizado para armazenar outros dados.

Atenção

Não se pode liberar o mesmo bloco de memória duas vezes. Ou seja, basta utiliza uma vez a função `free()` para cada bloco de memória.



```
float *ptr;
ptr = (float*) malloc (20*sizeof(float));
. . .
free(ptr);
free(ptr); // comando incorreto
```

Também não se pode usar `free()` para liberar memória criada com a declaração de variáveis.

```
int soma;
soma=80;
. . .
free(soma); // comando incorreto
```

O Exemplo 4.1 [70] mostra que um ponteiro pode ser utilizado várias vezes para se alocar memória dinamicamente. O cuidado que o programador deve ter é liberar o espaço de memória alocado dinamicamente antes de fazer uma nova alocação usando o mesmo ponteiro. Outra observação importante é que quando esse código foi executado no computador do autor, o mesmo endereço de memória foi impresso nas linhas 15 e 23. Ou seja, após a liberação do espaço de memória alocado dinamicamente (linha 16), o sistema operacional pode reutilizá-lo para outros fins, que nesse caso foi uma nova alocação dinâmica de memória (linha 19).

Exemplo 4.1 Reutilizando um ponteiro

Código fonte /reutilizando_ponteiro.c[code/cap4/reutilizando_ponteiro.c]

```
/* programa que usa o mesmo ponteiro para alocar memória
dinamicamente em duas situações diferentes*/

#include <stdio.h>
#include <stdlib.h>
int main() {
    int *ptr; // declara um ponteiro para um inteiro
    ptr = (int*) malloc(sizeof(int)); // aloca memória para inteiro
    if(ptr==NULL) {
        printf ("Não há memória suficiente para alocação.");
        exit(EXIT_FAILURE);
    }
    *ptr=69; // coloca um valor lá
    printf("Valor inteiro = %d \n", *ptr); // imprimi valor
    printf("Localização na memória= %p \n\n", ptr); // imprimi endereço
    free(ptr); // o espaço de memória é liberado

    // usa o mesmo ponteiro para alocar memória para outro inteiro
    ptr = (int*) malloc(sizeof(int));

    *ptr=45;
    printf("Valor inteiro = %d \n", *ptr); // imprimi valor
    printf("Localização na memoria= %p \n\n", ptr); // imprimi endereço
```

```
free(ptr); // o espaço de memória é liberado novamente

// system("pause");
return EXIT_SUCCESS;
}
```

Saída da execução

Valor inteiro = 69
Localização na memória= 0x9929008

Valor inteiro = 45
Localização na memória= 0x9929008

Por fim, é possível fazer uma realocação de um espaço de memória que foi alocado dinamicamente. Isso é feito usando a função `realloc()` [171]. O protótipo da função é:

```
void *realloc (void *ptr, tamanho do espaço a ser realocado);
```

No exemplo a seguir é feita uma realocação em que o espaço alocado passa de 20 para 25 tipos `float`. E depois de 25 para 12 tipos `floats`.

Exemplo 4.2 Realocação de espaço

```
float *ptr = (float* ) malloc (20*sizeof(float));
// ... (sequência de comandos)
ptr = (float* ) realloc (ptr, 25*sizeof(float));
// ... (sequência de comandos)
ptr = (float* ) realloc (ptr, 12*sizeof(float));
```

4.2 Arrays dinâmicos

4.2.1 Vetores dinâmicos

Na maioria das vezes reserva-se o uso de alocação dinâmica para os casos em que a dimensão do vetor é desconhecida.

Dica



Quando se sabe a dimensão de um vetor, é preferível usar vetores alocados estaticamente, visto que do ponto de vista de tempo de execução do programa, esse tipo de alocação de vetores é mais rápida. Um ponto importante é que caso o vetor seja definido dentro do escopo de uma função, então ele somente existirá enquanto a função estiver sendo executada. Portanto, o programador deve atentar para a utilização de vetores dentro de funções.

Para alocar dinamicamente um vetor, pode-se usar o seguinte comando:

```
float *ptr;
ptr = (float*) malloc(n*sizeof(float));
```

Em que `ptr` é um ponteiro que irá armazenar o valor retornado pela função `malloc`, note que `ptr` é um ponteiro para um tipo `float` já que se trata de alocação de um vetor de tipos `float`, e `n` representa a dimensão do vetor. O Exemplo 4.3 [72] mostra como se utilizar um vetor dinâmico que calcula a soma de dois vetores.

Exemplo 4.3 Programa que realiza soma de dois vetores

Código fonte `/soma_vetores.c`[code/cap4/soma_vetores.c]

```
/* programa que soma dois vetores usando alocação dinâmica*/
#include <stdio.h>
#include <stdlib.h>
int* soma_vet(int[], int[]);
int main() {
    int i, *ptr1; // declara um inteiro e um ponteiro para inteiro
    int va[8], vb[8]; // declara dois vetores 8 posições
    for(i=0; i<8; i++) { // preenchendo os vetores va e vb
        va[i]=i;
        vb[i]=i+1;
    }
    ptr1=soma_vet(va, vb); // chamando a função soma_vet
    for(i=0; i<8; i++) { // imprimindo o vetor va
        printf("%d\t", va[i]);
    }
    printf("\n");
    for(i=0; i<8; i++) { // imprimindo o vetor vb
        printf("%d\t", vb[i]);
    }
    printf("\n");
    for(i=0; i<8; i++) { // imprimindo o vetor soma
        printf("%d\t", ptr1[i]);
    }
    printf("\n");
    //system("pause");
    return EXIT_SUCCESS;
}
int* soma_vet(int va[8], int vb[8]) {
    int *ptr2;
    ptr2 = (int*) malloc(8*sizeof(int));
    if(ptr2==NULL) {
        printf ("Não há memória suficiente. O programa será encerrado.");
        exit(EXIT_FAILURE);
    }
    for(int j=0; j<8; j++) {
        ptr2[j]= va[j] + vb[j];
    }
    return ptr2;
}
```

Saída da execução

0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
1	3	5	7	9	11	13	15

Note que há outras maneiras de se resolver o problema da soma de dois vetores. Esse exemplo foi dado somente para ilustrar o uso de alocação de um vetor dinamicamente. Veja que a alocação dinâmica de memória ocorreu no escopo da função `soma_vet` (linha 30). Como o vetor é de oito posições, foi alocado um espaço para oito números inteiros no comando da linha 30. O ponteiro `ptr2` foi utilizado como marcador inicial do vetor para guardar a soma dos vetores `va` e `vb`. Um último detalhe é que o comando `free(ptr2)` não necessitou ser dado, já que após o comando de `return` na linha 38 o bloco de memória alocado será liberado automaticamente já que a função foi encerrada.

4.2.2 Matrizes dinâmicas

Para se definir matrizes dinâmicas na linguagem C enfrenta-se a limitação de que na linguagem C só é permitida fazer alocação dinâmica de memória de estruturas unidimensionais, como é o caso de vetores. Como uma matriz é uma estrutura com duas dimensões (linhas e colunas), para fazer uma alocação dinâmica de uma matriz é preciso utilizar artifícios de programação utilizando vetores.

Por exemplo, partindo da ideia de que para se alocar uma matriz na memória é preciso ter espaço suficiente para seus elementos, pode-se utilizar um vetor para tal fim. O tamanho do vetor alocado dinamicamente seria determinado de acordo com as dimensões da matriz que se deseja alocar. A ideia, portanto, é transformar, do ponto de vista conceitual, a matriz em um vetor unidimensional. A seguir é apresentada uma maneira de fazer tal transformação.

Considere uma matriz com `l` linhas e `c` colunas, a qual pode ser representada na linguagem C como `mt_r[l][c]`. É possível criar um vetor com `l*c` elementos, aqui chamado de `vet[l*c]`, que representará a matriz `mt_r[l][c]`. Para a correspondência entre a matriz e o vetor seja atendida, um elemento a_{ij} da matriz é mapeado no elemento $k=i*c+j$, em que `c` é o número de colunas da matriz, como foi definido no início desse parágrafo. Essa relação é utilizada para se encontrar um elemento `k` do vetor `vet[l*c]`, ou seja, o elemento `vet[i*c+j]`. A Figura 4.3 [73] mostra uma ilustração desse mapeamento de matriz em vetor.

O ponto negativo dessa estratégia é que é preciso usar a notação `vet[i*c + j]` para acessar os elementos da matriz no vetor.

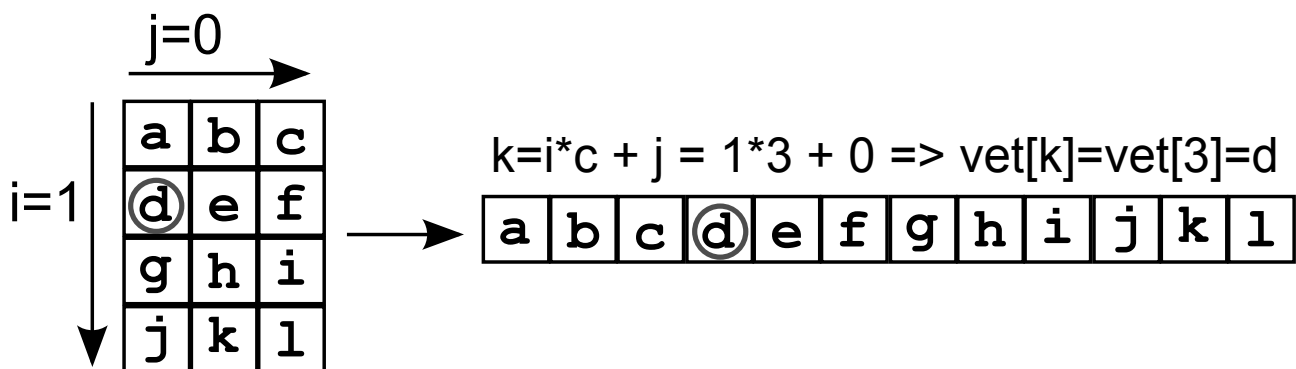


Figura 4.3: Ilustração do mapeamento de uma matriz em um vetor.

Usando esse mapeamento, a alocação dinâmica de uma matriz recai no problema de alocação dinâmica de um vetor. De forma geral, se quisermos fazer a alocação dinâmica de uma matriz com `l` linha e `c` colunas para armazenar números reais, basta fazer como se segue:

```
// ponteiro para guardar o endereço onde inicia a alocação
float *mt_r;
```



```
// note que foram alocados l*c elementos
mtr = (float*) malloc(l*c*sizeof(float));
```

O Exemplo 4.4 [74] mostra o uso dessa estratégia para uma matriz com 4 linhas e 3 colunas.

Exemplo 4.4 Utilização de vetor para alocação de uma matriz

Código fonte /vetor_matriz.c[[code/cap4/vetor_matriz.c](#)]

```
/* programa que usa um vetor para alocar dinamicamente uma matriz */
#include <stdio.h>
#include <stdlib.h>
int main() {
    int l=4, c=3; // declara as linhas e colunas da matriz

    float *mtr; // declara dois floats e um ponteiro para float
    mtr = (float*) malloc(c*l*sizeof(float)); // alocação dinâmica
    if(mtr==NULL) {
        printf ("Memoria insuficiente para alocar os c*l elementos.");
        exit(EXIT_FAILURE);
    }
    // preenchendo o vetor (diretamente) e a matriz (indiretamente)
    for(int i=0; i<l; i++) {
        for(int j=0; j<c; j++) {
            mtr[i*c + j]= (9*i+j)/4.0;
        }
    }
    // imprimindo a matriz
    for (int i=0; i<l; i++) {
        for (int j=0; j<c; j++) {
            printf ("%0.1f\t", mtr[i*c + j]);
        }
        printf ("\n");
    }
    printf("\n");
    free(mtr); // libera do espaço de memória alocado dinamicamente
    //system("pause");
    return EXIT_SUCCESS;
}
```

Saída da execução

```
0.0 0.2 0.5
2.2 2.5 2.8
4.5 4.8 5.0
6.8 7.0 7.2
```

Como comentário final, existem outras maneiras de se alocar dinamicamente uma matriz, todavia, em geral, é preciso utilizar um vetor como estrutura auxiliar para representar a matriz.

4.2.3 Registros (estruturas) dinâmicas

É possível também alocar dinamicamente um registro ou estrutura na linguagem C. Considere um registro definido como a seguir:

Tipo estrutura jogador

```
typedef struct jogador_ {  
    char nome[40];  
    float salario;  
    unsigned gols;  
} jogador;
```

A alocação de registro dinamicamente segue a mesma lógica que já foi vista. Ou seja, deve-se inicialmente definir um ponteiro para o tipo de estrutura/tipo que se deseja alocar e em seguida usa-se a função `malloc` para fazer a alocação. Seguindo esse raciocínio, tem-se para o caso de um registro `jogador`:

```
jogador *ptr;  
ptr = (jogador*) malloc(sizeof(jogador));
```

Nesses dois comandos é alocado dinamicamente um registro do tipo `jogador` que é capaz de armazenar o nome do jogador com até 40 caracteres, o salário e o números de gols. O espaço de memória reservado é igual à soma do espaço dos campos pertencentes ao registro `jogador`. O comando `sizeof(jogador)`, por sua vez, automaticamente passa para a função `malloc` a quantidade de bytes necessários para alocar a estrutura `jogador`. Note que o programador não precisa se preocupar em computar a quantidade de memória necessária, como foi dito, isso é feito automaticamente pelo comando `sizeof(jogador)`.

Um ponto importante a ser observado é que o operador para acessar os campos de um registro alocado dinamicamente é o `->` e não o operador ponto (`.`). O Exemplo 4.5 [75] ilustra a alocação dinâmica de registro e o preenchimento de seus campos. Perceba a alocação dinâmica do registro é feita na linha 14, o preenchimento de seus campos é feitos nas linhas 16 a 18 e na linhas 20 os valores gravados são impressos. Note que o operador `->` foi utilizado para acessar os campos do registro, tanto no preenchimento quanto na impressão. Na linha 21 a memória alocada é liberada usando o comando `free(ptr)`.

Exemplo 4.5 Programa que demonstra alocação de registro dinamicamente

Código fonte `/aloca_registro_dinamicamente.c`[[code/cap4/aloca_registro_dinamicamente.c](#)]

```
/* programa que aloca dinamicamente um registro */  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
typedef struct jogador_ {  
    char nome[40];  
    float salario;  
    unsigned gols;  
} jogador;  
  
int main() {  
    jogador *ptr; // define um ponteiro para o registro 'jogador'  
    // aloca dinamicamente um registro do tipo jogador  
    ptr = malloc(sizeof(jogador));  
    // preenchendo o registro  
    strcpy(ptr->nome, "Tulipa Goleador");  
    ptr->salario = 3000;  
    ptr->gols=2;  
    // imprimindo o registro
```

```

printf("Contratacao de %s com salario R$ %.1f e %u gols na temporada ←
      .\n", ptr->nome, ptr->salario, ptr->gols);
free(ptr);
//system("pause");
return EXIT_SUCCESS;
}

```

Saída da execução

Contratacao de Tulipa Goleador com salario R\$ 3000.0 e 2 gols na ←
temporada.

É possível também definir vetores de registros, i.e., vetores cujos elementos são registros. Em outras palavras, em cada posição desse vetor é gravado um registro inteiro. A Figura 4.4 [76] mostra um vetor de registro para o caso da estrutura `jogador` definida no início dessa seção. Note que, como destacado na figura, em cada posição do vetor tem-se armazenado os campos definidos na estrutura `jogador`, ou seja, os campos `nome`, `salario` e `gols`.

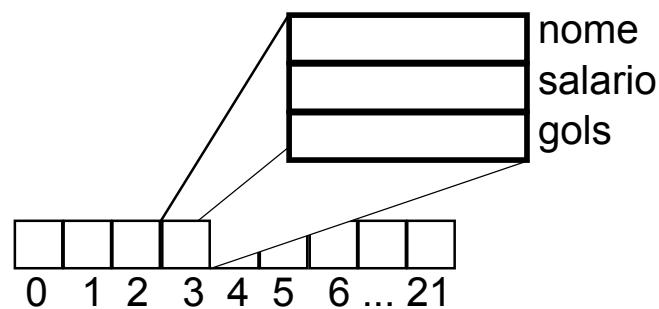


Figura 4.4: Ilustração de um vetor de estruturas.

Primeiramente, será mostrado como definir um vetor de estruturas para, em seguida apresentar como é feita a alocação dinâmica de um vetor de estruturas. Considerando a estrutura `jogador`, definida no início da seção, um vetor de estruturas pode ser definido da seguinte forma:

```
jogador jogadores[22];
```

Em que, `struct` é a palavra obrigatória na sintaxe do comando para especificar que trata-se de um registro ou estrutura, `jogador` é o tipo de registro que se pretende criar o vetor e `22` é o nome do vetor que acabou de ser criado. Observe que foi criado um vetor de estruturas com 22 posições.

O Exemplo 4.1 [70] mostra um código para preenchimento de um vetor de estruturas com 22 posições ilustrando o cadastramento de uma equipe de futebol com 22 jogadores. Note nas linhas 17 a 19, 22 e 23 que o operador ponto “.” é utilizado para acessar os campos do registro.

Exemplo 4.6 Utilização de registros estáticos

Código fonte /registros_estaticos.c[code/cap4/registros_estaticos.c]

```

/* programa que preenche um vetor de registros */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct jogador_ {

```

```
char nome[40];
float salario;
unsigned gols;
} jogador;

int main() {
    int QUANTIDADE_DE_JOGADORES = 22;
    jogador jogadores[QUANTIDADE_DE_JOGADORES];
    for (int i=0; i<QUANTIDADE_DE_JOGADORES; i++) {
        printf("Digite o nome, salario e gols do jogador %d: \n", i+1);
        scanf("%s", &jogadores[i].nome);
        scanf("%f", &jogadores[i].salario);
        scanf("%u", &jogadores[i].gols);
    }
    printf("Jogadores do time: \n");
    for(int i=0; i<QUANTIDADE_DE_JOGADORES; i++) {
        printf("%s ganhando %0.1f \n", jogadores[i].nome,
            jogadores[i].salario);
    }
    //system("pause");
    return EXIT_SUCCESS;
}
```

A alocação de um vetor de registros dinamicamente segue a mesma lógica que já foi vista para um vetor de outros tipos (int, float, double, etc.). Ou seja, deve-se inicialmente definir um ponteiro para o tipo de estrutura/tipo que se deseja alocar e em seguida usa-se a função `malloc` para fazer a alocação dos espaços necessários que iram formar o vetor de registros. Seguindo esse raciocínio, tem-se para o caso de um registro `jogador`:

```
jogador *jogadores;
jogadores = (jogador*) malloc(22*sizeof(jogador));
```

No Exemplo 4.7 [77] temos um código semelhante ao anterior só que nesse caso o vetor de registros foi alocado dinamicamente. Note que na linha 32 o ponteiro `jogadores` é liberado. Um detalhe de implementação é que o operador ponto “.” foi utilizado para acessar os campos do vetor de registros pois a notação `jogadores[i]` é a notação de vetor. A notação de ponteiro é demonstrada na linha 30.

Exemplo 4.7 Utilização de registros alocados dinamicamente

Código fonte /registro_dinamico.c[[code/cap4/registro_dinamico.c](#)]

```
/* Aloca e preenche dinamicamente um vetor de registros */
#include <stdio.h>
#include <stdlib.h>

typedef struct jogador_ {
    char nome[40];
    float salario;
    unsigned gols;
} jogador;

int main() {
    int QUANTIDADE_DE_JOGADORES = 22;
```

```
jogador* jogadores;
jogadores = malloc(QUANTIDADE_DE_JOGADORES * sizeof(jogador));
for (int i=0; i<QUANTIDADE_DE_JOGADORES; i++) {
    printf("Digite o nome, salario e gols do jogador %d: \n", i+1);
    scanf("%s", &jogadores[i].nome);
    scanf("%f", &jogadores[i].salario);
    scanf("%u", &jogadores[i].gols);
}
printf("Time do Treze: \n");
for(int i=0; i<QUANTIDADE_DE_JOGADORES; i++) {
    printf("%s ganhando %0.1f \n", jogadores[i].nome,
        jogadores[i].salario);
}

// Demonstrando notação de ponteiro
jogador* segundoJogador = &jogadores[1];
printf("Segundo jogador: %s, recebe R$ %0.1f \n",
    segundoJogador->nome, segundoJogador->salario);

free(jogadores);
//system("pause");
return EXIT_SUCCESS;
}
```

4.3 Atividades

1. Explique a diferença entre alocação estática e dinâmica de memória, citando vantagens e dando exemplo de uso. Dê pelos menos cinco situações nas quais alocação dinâmica de memória se faz necessária. Na sua resposta, use as Figura 4.1 [67] e Figura 4.2 [69] para ilustrar a utilização da memória do computador em cada um dos esquemas de alocação.
2. Explique, dando exemplos, como funciona as funções `malloc()`, `realloc()` e `free()` são utilizadas em conjunto.
3. A função `calloc()` [170] também costuma ser utilizada na alocação de memória, seu diferencial é que todo o espaço alocado é inicializado com zeros. Para conhecer a utilização desta função, recomendamos que o reescreva o código do Exemplo 4.7 [77] invocando a função `calloc` no lugar de `malloc`.
4. Um professor de uma disciplina deseja montar um programa em linguagem C que seja capaz de receber as médias dos alunos de uma turma. O programa deve gravar as médias em um vetor que é alocado dinamicamente e ao final é liberado. Para dar uma ajuda ao professor, faça um programa que atenda o que foi dito. O programa deve conter uma função ou subrotina que receba o vetor com as médias, o qual foi alocado dinamicamente no programa, e calcular e retornar: i) a média da turma; ii) quantos alunos foram aprovados por média; iii) quantos alunos estão na prova final. Dica: considere que o aluno é aprovado por média se $media \geq 7,0$. Se $3,5 < media < 7,0$, o aluno estará na prova final.
5. Considerando a estrutura:

```
typedef struct ponto_ {  
    int x;  
    int y;  
} Ponto;
```

Para representar um ponto em uma grade 2D. Implemente um programa em C que contenha uma função ou sub-rotina que indique se um ponto p está localizado dentro ou fora de um retângulo. O retângulo é definido por seus vértices inferior esquerdo $v1$ e superior direito $v2$. A função deve retornar 1 caso o ponto esteja localizado dentro do retângulo e 0 caso contrário. Use alocação dinâmica de memória para alocar dinamicamente a estrutura `Ponto`. Essa função deve obedecer ao seguinte protótipo:

```
int dentroRet (Ponto* v1, Ponto* v2, Ponto* p).
```

6. Considerando a estrutura Jogador (Tipo estrutura jogador [75]), implemente um programa em C que preencha e imprima na tela todos os jogadores de todos os times de futebol do Campeonato Brasileiro da Série A. Considere que cada time possui 11 jogadores. Use alocação dinâmica de memória na sua solução.

Dica

Crie uma função ou sub-rotina no programa, algo como `cadastro_equipe`, para cadastrar uma equipe de futebol. Você poderá utilizar essa função para cadastrar as 20 equipes de futebol do Campeonato Brasileiro da Série A. Faça o mesmo para a parte da impressão na tela.

7. Discuta o conceito de registros dinâmicos, matrizes dinâmicas e vetores dinâmicos dando exemplos e fazendo comparações. Usando o método da Seção 4.2.2 [73] para representar matrizes de duas dimensões, é possível representar matrizes com três dimensões? Justifique a sua resposta.

Feedback sobre o capítulo

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

Para compreender melhor como feedbacks funcionam consulte o guia do curso.

Capítulo 5

Arquivos

OBJETIVOS DO CAPÍTULO

Ao final deste capítulo você deverá ser capaz de:

- Entender o conceito de arquivos e seus tipos;
- Conhecer as semelhanças e diferenças entre Arquivos e Fluxos;
- Escrever programas em C que leiam e escrevam corretamente na entrada padrão, na saída padrão e na saída de erros;
- Verificar se houve erro durante as operações de leitura e escrita ou se chegou ao final do arquivo;
- Utilizar as funções `fgetc` ou `getc` para ler caracteres e `fputc` ou `putchar` para escrever caracteres em arquivos;
- Utilizar as funções `fgets` e `fputs` para ler e escrever linhas em arquivos somente texto;
- Utilizar as funções `fread` e `fwrite` para ler e escrever em arquivos binários;
- Compreender o que é o *indicador de posição* de arquivos e identificar as funções que o manipula;

Até agora foi visto como é possível gravar e acessar informações/dados na memória primária/principal do computador (também chamada memória RAM). Para isso, foi visto que o uso de ponteiros é fundamental. O problema dessa abordagem é que as informações/dados de um programa em C gravadas na memória RAM são perdidas após o término da execução desse programa. Ou seja, não se pode mais acessá-las ou recuperá-las. Como fazer para, após a execução de um programa em C, ter acesso a dados/informações manipuladas nesse programa? Uma solução é gravar esses dados/informações em um arquivo na memória secundária do computador (também chamado de Disco Rígido – Hard Disk drive). Nesse caso, após a execução do programa em C, esses dados/informações estarão disponíveis no arquivo gravado no disco rígido da máquina.

Nesse capítulo serão abordados conceitos básicos sobre arquivos e formas de manipulação de dados no disco rígido na linguagem C, como por exemplo, estratégias para salvar e recuperar dados/informações em arquivos.

5.1 Os tipos de arquivos

Para compreender a utilização de arquivos em C precisamos conhecer os diferentes tipos de arquivos que existem:

Arquivos somente-texto

Estes arquivos possuem somente caracteres de texto, exemplos deles são arquivos TXT, XML, HTML entre outros. Outra característica importante é que seu conteúdo costuma ser separado por *quebras de linha*. Além disso, os sistemas operacionais codificam a quebra de linha de forma diferente. Alguns aplicativos aceitam entradas no formato somente-texto, causando erro de execução caso encontre caracteres binários.

Arquivos binários

Estes arquivos possuem bytes que não representam caracteres textuais, exemplos deles arquivos MP3, JPEG, ZIP etc. O sistema operacional Windows requer que as funções de escrita e leitura dos arquivos se comporte de forma diferente, de acordo com o tipo de arquivo.

Arquivos especiais

São arquivos que não armazenam conteúdo, veremos este tipo no Apêndice B [161].

5.2 Arquivos e fluxos

Nesta seção vamos apresentar as características de fluxo de dados e arquivos e suas similaridades importantes para linguagem C.

5.2.1 Fluxo de dados

Para facilitar o aprendizado na utilização de arquivos em C precisamos conhecer o conceito de fluxo de dados (*data stream*).

O áudio que escutamos numa rádio é exemplo de um *fluxo de dados*. As estações de rádio transmitem o fluxo e seu aparelho toca todos os trechos que recebe. Uma vez sintonizado no fluxo, não há como pausar, retroceder ou avançar nele, só é possível tocar o que está sendo recebido.

O sistema de televisão também transmite as imagens através de fluxo, no entanto, existem aparelhos que possibilitam pausar a programação. Neste caso o aparelho passa a gravar todo o conteúdo que continua sendo recebido pelo fluxo em um Buffer, geralmente um disco interno. Quando o usuário remove o pause, a programação é exibida a partir do conteúdo salvo. Também é possível avançar no vídeo, mas somente até o final do Buffer, quando atinge o ponto de recepção do fluxo novamente.

Importante



O fluxo diferencia de arquivo pois, no fluxo, não podemos avançar ou retroceder na leitura. A leitura do fluxo costuma ser sequencial, realizando o processamento dos dados logo após a leitura. Como os arquivos possuem início e fim é possível determinar o seu tamanho. Já os fluxos possuem início, mas nem todos possuem fim.

5.2.2 Arquivos

O principal propósito dos arquivos é salvar um conteúdo para ser acessado futuramente. Para isso os arquivos possuem nomes e são organizados em diretórios (ou pastas), para facilitar sua identificação.

A organização dos nomes dos arquivos não é a mesma em todos os sistemas operacionais. No Windows, por exemplo, existe um diretório raiz por disco (C:\, D:\ etc.) e os diretórios são separados pela carácter “\” (*Contra barra*). Já no Unix, existe um único diretório raiz / onde todos os demais diretórios são *descendentes* dele, o carácter separador de diretório é o “/” (*Barra*), diferente do Windows.

Os arquivos somente texto também são diferentes, enquanto no Windows o final de linha é codificado com dois caracteres, o Linux utiliza apenas um.

5.2.3 Arquivos em C



Importante

A principal noção que precisamos ter em mente quando trabalhamos com arquivos em C é que **fluxos e arquivos são manipulados através de uma interface única**.

O tipo de dado para manipular Arquivos e Streams é o `FILE`, definido no cabeçalho `stdio.h` [172].

Na estrutura `FILE` são registradas todas as informações necessárias para sua manipulação:

- Um indicador da posição no arquivo.
- Um ponteiro para seu Buffer (caso exista).
- Um indicador de erro, que registra caso houve algum erro de leitura/escrita.
- Um indicativo de *final de arquivo*, onde é registrado se o final do arquivo foi atingido.



Nota

Perceba que a estrutura `FILE` não possui registro do nome do arquivo que está sendo manipulado.

5.3 Cabeçalho `stdio.h`

Todas as funções que serão apresentadas neste capítulo estão definidas no cabeçalho `stdio.h`, requerendo a inclusão deste arquivo para sua utilização:

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

5.4 Abrindo e Fechando arquivos em C

Toda vez que desejamos ler ou escrever em um arquivo precisamos **abrir o arquivo** indicando esta intenção.

5.4.1 Abrindo um arquivo com `fopen`

A abertura de arquivos em C é realizada através da função `fopen()` [172], que retorna um ponteiro `*FILE` caso a operação foi realizada com sucesso ou `NULL` caso houve erro na abertura:

```
FILE* arquivo = fopen(nomeDoArquivo, modo);
```

A intenção da abertura do arquivo deve ser especificada no parâmetro `modo`:

r	abre arquivo de texto para leitura
w	abre arquivo de texto para escrita
wx	cria arquivo de texto para escrita
a	adiciona ao final; o indicador de posição de arquivo é posicionado no final do arquivo
rb	abre arquivo binário para leitura
wb	abre arquivo binário para escrita
ab	abre arquivo binário para escrita, no final do arquivo

Os códigos a seguir demonstram a abertura de arquivo para *leitura no modo texto* (Exemplo 5.1 [83]) e outro para *escrita no modo binário* (Exemplo 5.2 [84]).

Exemplo 5.1 Abrindo um arquivo de texto para leitura

Código fonte /abrindo_arquivo_de_texto.c[[code/cap5/abrindo_arquivo_de_texto.c](#)]

```
#include <stdio.h>

/* Este programa demonstra a abertura de um arquivo de texto
 * através da função 'fopen'.
 * Caso o arquivo 'entrada.txt' exista, a abertura será realizada
 * com sucesso. Caso contrário, fopen irá retornar NULL, indicando
 * que não houve sucesso na abertura.
 * Ver também: abrindo_arquivo_binario_para_escrita.c
 */

int main(void) {
    char* nomeDoArquivo = "entrada.txt";
    FILE* arquivo = fopen(nomeDoArquivo, "r");
    if (arquivo != NULL) {
        printf("Arquivo '%s' foi aberto com sucesso.\n", nomeDoArquivo);
        // inicia leitura de arquivo texto

        // fecha arquivo
    }
```

```
    } else {  
        // exibe mensagem de erro na saída padrão de erro  
        fprintf(stderr, "Erro na abertura de '%s'.\n", nomeDoArquivo);  
    }  
}
```

Saída da execução

Arquivo 'entrada.txt' foi aberto com sucesso.

Exemplo 5.2 Abrindo um arquivo binário para escrita

Código fonte /abrindo_arquivo_binario_para_escrita.c[[code/cap5/abrindo_arquivo_binario_para_escrita.c](#)]

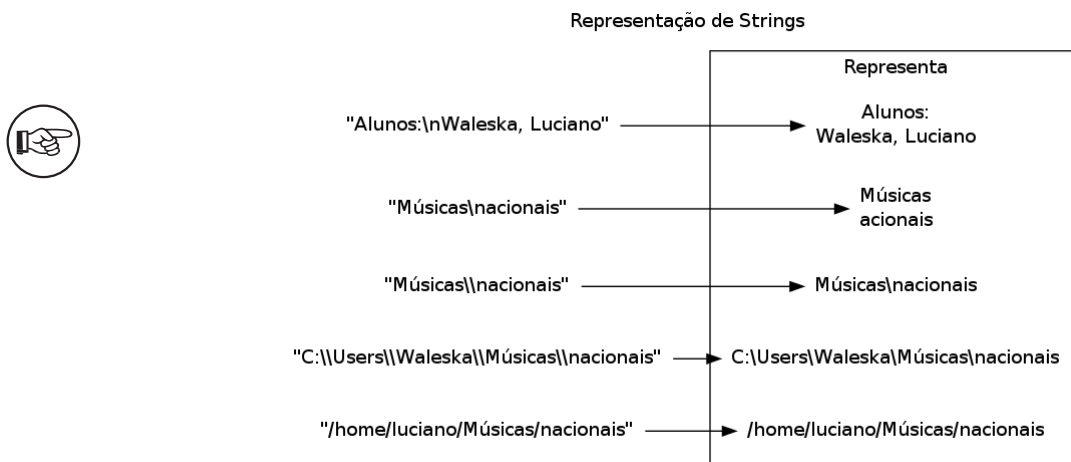
```
#include <stdio.h>  
  
/* Este programa demonstra a abertura de um arquivo binário  
 * através da função 'fopen'.  
 * Caso o arquivo 'arquivo-binario.bin' exista, a abertura será  
 * realizada com sucesso. Caso contrário, fopen irá retornar NULL,  
 * indicando que não houve sucesso na abertura.  
 * Ver também: abrindo_arquivo_de_texto.c, cria_arquivo_binario.c  
 */  
  
int main(void) {  
    char* nomeDoArquivo = "arquivo-binario.bin";  
    FILE* arquivo = fopen(nomeDoArquivo, "wb");  
    if (arquivo != NULL) {  
        printf("Arquivo '%s' foi aberto com sucesso.\n", nomeDoArquivo);  
        // inicia escrita binária do arquivo  
  
        // fecha arquivo  
    } else {  
        // exibe mensagem de erro na saída padrão de erro  
        fprintf(stderr, "Erro na abertura de '%s'.\n", nomeDoArquivo);  
    }  
}
```

Saída da execução

Arquivo 'arquivo-binario.bin' foi aberto com sucesso.

Dica

A especificação dos nomes dos arquivos requer que recordemos como os strings são escritos e representados na linguagem C:



5.4.2 Entrada e saídas padrões

O cabeçalho `stdio.h` [172] também define os seguintes arquivos como entradas e saídas padrões, que dispensam a operação de abertura:

FILE* stdin

Arquivo que representa a entrada padrão.

FILE* stdout

Arquivo que representa a saída padrão.

FILE* stderr

Arquivo que representa a saída de erro padrão.

**Nota**

Perceba que nos exemplos Exemplo 5.2 [84] e Exemplo 5.1 [83] a impressão das mensagens de erro são escritas da saída de erro `stderr`.

5.4.3 Fechando um arquivo

Independente do modo utilizado para abrir o arquivo, todo arquivo aberto deve ser fechado após sua utilização. Para fechar um arquivo utilizamos a função `fclose()` [173], passando como parâmetro o ponteiro `FILE*` retornado na abertura do arquivo:

Exemplo 5.3 Fechando um arquivo

Código fonte /fechando_arquivo.c[code/cap5/fechando_arquivo.c]

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

```
/* Este programa demonstra o FECHAMENTO de um arquivo,
 * através da função 'fclose'.
 * Supondo que o arquivo 'entrada.txt' exista e a operação de
 * abertura foi realizada com sucesso, após a finalização da leitura
 * do arquivo (não mostrado aqui), o arquivo deve ser fechado quando
 * não foi mais utilizado.
 */

int main(void) {
    char* nomeDoArquivo = "entrada.txt";
    FILE* arquivo = fopen(nomeDoArquivo, "r");
    if (arquivo != NULL) {
        printf("Arquivo aberto com sucesso...\n");
        // inicia leitura do arquivo

        printf("Fechando o arquivo...\n");
        fclose(arquivo); // ❶ fechando arquivo
    } else {
        // exibe mensagem de erro
        fprintf(stderr, "Erro na abertura de '%s'.\n", nomeDoArquivo);
    }
}
```

- ❶ A função para fechar o arquivo só deve ser invocada caso o arquivo foi aberto com sucesso.

Saída da execução

```
Arquivo aberto com sucesso...
Fechando o arquivo...
```

5.5 Indicadores de erro e final de arquivo

Na próximas seções serão apresentadas várias funções para ler e escrever em arquivos. Todas as funções que serão apresentadas registram eventos de erro e final de arquivo em indicadores na estrutura `FILE` que poderão ser consultado através das funções que serão apresentadas nesta seção.

5.5.1 Erro na leitura/escrita

Toda vez que um erro ocorreu nas operações de escrita ou leitura, a função que estava realizando a operação registra o ocorrido num indicador de erro em `FILE`, você poderá consultar o erro invocando a seguinte função:

```
int ferror(FILE *stream);
```

Haverá ocorrido erro caso o valor retornado for **diferente de zero**.

5.5.2 Atingiu o final do arquivo

Quando uma função atingiu o final do arquivo ela registra o ocorrido no indicador de final de arquivo, que pode ser consultado através da seguinte função:

```
int feof(FILE *stream);
```

Um retorno diferente de zero indica que o final do arquivo foi atingido.

5.6 Lendo e escrevendo um carácter por vez

Nesta seção vamos apresentar como ler e escrever no arquivo um carácter por vez.

5.6.1 Lendo um carácter com `fgetc` ou `getc`

A função `fgetc()` [173] ler um carácter do arquivo, converte-o para `int` e retorna-o. Caso não haja mais caracteres disponível, devido ao final do arquivo, retorna a constante `EOF` (do inglês, *End of File*, que indica o final do arquivo):

```
int fgetc(FILE *arquivo);
```

Após a leitura do carácter o indicador de posição de leitura do arquivo é incrementado em uma posição.

A função `getchar()` [173] realiza a mesma operação, mas não requer parâmetro, utilizando a entrada padrão (`stdin`) como arquivo de entrada:

```
int getchar(void);
```



Nota

Perceba a função de ler um carácter não retorna um **char**, retorna um **int**. Isto porque a `EOF` não é um carácter que indica o final final do arquivo, mas uma constante do tipo `int` que a função de leitura de carácter retorna ao atingir o final do arquivo.

5.6.2 Escrevendo um carácter por vez com `fputc` e `putchar`

A função `fputc()` [173] escreve um carácter (convertido para `unsigned char`) na posição atual do arquivo:

```
int fputc(int character, FILE *stream);
```

A função retorna o carácter escrito ou `EOF` se houve erro durante a escrita.

A função `putchar()` [173] realiza a mesma função que `fputc()` [173] para opera somente na saída padrão (`stdout`):

```
int putchar(int character);
```

5.6.3 Exemplo completo de leitura e escrita

O Exemplo 5.4 [88] demonstra um a utilização das funções de leitura e escrita com caracteres:

Exemplo 5.4 Lendo e escrevendo caracteres

Código fonte /lendo_e_escrevendo_char.c[code/cap5/lendo_e_escrevendo_char.c]

```
#include <stdio.h>

/* Este programa demonstra a leitura e escrita de arquivos
 * sendo realizadas um carácter por vez, através das
 * funções 'fgetc' e 'fputc'.
 * Para executar este programa certifique-se de que exista
 * conteúdo no arquivo 'entrada.txt'. Caso o arquivo 'saida.txt'
 * exista, ele será sobrescrito, caso não exista, ele será criado.
 * Ao final, o conteúdo de 'saida.txt' deve ser o mesmo
 * de 'entrada.txt'.
 * Ver também: lendo_e_escrevendo_linha.c
 */

int main(void) {
    char* arquivoParaLeitura = "entrada.txt";
    FILE* entrada = fopen(arquivoParaLeitura, "r");//❶
    if (entrada != NULL) { //❷
        char* arquivoParaEscrita = "saida.txt";
        FILE* saida = fopen(arquivoParaEscrita, "w");//❸
        if (saida != NULL) { //❹
            int charLido;
            do {
                charLido = fgetc(entrada); //❺
                if (charLido != EOF) { //❻
                    fputc(charLido, saida); //❼
                }
            } while (!feof(entrada)); //❽

            printf("Todos as caracteres de %s foram escritos em %s\n",
                arquivoParaLeitura, arquivoParaEscrita);

            fclose(saida); //❾
        } else {
            // exibe mensagem de erro, na saída padrão de erro 'stderr'
            fprintf(stderr, "Erro na abertura do arquivo 'entrada.txt'.\n");
        }

        fclose(entrada); //❿
    } else {
        // exibe mensagem de erro, não consegue ler
    }
}
```

❶, ❸ Abertura dos arquivos para leitura e escrita respectivamente.

❷, ❹ Verificando se os arquivos foram abertos com sucesso.

5, 7 Ler um carácter da entrada e escreve o carácter lido na saída.

6, 8 Verifica se atingiu o final do arquivo.

9, 10 Fecha os arquivos abertos.

Saída da execução

Todos as caracteres de entrada.txt foram escritos em saida.txt

5.7 Lendo e escrevendo uma linha por vez com `fgets` e `fputs`

Nesta seção vamos apresentar como ler e escrever no arquivo uma linha por vez.

5.7.1 Lendo uma linha com `fgets`

A função `fgets()` [174] ler uma linha do arquivo e salva o conteúdo lido no `buffer` passado como parâmetro. O final de linha é retido e retornando junto com a linha lida. A função copia caracteres do arquivo até encontrar o final da linha, ou chegou ao final do arquivo ou leu $n-1$ caracteres. O carácter `\0` é escrito após o último carácter lido, transformando `buffer` em um string terminado em zero, compatível com as funções `string.h` [171].

```
char *fgets(char * buffer, int n, FILE * arquivo);
```

A função retorna `buffer`, contendo a linha lida, ou `NULL` se houve algum erro durante a leitura.

Nota



Para entender o funcionamento da função `fgets()` [174] é importante perceber como o computador funciona. Inicialmente alguém poderia desejar que esta função retornasse um string contendo a linha lida. Mas pense bem: Se se o arquivo lido for enorme e não possuir quebra de linhas, a operação de leitura de uma linha poderia retornar um string que não coubesse na memória.

Como evitar que isto aconteça? Passando um parâmetro `n`, indicando a quantidade máxima de caracteres que deve ser lido na procura do final de linha.

Além disso, para evitar erros de alocação de memória, é solicitado que o usuário forneça previamente o Buffer onde será copiado os caracteres da linha lida.

5.7.2 Escrevendo um string com `fputs`

A função `fputs()` [174] escreve um string, terminado com o carácter zero, no arquivo.

```
int fputs(const char * string, FILE * arquivo);
```

A função retorna EOF em caso de erro.

Nota



Perceba que não há necessidade de especificar o tamanho na escrita, ela termina quando encontra o carácter nulo (`\0`). Outro fato importante é que **esta função não deve ser utilizado para escrita binária, pois ela não escreve o carácter nulo**.

5.7.3 Exemplo completo de leitura e escrita

O seguinte código é um exemplo completo utilizando as funções de leitura e escrita com linha:

Exemplo 5.5 Lendo e escrevendo linhas

Código fonte /lendo_e_escrevendo_linha.c[code/cap5/lendo_e_escrevendo_linha.c]

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

/* Este programa demonstra a leitura e escrita de arquivos
 * sendo realizadas uma linha por vez, através das
 * funções 'fgets' e 'fputs'.
 * Para executar este programa certifique-se de que exista
 * conteúdo no arquivo 'entrada.txt'. Caso o arquivo 'saida.txt'
 * exista, ele será sobrescrito, caso não exista, ele será criado.
 * Ao final, o conteúdo de 'saida.txt' deve ser o mesmo
 * de 'entrada.txt'.
 * Ver também: lendo_e_escrevendo_char.c
 */

int main(void) {
    char* arquivoParaLeitura = "entrada.txt";
    FILE* entrada = fopen(arquivoParaLeitura, "r");
    int bufferSize = 2048;
    char* buffer = calloc(bufferSize, 1);
    if (entrada != NULL) {
        char* arquivoParaEscrita = "saida.txt";
        FILE* saida = fopen(arquivoParaEscrita, "w");
        if (saida != NULL) {
            do {
                char* linha = fgets(buffer, bufferSize, entrada); //❶
                if (linha!=NULL){
                    fputs(buffer, saida); //❷
                }
            } while (!feof(entrada));

            printf("As linhas de %s foram escritos em %s\n",
                arquivoParaLeitura, arquivoParaEscrita);
            fclose(saida); // fechando arquivoParaEscrita
        } else {
            // exibe mensagem de erro, não consegue escrever na saída
            fprintf(stderr, "Erro na escrita em %s.\n",arquivoParaEscrita);
        }

        fclose(entrada); // fechando arquivoParaLeitura
    } else {
        // exibe mensagem de erro na saída padrão de erro
        fprintf(stderr, "Erro na abertura de '%s'.\n",arquivoParaLeitura);
    }
}
```

❶ Ler uma linha da entrada.

- ② Escreve o string lido na saída.

Saída da execução

As linhas de entrada.txt foram escritos em saida.txt

5.8 Lendo e escrevendo dados binários

As funções de leitura e escrita de dados binários diferem dos arquivos de textos devido a ao conteúdo que será salvo e lido.

5.8.1 Lendo uma sequencia de elementos de um arquivo com `fread`

A função `fread()` [174] ler uma sequência de elementos de um arquivo. Na função são passados como parâmetro o `buffer` onde será salvo a sequência lida, o `tamanho` do tipo de dado que será lido, a quantidade de elementos que serão lidos e o `arquivo` de onde será lido os dados.

```
size_t fread(void * buffer, size_t tamanho,
             size_t quantidade, FILE * arquivo);
```

A função retorna a quantidade de elementos lidos, que pode ser menor do que `quantidade`, caso o final do arquivo foi encontrado ou houve algum erro durante a leitura. Se `quantidade` ou `tamanho` for zero, então `buffer` permanece inalterado e a função retorna zero.

5.8.2 Escrevendo uma sequencia de elementos de um arquivo com `fwrite`

A função `fwrite()` [174] escreve uma sequência de elementos num arquivo. Na função são passados como parâmetro o `buffer` que é um ponteiro para o início da sequência de dados que se deseja escrever, o `tamanho` do tipo de elemento, a quantidade de elementos que se deseja escreve e o `arquivo` onde o conteúdo será escrito.

```
size_t fwrite(const void * buffer, size_t tamanho,
              size_t quantidade, FILE * arquivo);
```

A função retorna o número de elementos que foram escritos, que pode ser menor do que `quantidade` somente se houve erro durante a escrita. Caso `quantidade` ou `tamanho` seja zero então o arquivo permanece inalterado e a função retorna zero.

5.8.3 Exemplo completo de leitura e escrita binária

O seguinte código é um exemplo completo utilizando as funções de leitura e escrita apresentadas nesta seção.

Alguns jogos mantém uma lista de recordistas, que salva o nome do jogador e pontuação obtida. Neste código criamos uma lista contendo a pontuação inicial de três jogadores fictícios e salvamos num arquivo. Em seguida, demonstramos a leitura do arquivo e finalmente imprimimos os recordes lidos.

Exemplo 5.6 Escrevendo e lendo elementos em binário**Código fonte** /escrevendo_e_lendo_binario.c[[code/cap5/escrevendo_e_lendo_binario.c](#)]

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

/* O propósito deste programa é demonstrar a criação e leitura
 * de arquivos no formato binário.
 * Ele simula a criação de um arquivo 'pontuacao.data' que mantém
 * os recordes de jogadores de um jogo. Foi optado por salvar o
 * arquivo no formato binário para evitar que alguém edite a pontuação
 * com um editor de texto.
 * A função 'criaArquivoDeRecordistasIniciais' irá criar o arquivo
 * no formato binário, e 'lerArquivoDeRecordistas' irá ler o conteúdo
 * do arquivo. Por fim, a função 'imprimeRecordistas' irá imprimir os
 * dados lidos do arquivo, convertendo-os para caracteres, através da
 * função 'printf'.
 */

typedef struct Jogador_{
    int pontuacao;
    char nome[32];
} Jogador; //❶

Jogador recordistasIniciais[]={ //❷
    { .nome = "Anjinho", .pontuacao = 50 },
    { .nome = "Bido", .pontuacao = 25 },
    { .nome = "Chico Bento", .pontuacao = 0 }
};

int QUANTIDADE_DA_LISTA = 3; // Anjinho,Bido,Chico
char* nomeDoArquivoDePontuacao = "pontuacao.data";

void criaArquivoDeRecordistasIniciais(){
    FILE* arquivo = fopen(nomeDoArquivoDePontuacao, "wb");
    if (arquivo != NULL) {
        fwrite(&recordistasIniciais, sizeof(Jogador), //❸
            QUANTIDADE_DA_LISTA, arquivo);
        fclose(arquivo);
    } else {
        // mensagem de erro
        fprintf(stderr, "Não foi possível abrir o arquivo: %s\n",
            nomeDoArquivoDePontuacao);
    }
}

Jogador* lerArquivoDeRecordistas(){
    Jogador* jogadores = calloc(QUANTIDADE_DA_LISTA,
        sizeof(Jogador));
    FILE* arquivo = fopen(nomeDoArquivoDePontuacao, "rb");
    if (arquivo != NULL) {
        fread(jogadores, sizeof(Jogador), QUANTIDADE_DA_LISTA, //❹
```

```
        arquivo);
    fclose(arquivo);
} else {
    // mensagem de erro
    fprintf(stderr, "Não foi possível abrir o arquivo: %s\n",
        nomeDoArquivoDePontuacao);
}
return jogadores;
}

void imprimeRecordistas(Jogador* jogadores){//❶
    printf("Os seguintes valores foram lidos do arquivo binário:\n");
    for(int i=0; i<QUANTIDADE_DA_LISTA; i++){
        printf("%s %d\n", jogadores[i].nome, jogadores[i].pontuacao);
    }
}

int main(void) {
    criaArquivoDeRecordistasIniciais();
    Jogador* recordistas = lerArquivoDeRecordistas();
    imprimeRecordistas(recordistas);
}
```

- ❶ Declaração do tipo `Jogador`, com dois dados: seu nome e sua pontuação.
- ❷ Inicialização de um array de recordistas iniciais que serão salvos mais adiante.
- ❸ Salva os três recordistas no arquivo.
- ❹ Ler os três recordistas do arquivo e salva no buffer `jogadores`.
- ❺ Imprime os recordistas.

Saída da execução

```
Os seguintes valores foram lidos do arquivo binário:
Anjinho 50
Bido 25
Chico Bento 0
```

5.9 Descarregando o buffer com `fflush`

Alguns arquivos são abertos com buffers para escrita, isto implica que as operações de escritas no arquivo não são persistidas no arquivo até que o buffer esteja cheio ou o arquivo seja fechado. Para determinar que o conteúdo do buffer seja escrito no arquivo utilizamos a função `fflush()` [175]:

```
int fflush(FILE *stream);
```

A função retorna EOF caso houver erro na escrita do arquivo, caso contrário retorna zero.

5.10 Escrevendo e lendo dados formatados

Finalizando as seções de escrita e leitura de dados, vamos conhecer funções provavelmente lhe são familiares.

5.10.1 Ler dados formatados

Para ler dados formatados você pode utilizar a função `fscanf()` [175], que é similar a função `scanf()` [175], a única diferença é que `scanf` trabalha apenas com a entrada padrão, enquanto `sscanf` especifica o arquivo de entrada:

```
int fscanf(FILE * arquivo,
           const char * formato, ...);
```

5.10.2 Escrever dados formatados

A escrita de dados formatados é feita através da função `fprintf()` [176], que é similar a `printf()` [176] (que trabalha somente na saída padrão), a única diferença é que `fprintf` permite especificar o arquivo de saída:

```
int fprintf(FILE * arquivo,
           const char * formato, ...);
```

5.11 Movendo o indicador de posição

Como foi apresentado anteriormente, o **indicador de posição** mantém o registro da posição atual do arquivo.

Em alguns casos podemos ter a necessidade de avançar ou retroceder o ponto de leitura ou escrita do arquivo, as funções a seguir servem a este propósito.



Importante

As funções `rewind`, `ftell` e `fseek` serão utilizadas no Capítulo 6 [99]. Por enquanto é importante apenas que você reconheça a sua existência.

5.11.1 Retrocede para o início do arquivo com `rewind`

A função `rewind()` [175] retrocede o indicador de posição para o início do arquivo:

```
void rewind(FILE *arquivo);
```

5.11.2 Lendo a posição atual

A função `ftell()` [175] retorna a posição atual no arquivo:

```
long int ftell(FILE *arquivo);
```

5.11.3 Indo para o final do arquivo

Para ir para o final do arquivo invocamos podemos utilizar a função `fseek()` [175] da seguinte forma:

```
long fseek(arquivo, 0, SEEK_END);
```

5.12 Recapitulando

Neste capítulo foi apresentado os tipos de arquivos que existem, as similaridades entre fluxos e arquivos e como a linguagem abstrai os arquivos como fluxos. Em seguida aprendemos como abrir os arquivos antes de realizar operações neles, e como fechar para finalizar a operação. As operações para ler e escrever caracteres no arquivo possuíam um exemplo demonstrando sua utilização. Aprendemos que os arquivos possuem indicadores de erro, final de arquivo e de posição que são alterados durante as execuções das funções e leitura e escrita, e como consultar o estado destes indicadores. Vimos que existem funções próprias para trabalhar com arquivos de texto, que processam o arquivo por linhas e como utilizá-las. As funções de escrita e leitura de arquivos binários necessitam do tamanho do tipo de dado, a quantidade de dados que seriam processados e um buffer para realização das operações. Por último, vimos como alterar e consultar o indicador de posição do arquivo.

5.13 Atividades

1. Quais os tipos de arquivos que existem?
2. Quais as semelhanças e diferenças entre fluxo de dados e arquivos?
3. Faça um programa que leia um arquivo e conte quantas vezes a letra `b` está presente no arquivo.
4. Faça um programa semelhante ao anterior, mas que leia o conteúdo da entrada padrão.
5. Faça um programa que leia um arquivo e conte quantas linhas ele possui.
6. Alguns jogos possuem configurações padrões e permite que o usuário sobrescreva as configurações do usuário. Quando o jogo é iniciada o arquivo de configuração padrão é lido primeiro, em seguida o arquivo de configuração do usuário é lido, sobrepondo as configurações. Faça um programa que leia dois arquivos de configurações e imprima a configuração resultante deles:

Arquivo configuracao-padrao.txt

```
volume=100
dificuldade=3
musica=1
efeitos=1
```

Arquivo configuracao-usuario.txt

```
dificuldade=2
musica=0
```

7. Utilizando as mesma lógica dos arquivos de configurações da questão anterior, faça um programa que leia o nome de um parametro e o seu valor, em seguida altere somente o parâmetro fornecido no arquivo de configuração do usuário. Caso o parâmetro não exista, ele é criado no mesmo arquivo.

Por exemplo, ao ler os valores `musica 1`, o arquivo de configuração do usuário seria modificado para:

Arquivo configuracao-usuario.txt resultante da modificação

```
dificuldade=2
musica=1
```

Para os valores `volume 70`, o arquivo resultante seria:

Arquivo configuracao-usuario.txt resultante da modificação

```
dificuldade=2
musica=0
volume=70
```

8. A linguagem de programação C possui dois tipos de comentários: comentário de linha e comentário em bloco. Faça um programa que leia um arquivo de código fonte e imprima o código sem os comentários. As linhas em branco *podem* ser suprimidas.

Exemplo de arquivo com comentários

```
#include <stdio.h>

int main(void) {
    char* nomeDoArquivo = "arquivo-binario.bin";
    FILE* arquivo = fopen(nomeDoArquivo, "wb"); // abre arquivo
    if (arquivo != NULL) {
        /* arquivo aberto com sucesso...
        inicia escrita no arquivo */

        // fecha arquivo
    } else {
        // exibe mensagem de erro
    }
}
```

Impressão do arquivo sem os comentários

```
#include <stdio.h>
int main(void) {
    char* nomeDoArquivo = "arquivo-binario.bin";
    FILE* arquivo = fopen(nomeDoArquivo, "wb");
    if (arquivo != NULL) {
    } else {
    }
}
```

9. Em bancos de dados os conteúdos são armazenados em tabelas, cada linha é chamada de registro da tabela. Faça um programa que salve, no formato binário, a quantidade de habitantes de uma cidade, e sua posição geográfica. Para realização desta utilize o seguinte registro:

Registro Cidade

```
typedef struct Cidade_{  
    char cidade[32];  
    long double longitude;  
    long double latitude;  
    long habitantes;  
} Cidade;
```

Salve no mínimo 3 registros.



Nota

Ao realizar esta atividade você irá compreender a semelhança entre os conceitos de Registros em C e nos banco de dados.

10. Faça um programa que adicione novos registros ao final do arquivo do exercício anterior. Dica: utilize o modo `ab` para abrir o arquivo.
11. Faça um programa que leia o arquivo gerado na questão anterior e imprima somente as cidades com a quantidade de habitantes maior do que um valor que será solicitado ao usuário.



Feedback sobre o capítulo

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

Para compreender melhor como feedbacks funcionam consulte o guia do curso.

Parte II

Projeto

Capítulo 6

Lingua do i

OBJETIVOS DO CAPÍTULO

Ao final deste capítulo você deverá ser capaz de:

- Desenvolver um projeto em C;
- Utilizar testes para guiar a elaboração do projeto;
- Desenvolver uma aplicação que leia e escreva em arquivos;

Nós vamos aprender a manipular arquivos em C, através da construção de um pequeno aplicativo que possa ler de um arquivo o texto em português e convertê-lo para a Língua do i.

A **Lingua do i** é uma brincadeira infantil que consiste em falar ou escrever palavras trocando todas as vogais por “i”:

Ela ama banana. (em português)
Ili imi binini. (na língua do i)

O projeto será desenvolvido em diversas etapas, ao final de cada etapa você poderá consultar o estado final dos arquivos. Na primeira etapa iremos criar a estrutura inicial do projeto, o primeiro teste da aplicação e o Makefile para compilar o projeto.

6.1 Etapa 1: Estrutura inicial do projeto

Vamos começar nosso projeto criando uma estrutura inicial do nosso projeto, que consistirá em:

LISTA DE ARQUIVOS DO PROJETO

lingua-do-i.c

Conterá o `main` da aplicação.

lingua-do-i-core.c

Conterá as implementações das principais funções da aplicação, frequentemente este arquivo será referenciado apenas como o *core*.

lingua-do-i-core.h

Conterá as definições das funções de *core*.

lingua-do-i-test.c

Conterá os testes da aplicação.

musica-trecho.txt

Trecho de música que será utilizado no teste de leitura de arquivo.

Makefile

Makefile para compilar o projeto e executar os testes.

6.1.1 Criando um main inocente

Embora não iremos nos preocupar com o main da aplicação ainda, nosso primeiro passo será criar o main da aplicação `lingua-do-i` apenas registrar didaticamente o propósito que este arquivo terá.

Código fonte `/etapa1/src/lingua-do-i.c` [`code/lingua-do-i/etapa1/src/lingua-do-i.c`]

Código que contém o main do aplicativo, com implementação inocente

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

int main(void) {
    puts("!!!Hello World!!!"); // ❶
    return EXIT_SUCCESS; // ❷
}
```

- ❶ **Implementação inocente** para o main da aplicação.
- ❷ Retorna `EXIT_SUCCESS` indicando que a aplicação saiu sem erros.

**Nota**

`EXIT_SUCCESS` é definido em `stdlib.h` [170], junto com `EXIT_FAILURE` que tem o propósito inverso: indicar que houve erro na execução da aplicação.

**Dica**

O termo técnico **implementação inocente** provém do inglês: **dummy implementation**. Geralmente utilizado quando desejamos aguardar o momento oportuno para realizar a implementação apropriada, também costuma ter o propósito de fazer o código compilar ou apenas preencher algum trecho do código.

Nas próximas seções daremos mais atenção aos testes, enquanto iremos implementar, aos poucos, as funcionalidades que o aplicativo terá.

Por último, iremos integrar as funcionalidades implementadas ao aplicativo em `main`.

6.1.2 Iniciando um teste



Importante

Ao decorrer do capítulo iremos escrever vários testes, este será o primeiro deles. Leia com bastante atenção para perceber como eles são elaborados.

Decidimos começar os nossos testes tentando ler o conteúdo de um arquivo, o código a seguir foi escrito em `lingua-do-i-test.c`:

Primeiro teste

```
#include "lingua-do-i-core.h"

char* NOME_DO_ARQUIVO = "musica-trecho.txt";
char* CONTEUDO_ESPERADO = "Oh! Deus, perdoe este pobre coitado";
void testLerConteudoDoArquivo() {
    char* conteudo = lerConteudoDoArquivo(NOME_DO_ARQUIVO);
    verificaConteudosSaoIguais(conteudo, CONTEUDO_ESPERADO);
}
```

Você é capaz de identificar a ideia da implementação deste teste? Pense um pouco antes de prosseguir!

PONTOS A SEREM OBSERVADOS NO TESTE

1. O primeiro ponto a ser observado é o nome do teste: `testLerConteudoDoArquivo`. O nome de uma função deve ser suficiente para descrever qual é o seu propósito.
2. Em seguida, vemos a chamada da função `lerConteudoDoArquivo`, que ainda não existe, mas estamos planejando sua implementação.
Durante a elaboração dos testes nós escrevemos funções supondo a sua existência, mas tarde implementaremos elas.
Outro ponto importante, é que estamos definindo qual será interface da função `lerConteudoDoArquivo`: (a) ela terá somente um parâmetro que será o nome do arquivo que será lido; (b) ela irá retornar um ponteiro (`char*`) que conterà o conteúdo lido no arquivo.
3. O último comando irá chamar a função `verificaConteudosSaoIguais`, que também não foi implementada, mas comunica qual será seu propósito: verificar que o conteúdo retornado pela função `lerConteudoDoArquivo` corresponde ao conteúdo do arquivo.
4. A constante `NOME_DO_ARQUIVO` corresponde ao nome de um arquivo¹ que supostamente possui o mesmo conteúdo da variável `CONTEUDO_ESPERADO`.
Embora as constantes poderiam ser declaradas dentro das funções, a declaração externa possibilita uma maior legibilidade no código do teste, mas não há certo ou errado aqui, apenas estilo.
5. Um ponto que talvez passe despercebido é a inclusão do arquivo de cabeçalho `lingua-do-i-core.h`. Ao fazermos isso, estamos antecipando uma dependência que nosso teste terá com `lingua-do-i-core.c`. Estamos planejando implementar a função `lerConteudoDoArquivo` neste arquivo.
Durante o processo de compilação dos testes precisaremos ligar (*linkar*) o arquivo de teste com este código fonte. O Makefile do projeto deverá refletir esta ligação.

Nosso próximo passo será elaborar o Makefile para que será responsável pela compilação do projeto.

¹O arquivo deve existir no mesmo diretório do executável

6.1.3 Elaborando o Makefile

Estamos planejando a elaboração de uma aplicação e um teste, portanto nosso Makefile ficou da seguinte forma:

Código fonte /etapa1/src/Makefile[[code/lingua-do-i/etapa1/src/Makefile](#)]

Makefile do projeto, demonstra como compilar os programas

```
CC=gcc
CFLAGS=-Wall -g -std=c1x

all: lingua-do-i lingua-do-i-test

lingua-do-i: lingua-do-i.c
    $(CC) $(CFLAGS) lingua-do-i.c lingua-do-i-core.c -o lingua-do-i

lingua-do-i-test: lingua-do-i-test.c
    $(CC) $(CFLAGS) lingua-do-i-test.c lingua-do-i-core.c -o lingua-do-i-test

test_all:
    ./lingua-do-i-test

clean:
    rm -rf lingua-do-i lingua-do-i-test
```



Nota

Este arquivo foi criado para funcionar no Linux, para funcionar em outras plataformas serão necessários pequenos ajustes.

CC

Indica qual o compilador estamos utilizando. Caso utilize outro, configure o seu compilador aqui.

CFLAGS

Configura algumas *flags* de compilação. Inclui a utilização do padrão C1X (Seção D.1 [169]). Caso seu compilador aceite estas *flags*, não haverá necessidade de configuração.

all

Indica quais as regras serão executadas por padrão quando executamos o comando `make` sem nenhum parâmetro.

lingua-do-i e lingua-do-i-test

Regras para construção dos aplicativos.

test_all

Regra para execução dos testes. Consiste em invocar o aplicativo de teste.

clean

Regra para apagar os arquivos construídos. A implementação desta regra consiste em invocar o comando `rm` no Linux, que realiza a exclusão dos arquivos passados como parâmetros.

Para compilar os aplicativos podemos executar

```
make clean && make all
```

Para compilar e testar os aplicativos podemos executar

```
make clean && make all && make test_all
```

6.1.4 Criando um main e compilando o teste

Vamos adicionar um main ao nosso arquivo de teste e chamar a função `testLerConteudoDoArquivo` a partir dele:

Atualizando `lingua-do-i-test.c` para invocar o teste

```
#include <stdlib.h>
#include "lingua-do-i-core.h"

char* NOME_DO_ARQUIVO = "musica-trecho.txt";
char* CONTEUDO_ESPERADO = "Oh! Deus, perdoe este pobre coitado";
void testLerConteudoDoArquivo() {
    char* conteudo = lerConteudoDoArquivo(NOME_DO_ARQUIVO);
    verificaConteudosSaoIguais(conteudo, CONTEUDO_ESPERADO);
}

int main(void) {
    testLerConteudoDoArquivo();

    return EXIT_SUCCESS;
}
```

Nós temos algumas funções que ainda não foram escritas, mas vamos tentar compilar o código mesmo assim, para verificar os erros que serão apresentados pelo compilador:

Execução do comando para compilação

```
$ make clean && make lingua-do-i-test
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o ↵
    lingua-do-i-test
lingua-do-i-test.c: Na função 'testLerConteudoDoArquivo':
lingua-do-i-test.c:7:2: aviso: implicit declaration of function ↵
    'lerConteudoDoArquivo' [-Wimplicit-function-declaration]
lingua-do-i-test.c:7:19: aviso: initialization makes pointer ↵
    from integer without a cast [habilitado por padrão]
lingua-do-i-test.c:8:2: aviso: implicit declaration of function ↵
    'verificaConteudosSaoIguais' [-Wimplicit-function-declaration ↵
    ]
/tmp/ccEcEbRI.o: In function 'testLerConteudoDoArquivo':
/home/santana/asciibook/linguagem-de-programacao-i-livro/livro/ ↵
    capitulos/code/lingua-do-i/etapa1/src/lingua-do-i-test.c:7:
```

```
undefined reference to `lerConteudoDoArquivo' ❶  
/home/santana/asciibook/linguagem-de-programacao-i-livro/livro/ ↵  
  capitulos/code/lingua-do-i/etapa1/src/lingua-do-i-test.c:8:  
undefined reference to `verificaConteudosSaoIguais' ❷  
collect2: ld returned 1 exit status  
make: ** [lingua-do-i-test] Erro 1
```

- ❶, ❷ Mensagem informando que não foram encontradas referências às funções `lerConteudoDoArquivo` e `verificaConteudosSaoIguais`.

Como esperado, ocorreu erro no processo de compilação devido a utilização de funções que não foram implementadas ainda.

Dica

Uma abordagem comum no desenvolvimento utilizando testes é:



1. Escrever o teste supondo a existência das funções.
 2. Fazer o teste compilar utilizando o compilador para determinar o próximo ponto que será implementado.
 3. Fazer o teste falhar.
 4. Fazer o teste passar.
 5. Realizar ajustes se necessário.
-

6.1.5 Implementando a função `verificaConteudosSaoIguais`

Nosso próximo passo será a implementação da função que verifica se o teste irá falhar ou não:

Implementando função `verificaConteudosSaoIguais` em `lingua-do-i-test.c`

```
#include <string.h> ❶  
  
...  
  
void verificaConteudosSaoIguais(char* conteudo, char* esperado) {  
    if (conteudo == NULL) {  
        exit(EXIT_FAILURE); // não pode ser NULL ❷  
    }  
    int comparacao = strcmp(conteudo, esperado); // ❸  
    if (comparacao != 0) {  
        exit(EXIT_FAILURE); // strings tem que ser iguais ❹  
    };  
}
```

- ❶ Inclusão de `string.h`, que possui várias funções de manipulação de strings, inclusive comparação.
-

- ❷, ❹ Finaliza aplicação indicando que houve um erro.
- ❸ Executa `strcmp` (definida em `string.h` [171]) que compara dois strings, retorna 0 somente se ambos os string forem iguais. A variável `comparacao` irá guardar o resultado da comparação.

Com a implementação desta função esperamos que o teste falhe caso `conteudo` for diferente de esperado. Vamos compilar novamente para verificar os erros:

Compilação após implementação da função `verificaConteudosSaoIguais`

```
$ make clean && make lingua-do-i-test
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o
lingua-do-i-test
lingua-do-i-test.c: Na função 'testLerConteudoDoArquivo':
lingua-do-i-test.c:18:2: aviso: implicit declaration of function
'lerConteudoDoArquivo' [-Wimplicit-function-declaration]
lingua-do-i-test.c:18:19: aviso: initialization makes pointer ↵
    from
integer without a cast [habilitado por padrão]
/tmp/ccqV7RJy.o: In function 'testLerConteudoDoArquivo':
/home/santana/asciibook/linguagem-de-programacao-i-livro/livro/ ↵
    capitulos/code/lingua-do-i/etapa1/src/lingua-do-i-test.c:18:
undefined reference to 'lerConteudoDoArquivo'
collect2: ld returned 1 exit status
make: ** [lingua-do-i-test] Erro 1
```

Ótimo, agora só estamos com um erro, devido a ausência da implementação de `lerConteudoDoArquivo`.

6.1.6 Implementação inocente de `lerConteudoDoArquivo`

Vamos adicionar uma implementação inocente de `lerConteudoDoArquivo` somente para conseguir compilar o código.

Código fonte `/etapa1/src/lingua-do-i-core.c` [`code/lingua-do-i/etapa1/src/lingua-do-i-core.c`]

Core do programa, onde as funções serão implementadas

```
#include <stdio.h>
#include <stdlib.h>
#include "lingua-do-i-core.h" // ❶

char* lerConteudoDoArquivo(char* nomeDoArquivo) {
    return NULL; // ❷
}
```

- ❶ Inclusão do arquivo de cabeçalho, onde estarão definidas as funções públicas.
- ❷ Implementação inocente da função que desejamos implementar.

Com esta função implementada, vamos tentar compilar novamente:

Compilação após implementação da função que faltava

```
$ make clean && make lingua-do-i-test
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o ↵
    lingua-do-i-test
lingua-do-i-test.c: Na função 'testLerConteudoDoArquivo':
lingua-do-i-test.c:18:2: aviso: implicit declaration of function ↵
    'lerConteudoDoArquivo' [-Wimplicit-function-declaration]
lingua-do-i-test.c:18:19: aviso: initialization makes pointer ↵
    from integer without a cast [habilitado por padrão]
```

Algo estranho aconteceu, apesar de realizar a implementação da última função que estava faltando. O compilador continua indicando que não encontrou a função! Por que será?

6.1.7 Implementando o cabeçalho de core

Percebemos que ao compilar o arquivo de teste, não foi possível encontrar a função `lerConteudoDoArquivo` que estava implementada no arquivo `lingua-do-i-core.c`.

Isto costuma ocorrer pelas seguintes razões:

- A função não existe
- A função existe mas não está visível no processo de compilação devido a ligação ou falta de inclusão do arquivo de cabeçalho, indicando as definições das funções.

No nosso caso, faltou implementar o arquivo de cabeçalho, que já estava sendo incluído em ambos os arquivos:

Código fonte /etapa1/src/lingua-do-i-core.h[code/lingua-do-i/etapa1/src/lingua-do-i-core.h]

Cabeçaho do Core

```
#ifndef LINGUA_DO_I_CORE_H_ //❶
#define LINGUA_DO_I_CORE_H_ //❷

char* lerConteudoDoArquivo(char* nomeDoArquivo); //❸

#endif //❹
```

❶, ❷, ❹ Estratégia para definição de cabeçalho que evita erros caso o arquivo seja incluído mais de uma vez. Todos os arquivos de cabeçalho da biblioteca padrão de C utiliza esta estratégia.

❸ Definição da função que foi implementada no arquivo `.c`

Vamos compilar novamente!

Compilando após implementação do cabeçalho

```
$ make clean && make lingua-do-i-test
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o ↵
    lingua-do-i-test
```

Ótimo! O arquivo agora está compilando.

6.1.8 Arquivo texto para o teste

Antes de executar o teste, ainda é preciso criar o arquivo de texto contendo o conteúdo de `CONTEUDO_ESPERADO`. Escolhi uma música de Luiz Gonzaga, chamada *Súplica Cearense*, e salvei apenas a primeira frase no arquivo `musica-trecho.txt`.

Código fonte `/etapa1/src/musica-trecho.txt`[code/lingua-do-i/etapa1/src/musica-trecho.txt]

Conteúdo do arquivo que está sendo utilizado no testes

```
Oh! Deus, perdoe este pobre coitado
```

O conteúdo deste arquivo contém apenas uma frase, para simplificar a execução do teste. A codificação utilizada na escrita do arquivo foi a UTF-8.

O programa `hexdump`, disponível no Linux, exibe o conteúdo do arquivo em hexadecimal. Nele você poderá ver como os caracteres estão escritos no disco:

Execução do `hexdump` para verificar conteúdo do arquivo em hexadecimal

```
$ hexdump musica-trecho.txt
00000000 684f 2021 6544 7375 202c 6570 6472 656f
00000100 6520 7473 2065 6f70 7262 2065 6f63 7469
00000200 6461 006f
0000023
```

Pela execução do `hexdump`, verificamos que o arquivo possui **0x23 bytes de tamanho** (equivale a $2 \times 16 + 3 = 35$), indicado na última linha.

Também podemos perceber que todos os caracteres foram codificados com 1 byte cada (basta contar os caracteres).

O arquivo não termina com o carácter de fim de linha — ele finaliza apenas com o carácter `o`, que tem seu código ASCII igual a `0x6f`.



Dica

Você pode consultar a tabela ASCII em <http://www.asciitable.com>, ou se estiver no Linux através do comando `man ascii`.



Importante

Ao criar um arquivo de texto para os testes, certifique-se de conhecer o conteúdo do arquivo criado. As vezes os editores de texto podem adicionar caracteres de fim de linha que talvez não esteja visível para você. A execução do `hexdump` é recomendada nestes casos.

6.1.9 Verificando o teste falhar

O nosso código já está compilando, agora vamos executar o teste e verificar o resultado:

Execução do teste

```
$ make clean && make lingua-do-i-test && make test_all
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o ↵
    lingua-do-i-test
./lingua-do-i-test
make: ** [test_all] Erro 1
```

Como esperado, nosso teste está falhando. Lembra que nossa função `lerConteudoDoArquivo` possui uma implementação inocente? Não há como passar com uma implementação daquela.

Apesar do erro esperado, a mensagem de erro do teste não foi muito amigável, indicando apenas que houve `Erro 1`. Vamos corrigir isto na próxima etapa.

6.1.10 Código da etapa

Você pode consultar o código final desta etapa nos seguintes arquivos:

Código fonte /etapa1/src/lingua-do-i.c[code/lingua-do-i/etapa1/src/lingua-do-i.c]

Código fonte /etapa1/src/lingua-do-i-test.c[code/lingua-do-i/etapa1/src/lingua-do-i-test.c]

Código fonte /etapa1/src/lingua-do-i-core.c[code/lingua-do-i/etapa1/src/lingua-do-i-core.c]

Código fonte /etapa1/src/lingua-do-i-core.h[code/lingua-do-i/etapa1/src/lingua-do-i-core.h]

Código fonte /etapa1/src/musica-trecho.txt[code/lingua-do-i/etapa1/src/musica-trecho.txt]

Código fonte /etapa1/src/Makefile[code/lingua-do-i/etapa1/src/Makefile]

Certifique-se de ter compreendido esta etapa antes de prosseguir para a próxima.

6.2 Etapa 2: Utilizando `assert` em vez de `exit`

Na final da etapa anterior vimos que nosso teste não apresentava um mensagem de erro amigável (informando qual a razão da falha). Nesta etapa vamos apenas utilizar a função `assert` em vez de `exit (EXIT_FAILURE)`, para mudar isso.

A função `assert`, definida em `assert.h` [169], costuma ser utilizada para depuração dos programas. Caso a função receba uma expressão com o valor igual a 0 (zero) causará uma falha na execução da aplicação, indicando, através de uma mensagem, a linha do código fonte onde a falha está ocorrendo.

6.2.1 Utilizando `assert` no teste

Neste passo incluímos a biblioteca `assert.h` [169] e atualizamos a função `verificaConteudosSaoIguais` para chamar a função `assert`, incluindo uma mensagem para descrever o erro:

```
#include <assert.h> ❶
// (...)
void verificaConteudosSaoIguais(char* conteudo, char* esperado) {
    assert(conteudo != NULL && "conteúdo não pode ser NULL"); ❷
```

```
assert( strcmp(conteudo, esperado) == 0 ❸
        && "conteúdo deve ser igual ao esperado");
}
```

❶ Inclusão da biblioteca `assert.h`

❷, ❸ Atualização das verificações utilizando `assert`

6.2.2 Compilação e execução do teste

Vamos compilar e executar o teste, verificando que ele continuará falhando, mas agora com uma mensagem de erro amigável:

```
$ make clean && make && make test_all
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i.c lingua-do-i-core.c -o lingua-do-i
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o lingua-do-i-test
./lingua-do-i-test
lingua-do-i-test: lingua-do-i-test.c:7:
    verificaConteudosSaoIguais: Assertion 'conteudo != ((void *)
    0) && "conteúdo não pode ser NULL"' failed.
make: *** [test_all] Abortado (arquivo core criado)
```

Apesar do teste continuar falhando, agora nós temos certeza do ponto onde está ocorrendo a falha, sabemos exatamente o arquivo e a linha, como indicado na mensagem: `lingua-do-i-test.c:7`.

Maravilha! Agora estamos recebendo mensagens mais informativas.

6.2.3 Código da etapa

Você pode consultar o código final desta etapa nos seguintes arquivos:

Código fonte /etapa2/src/lingua-do-i.c[[code/lingua-do-i/etapa2/src/lingua-do-i.c](#)]

Código fonte /etapa2/src/lingua-do-i-test.c[[code/lingua-do-i/etapa2/src/lingua-do-i-test.c](#)]

Código fonte /etapa2/src/lingua-do-i-core.c[[code/lingua-do-i/etapa2/src/lingua-do-i-core.c](#)]

Código fonte /etapa2/src/lingua-do-i-core.h[[code/lingua-do-i/etapa2/src/lingua-do-i-core.h](#)]

Código fonte /etapa2/src/musica-trecho.txt[[code/lingua-do-i/etapa2/src/musica-trecho.txt](#)]

Código fonte /etapa2/src/Makefile[[code/lingua-do-i/etapa2/src/Makefile](#)]

6.3 Etapa 3: Fazendo o teste passar

Nesta etapa iremos fazer o teste passar, para garantir que nosso teste está funcionando corretamente.

Para isso é necessário apenas que nossa função `lerConteudoDoArquivo` retorne o valor esperado, que corresponde ao conteúdo do arquivo.

6.3.1 Implementação para fazer o teste passar

Este passo consiste em realizar as alterações necessárias para o teste passar.

Código fonte /etapa3/src/lingua-do-i-core.c[code/lingua-do-i/etapa3/src/lingua-do-i-core.c]

Core do programa, onde as funções serão implementadas

```
#include <stdio.h>
#include <stdlib.h>
#include "lingua-do-i-core.h"

char* musica = "Oh! Deus, perdoe este pobre coitado"; // ❶
char* lerConteudoDoArquivo(char* nomeDoArquivo) {
    return musica; // ❷
}
```

❶ inclusão de variável com o valor esperado

❷ retornando o valor esperado



Importante

Com esta implementação parece óbvio o teste passará, mas é importante executá-lo e vê-lo passando, pois as vezes a verificação do teste está implementada errada, proporcionando conclusões erradas.

Após as modificações vamos verificar se as funções copiadas realmente fazem o que se propões a fazer corretamente:

Compilando e verificando o teste passar

```
$ make clean && make && make test_all
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i.c lingua-do-i-core.c -o lingua ←
    -do-i
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o ←
    lingua-do-i-test
./lingua-do-i-test
```

6.3.2 Código da etapa

Você pode consultar o código final desta etapa nos seguintes arquivos:

Código fonte /etapa3/src/lingua-do-i.c[code/lingua-do-i/etapa3/src/lingua-do-i.c]

Código fonte /etapa3/src/lingua-do-i-test.c[code/lingua-do-i/etapa3/src/lingua-do-i-test.c]

Código fonte /etapa3/src/lingua-do-i-core.c[code/lingua-do-i/etapa3/src/lingua-do-i-core.c]

Código fonte /etapa3/src/lingua-do-i-core.h[code/lingua-do-i/etapa3/src/lingua-do-i-core.h]

Código fonte /etapa3/src/musica-trecho.txt[code/lingua-do-i/etapa3/src/musica-trecho.txt]

Código fonte /etapa3/src/Makefile[code/lingua-do-i/etapa3/src/Makefile]

6.4 Etapa 4: Lendo do arquivo

Nesta etapa precisamos implementar a leitura do arquivo e verificar o teste passando.



Dica

No link <https://gist.github.com/edusantana/8291576> existem vários testes com operações em arquivo, você pode utilizá-lo para consultar como realizar diversas operações, algumas funções deste capítulo estão disponíveis lá. Este link é um bom recurso de aprendizagem.

6.4.1 Abrindo e fechando arquivo para leitura

Antes de iniciar a leitura do conteúdo do arquivos, é necessário abrir o arquivo para leitura:

```
char* lerConteudoDoArquivo(char* nomeDoArquivo) {  
    char* conteudo; ❶  
  
    FILE* arquivo = fopen(nomeDoArquivo, "r"); // ❷  
    if (arquivoAbertoComSucesso(arquivo)) { ❸  
        conteudo = lerConteudoDeArquivoArberto(arquivo); ❹  
        fclose(arquivo); // fecha arquivo ❺  
    } else {  
        conteudo = NULL; ❻  
    }  
  
    return conteudo; ❼  
}
```

- ❶ Variável `conteudo` irá armazenar o conteúdo do arquivo.
- ❷ Utilização da função `fopen()` [172] para abrir o arquivo para leitura.
- ❸ Função que irá verificar se o arquivo foi aberto com sucesso.
- ❹ Função que irá ler o conteúdo do arquivo que já foi aberto com sucesso.
- ❺ Todo arquivo que é aberto precisa ser fechado após sua utilização.
- ❻ Retorna `NULL` em caso de erro na abertura.
- ❼ Retorna o conteúdo lido ou `NULL` se houve algum erro na abertura do arquivo.

Esta função utiliza outras que ainda não foram criadas e serão apresentadas nas próximas seções: `arquivoAbertoComSucesso` e `lerConteudoDeArquivoArberto`.

6.4.2 Verificando se o arquivo foi aberto com sucesso

Verificar se um arquivo foi aberto com sucesso é muito simples:

Função para verificar se arquivo foi aberto com sucesso

```
#include <stdbool.h> ❶
...

bool arquivoAbertoComSucesso(FILE* arquivo) {
    return arquivo != NULL; ❷
}
```

- ❶ Inclusão da biblioteca `stdbool.h` [172], que define o tipo `bool`. Este tipo pode ser utilizado como retorno de expressões lógicas. Além disso, na biblioteca também estão definidos macro `true` com o valor 1 e `false` com o valor 0.
- ❷ A função `fopen` retorna `NULL` caso houve algum erro na abertura do arquivo.

Na prática, não há necessidade de utilizar o tipo `bool`, mas esta solução é mais elegante.

6.4.3 Lendo conteúdo do arquivo aberto

Para criar um string com o conteúdo do arquivo, primeiro precisamos alocar uma sequência contínua de memória que comporte o conteúdo que será lido. Além disso, devemos alocar um byte a mais pois um string deve terminar com o carácter `\0`, para utilização das funções de string. No entanto, nem sempre é possível alocar espaço na memória para armazenar o conteúdo inteiro do arquivo inteiro.

A função que ler o conteúdo do arquivo foi implementada assim:

```
char* lerConteudoDeArquivoArberto(FILE* arquivo) {
    int tamanhoDoArquivo = lerTamanhoDoArquivo(arquivo); ❶
    char* conteudo = calloc(1, tamanhoDoArquivo+1); ❷

    fread(conteudo, tamanhoDoArquivo, 1, arquivo); ❸

    return conteudo; ❹
}
```

- ❶ Nesta implementação estamos lendo o conteúdo inteiro do arquivo, para isso precisamos saber qual o *tamanho total* do arquivo. Vamos implementar a função `lerTamanhoDoArquivo` que será responsável apenas por obter o tamanho do arquivo.
- ❷ Com o tamanho do arquivo salvo na variável `tamanhoDoArquivo`, será necessário alocar (`calloc()` [170]) o espaço para guardar todo o conteúdo do arquivo e mais um byte zero, para indicar o final do String.
- ❸ Com o buffer `conteudo` inicializado com zeros, chamamos a função `fread()` [174] para ler o conteúdo do arquivo e salvá-lo no buffer `conteudo`.
- ❹ Por fim, retornamos o buffer com o conteúdo lido.

As funções devem possuir responsabilidades limitadas, por isso decidimos delegar a responsabilidade de ler o tamanho do arquivo para outra função apenas com este propósito: `lerTamanhoDoArquivo`.

6.4.4 Descobrindo o tamanho do arquivo

Para ler o tamanho de um arquivo é necessário ir para o final dele e solicitar a posição atual, que corresponde a quantidade de bytes que o arquivo possui. Caso o arquivo seja um fluxo (*stream*) talvez não seja possível ir **ao final do fluxo**.

Função lerTamanhoDoArquivo

```
/* Efeito colateral:
Cabeçote de leitura vai para o início do arquivo.*/
int lerTamanhoDoArquivo(FILE* arquivo) {
    fseek(arquivo, 0, SEEK_END); // vai para o final do arquivo
    int tamanho = ftell(arquivo); // pega posição atual (final)

    fseek(arquivo, 0, SEEK_SET); // volta para o início
    return tamanho;
}
```

Na implementação:

1. Utilizando a função `fseek()` [175] para mover a cabeça de leitura para o final do arquivo.
2. Em seguida, chamamos a função `ftell()` [175] para pegar a posição atual de leitura do arquivo, que irá corresponder ao tamanho do arquivo.
3. Retornamos para o início do arquivo, para que próximas leituras ocorram no início.
4. Retornamos o tamanho do arquivo
5. Registramos como comentário o **Efeito colateral** da função.

6.4.5 Executando o teste

Agora que implementamos as funções para leitura, vamos executar o teste:

Executando o teste

```
$ make clean && make lingua-do-i-test && make test_all
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o ↵
    lingua-do-i-test
./lingua-do-i-test
```

Nosso teste continua passando! Isto quer dizer que nossa função foi capaz de ler o conteúdo do arquivo corretamente.

6.4.6 Código da etapa

Você pode consultar o código final desta etapa nos seguintes arquivos:

Código fonte /etapa4/src/lingua-do-i.c[[code/lingua-do-i/etapa4/src/lingua-do-i.c](#)]

Código fonte /etapa4/src/lingua-do-i-test.c[[code/lingua-do-i/etapa4/src/lingua-do-i-test.c](#)]

Código fonte /etapa4/src/lingua-do-i-core.c[code/lingua-do-i/etapa4/src/lingua-do-i-core.c]

Código fonte /etapa4/src/lingua-do-i-core.h[code/lingua-do-i/etapa4/src/lingua-do-i-core.h]

Código fonte /etapa4/src/musica-trecho.txt[code/lingua-do-i/etapa4/src/musica-trecho.txt]

Código fonte /etapa4/src/Makefile[code/lingua-do-i/etapa4/src/Makefile]

6.5 Etapa 5: Trocando as vogais do string por i

6.5.1 Inclusão de teste de tradução

Vamos iniciar a implementação desta etapa construindo um teste de tradução, que consistirá em traduzir as vogais para a letra i:

```
char* MENSAGEM_ORIGINAL="Minhas vogais, tudo aqui.";❶
char* TRADUCAO_ESPERADA="Minhis vigiis, tidi iqui.";❷
void testTraducaoParaLinguaDoI() {
    char* mensagemTraduzida = traduzParaLinguaDoI(MENSAGEM_ORIGINAL);❸
    verificaConteudosSaoIguais(mensagemTraduzida, TRADUCAO_ESPERADA);❹
}
```

- ❶ Mensagem original que será traduzida.
- ❷ Tradução esperada para a mensagem original.
- ❸ Invoca função `traduzParaLinguaDoI` que deverá traduzir a mensagem passada como parâmetro e retornar sua tradução.
- ❹ Verifica que o resultado da tradução é idêntico ao esperado (`TRADUCAO_ESPERADA`).

Este teste consiste em invocar a função `traduzParaLinguaDoI` passando `MENSAGEM_ORIGINAL` para ser traduzida.

Com o teste elaborado, vamos invocar o compilador para descobrir o próximo passo:

Invocando o compilador para descobrir o próximo passo

```
$ make clean && make lingua-do-i-test && make test_all
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o ↵
    lingua-do-i-test
lingua-do-i-test.c: Na função 'testTraducaoParaLinguaDoI':
lingua-do-i-test.c:24:2: aviso: implicit declaration of function ↵
    'traduzParaLinguaDoI' [-Wimplicit-function-declaration]
lingua-do-i-test.c:24:28: aviso: initialization makes pointer ↵
    from integer without a cast [habilitado por padrão]
/tmp/ccsT5OZs.o: In function 'testTraducaoParaLinguaDoI':
/home/santana/asciibook/linguagem-de-programacao-i-livro/livro/ ↵
    capitulos/code/lingua-do-i/etapa5/src/lingua-do-i-test.c:24: ↵
    undefined reference to 'traduzParaLinguaDoI'
collect2: ld returned 1 exit status
make: ** [lingua-do-i-test] Erro 1
```

A função `traduzParaLinguaDoI` que utilizamos no teste ainda não existe, vamos criá-la.

6.5.2 Criando a função traduzParaLingaDoI

Primeiro precisaremos atualizar o cabeçalho do core:

Adicionando função em lingua-do-i-core.h

```
char* traduzParaLingaDoI(char* mensagemOriginal);
```

Em seguida, vamos criar a função com uma implementação inocente:

Implementação inocente de traduzParaLingaDoI em lingua-do-i-core.c

```
char* traduzParaLingaDoI(char* mensagemOriginal){  
    return NULL;  
}
```

Vamos compilar e executar os nossos testes para descobrir o próximo passo:

Compilando e executando os testes:

```
$ make clean && make lingua-do-i-test && ./lingua-do-i-test  
rm -rf lingua-do-i lingua-do-i-test  
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o ↵  
    lingua-do-i-test  
lingua-do-i-test: lingua-do-i-test.c:8: ↵  
    verificaConteudosSaoIguais: Assertion 'conteudo != ((void *) ↵  
    0) && "conteúdo não pode ser NULL"' failed.  
Abortado (imagem do núcleo gravada)
```

Ótimo! O código compilou e estamos com o teste falhando. Nossa próxima etapa será fazer o teste passar.

6.5.3 Código da etapa

Você pode consultar o código final desta etapa nos seguintes arquivos:

Código fonte /etapa5/src/lingua-do-i.c[[code/lingua-do-i/etapa5/src/lingua-do-i.c](#)]

Código fonte /etapa5/src/lingua-do-i-test.c[[code/lingua-do-i/etapa5/src/lingua-do-i-test.c](#)]

Código fonte /etapa5/src/lingua-do-i-core.c[[code/lingua-do-i/etapa5/src/lingua-do-i-core.c](#)]

Código fonte /etapa5/src/lingua-do-i-core.h[[code/lingua-do-i/etapa5/src/lingua-do-i-core.h](#)]

Código fonte /etapa5/src/musica-trecho.txt[[code/lingua-do-i/etapa5/src/musica-trecho.txt](#)]

Código fonte /etapa5/src/Makefile[[code/lingua-do-i/etapa5/src/Makefile](#)]

6.6 Etapa 6: Fazendo o teste passar

O propósito desta etapa é ver o teste passando com o menor esforço possível.

6.6.1 Fazendo o teste passar com esforço mínimo

Como fazer nosso teste passar com o esforço mínimo? Vamos tentar assim:

Código fonte /etapa6/src/lingua-do-i-core.c[code/lingua-do-i/etapa6/src/lingua-do-i-core.c]

Alterando função para retornar resultado esperado

```
char* TRADUCAO="Minhis vigiis, tidi iqui.";
char* traduzParaLingaDoI(char* mensagemOriginal){
    return TRADUCAO;
}
```

Vamos executar novamente os testes para verificar o teste passando:

Execução dos testes para verificar eles passando

```
$ make clean && make lingua-do-i-test && ./lingua-do-i-test
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o
lingua-do-i-test
```

Ótimo, nosso teste está passando! Na próxima etapa iremos atualizar nossa implementação.

Nota



Talvez você tenha achado esta etapa estranha, e se perguntando *qual o setindo de uma implementação viciada deste jeito?*

Agora que vimos o teste falhar e passar temos confiança de modificar o código, enquanto os testes permanecerem passando nossa implementação estará correta.

6.6.2 Código da etapa

Você pode consultar o código final desta etapa nos seguintes arquivos:

Código fonte /etapa6/src/lingua-do-i.c[code/lingua-do-i/etapa6/src/lingua-do-i.c]

Código fonte /etapa6/src/lingua-do-i-test.c[code/lingua-do-i/etapa6/src/lingua-do-i-test.c]

Código fonte /etapa6/src/lingua-do-i-core.c[code/lingua-do-i/etapa6/src/lingua-do-i-core.c]

Código fonte /etapa6/src/lingua-do-i-core.h[code/lingua-do-i/etapa6/src/lingua-do-i-core.h]

Código fonte /etapa6/src/musica-trecho.txt[code/lingua-do-i/etapa6/src/musica-trecho.txt]

Código fonte /etapa6/src/Makefile[code/lingua-do-i/etapa6/src/Makefile]

6.7 Etapa 7: Implementando troca das vogais

Agora que nossos testes estão passando, precisamos fornecer uma outra implementação de `traduzParaLingaDoI` que seja capaz de traduzir a mensagem e que mantenha nossos testes passando.

6.7.1 Implementando visão geral da tradução

Vamos atualizar a implementação de `traduzParaLinguaDoI` para traduzir a mensagem. A função precisará retornar um novo string, do mesmo tamanho da mensagem original, copiando os caracteres originais da mensagem e só traduzindo as vogais para i:

```
char* traduzParaLinguaDoI(char* mensagemOriginal) {  
    int tamanhoDaMensagem = strlen(mensagemOriginal);❶  
    char* resposta = calloc(1, tamanhoDaMensagem+1);❷  
    for(int i=0; i<tamanhoDaMensagem; i++){❸  
        char caracterDaMensagemOriginal=mensagemOriginal[i];❹  
        resposta[i] = traduzCaracterParaLinguaDoI(  
            caracterDaMensagemOriginal);❺  
    }  
    return resposta;❻  
}
```

- ❶ Consulta o tamanho do string. Para utilização de `strlen()` [171] é necessário incluir `string.h` [171] no início do arquivo.
- ❷ Alocação de espaço para guardar o string do mesmo tamanho da mensagem original.
- ❸, ❹ Percorre todos os caracteres de `mensagemOriginal` e salva em `caracterDaMensagemOriginal` para ser traduzido.
- ❺ Chama função `traduzCaracterParaLinguaDoI` para traduzir o carácter. Em seguida, salva sua tradução na posição correspondente em `resposta`.
- ❻ Retorna string do mesmo tamanho de `mensagemOriginal` com sua tradução.



Dica

Nós poderíamos ter adicionado neste função a lógica de verificar se o carácter é uma vogal, e só invocar a tradução nestes casos. No entanto, parecia responsabilidade demais para uma única função, decidimos atribuí-la a outra função.



Nota

Outra solução poderia ser alterar diretamente as vogais de `mensagemOriginal`, o que certamente pouparia espaço para alocação de memória. Neste caso teríamos que alterar o teste também, pois `MENSAGEM_ORIGINAL` não mais conteria a mensagem original.

6.7.2 Construindo função para tradução de caracteres

Como vimos anteriormente, vamos construir uma função cujo único propósito será traduzir vogais para i e retornar o valor original caso o carácter não seja uma vogal:

Implementando função de tradução de caracteres em `lingua-do-i-core.c`

```
char traduzCaracterParaLinguaDoI(char original) {
    char resposta;
    switch (original) {
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u':
            resposta = 'i';
            break;
        default:
            resposta = original;
    }
    return resposta;
}
```

Com nossa implementação super simples, vamos executar nossos testes com a esperança de que eles estejam passando:

Execução dos testes após nossa implementação

```
$ make clean && make lingua-do-i-test && ./lingua-do-i-test
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o ↵
    lingua-do-i-test
lingua-do-i-test: lingua-do-i-test.c:11: ↵
    verificaConteudosSaoIguais: Assertion `strcmp(conteudo, ↵
    esperado) == 0 && "conteúdo deve ser igual ao esperado" ↵
    failed.
Abortado (imagem do núcleo gravada)
```

Oops! O teste está falhando! Como isto é possível? Nossa implementação é extremamente simples, ele deveria passar!

Em casos de erros que não compreendemos a causa, **depurar** o programa pode ajudar bastante — é exatamente o que faremos na próxima etapa.

6.7.3 Código da etapa

Você pode consultar o código final desta etapa nos seguintes arquivos:

Código fonte /etapa7/src/lingua-do-i.c[code/lingua-do-i/etapa7/src/lingua-do-i.c]

Código fonte /etapa7/src/lingua-do-i-test.c[code/lingua-do-i/etapa7/src/lingua-do-i-test.c]

Código fonte /etapa7/src/lingua-do-i-core.c[code/lingua-do-i/etapa7/src/lingua-do-i-core.c]

Código fonte /etapa7/src/lingua-do-i-core.h[code/lingua-do-i/etapa7/src/lingua-do-i-core.h]

Código fonte /etapa7/src/musica-trecho.txt[code/lingua-do-i/etapa7/src/musica-trecho.txt]

Código fonte /etapa7/src/Makefile[code/lingua-do-i/etapa7/src/Makefile]

6.8 Etapa 8: Depurando a aplicação

Apesar da nossa simples implementação aparentar está correta, por uma razão ainda desconhecida o nosso teste está falhando.

Esta é uma boa oportunidade para **Depurar** a execução do programa. A depuração será abordada em um capítulo a parte: Apêndice A [155]. O processo de depuração possibilita evitar alterar o programa incluindo diversas mensagens com o propósito de compreender a execução do programa, como estas:

```
printf("passou por aqui...")
printf("O valor de var=%d", var)
```



Dica

Apesar de não ser um pré-requisito para continuação deste capítulo, convidamos você a ler o Apêndice A [155] antes de continuar, para conhecer o processo de depuração.

6.8.1 Depuração do programa

A depuração do programa pode ser vista na Seção A.2.4 [158] e a identificação do problema na Seção A.2.5 [160].

6.8.2 Correção realizada

Após a depuração identificamos o problema e corrigimos o erro que estava em nosso arquivo de teste:

Código fonte /etapa8/src/lingua-do-i-test.c[[code/lingua-do-i/etapa8/src/lingua-do-i-test.c](#)]

Correção do texto de TRADUCAO_ESPERADA no teste

```
char* TRADUCAO_ESPERADA= "Minhis vigiis, tidi iqii.";
```

Após a correção do código fonte, executamos o teste novamente:

Re-executando o teste após a correção

```
$ make clean && make && ./lingua-do-i-test
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i.c lingua-do-i-core.c -o lingua-do-i
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o lingua-do-i-test
```

Nosso teste está passando novamente!

6.8.3 Código da etapa

Você pode consultar o código final desta etapa nos seguintes arquivos:

Código fonte /etapa8/src/lingua-do-i.c[[code/lingua-do-i/etapa8/src/lingua-do-i.c](#)]

Código fonte /etapa8/src/lingua-do-i-test.c[code/lingua-do-i/etapa8/src/lingua-do-i-test.c]

Código fonte /etapa8/src/lingua-do-i-core.c[code/lingua-do-i/etapa8/src/lingua-do-i-core.c]

Código fonte /etapa8/src/lingua-do-i-core.h[code/lingua-do-i/etapa8/src/lingua-do-i-core.h]

Código fonte /etapa8/src/musica-trecho.txt[code/lingua-do-i/etapa8/src/musica-trecho.txt]

Código fonte /etapa8/src/Makefile[code/lingua-do-i/etapa8/src/Makefile]

6.9 Etapa 9: Inclusão de novos testes

Após a correção, nossos testes estão passando. Mas será que a implementação de `traduzParaLingaDoI` está correta? Vamos adicionar novos testes para garantir isso.

6.9.1 Adição de testes

```
char* MENSAGEM_ORIGINAL="Minhas vogais, tudo aqui.";❶
char* TRADUCAO_ESPERADA="Minhis vigiis, tidi iqii.";❷
char* MENSAGEM_ORIGINAL2="Oh! Deus, será que o senhor se zangou";❸
char* TRADUCAO_ESPERADA2="Ih! Diis, sirí qii i sinhir si zingii";❹
void testTraducaoParaLingaDoI() {
    char* mensagemTraduzida = traduzParaLingaDoI(MENSAGEM_ORIGINAL);❺
    verificaConteudosSaoIguais(mensagemTraduzida, TRADUCAO_ESPERADA);❻
    verificaConteudosSaoIguais(traduzParaLingaDoI(MENSAGEM_ORIGINAL2), ↵
        TRADUCAO_ESPERADA2);❼
}
```

❶, ❷, ❺, ❻ Techos já existiam no testes anteriormente.

❸, ❹, ❼ Techos adicionados ao teste.

Após adicionar mais uma mensagem para ser traduzida, re-executamos os testes:

Execução dos testes

```
$ make clean && make && ./lingua-do-i-test
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i.c lingua-do-i-core.c -o lingua ↵
-do-i
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o ↵
lingua-do-i-test
lingua-do-i-test: lingua-do-i-test.c:11: ↵
    verificaConteudosSaoIguais: Assertion `strcmp(contenido, ↵
    esperado) == 0 && "conteúdo deve ser igual ao esperado"' ↵
    failed.
Abortado (imagem do núcleo gravada)
```

Como a falha ocorreu devido a inclusão do novo teste, nosso próximo passo será atualizar a implementação para fazer o teste passar.

6.9.2 Atualizar a implementação para fazer o teste passar

A diferença do teste adicionado é que ele adiciona uma vogal maiúscula(O), e outra vogal com acento(â): **Oh! Deus, será...**

Vamos atualizar nossa implementação para possibilitar esta conversão também:

```
char traduzCaracterParaLinguaDoI(char original) {
    char resposta;
    switch (original) {
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u':
            resposta = 'i';
            break;
        case 'A':
        case 'E':
        case 'I':
        case 'O':
        case 'U':
            resposta = 'I'; // ❶
            break;
        case 'á':
        case 'é':
        case 'í':
        case 'ó':
        case 'ú':
            resposta = 'i'; // ❷
            break;
        default:
            resposta = original;
    }
    return resposta;
}
```

- ❶ Inclusão de conversão de letra maiúscula.
- ❷ Conversão de caracteres com acentos.

Após estas modificações vamos compilar e executar os testes novamente:

Compilação e execução dos testes após modificações

```
$ make clean && make && make test_all
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i.c lingua-do-i-core.c -o lingua -do-i
lingua-do-i-core.c: Na função 'traduzCaracterParaLinguaDoI':
lingua-do-i-core.c:64:7: aviso: constante de caractere multi- ←
    caractere [-Wmultichar]
lingua-do-i-core.c:64:2: aviso: case label value exceeds maximum ←
    value for type [habilitado por padrão]
```



```
...
lingua-do-i-core.c:69:14: aviso: constante de caractere multi- ←
    caractere [-Wmultichar]
lingua-do-i-core.c:69:3: aviso: overflow in implicit constant ←
    conversion [-Woverflow]
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o ←
    lingua-do-i-test
lingua-do-i-core.c: Na função 'traduzCaracterParaLinguaDoI':
lingua-do-i-core.c:64:7: aviso: constante de caractere multi- ←
    caractere [-Wmultichar]
lingua-do-i-core.c:64:2: aviso: case label value exceeds maximum ←
    value for type [habilitado por padrão]
...
lingua-do-i-core.c:69:14: aviso: constante de caractere multi- ←
    caractere [-Wmultichar]
lingua-do-i-core.c:69:3: aviso: overflow in implicit constant ←
    conversion [-Woverflow]
./lingua-do-i-test
lingua-do-i-test: lingua-do-i-test.c:11: ←
    verificaConteudosSaoIguais: Assertion 'strcmp(conteudo, ←
    esperado) == 0 && "conteúdo deve ser igual ao esperado"' ←
    failed.
make: *** [test_all] Abortado (arquivo core criado)
```

Quando executamos o teste, verificamos que o compilador apresentou vários alertas e o teste falhou. A falha provavelmente está relacionada aos acentos, vamos compreender melhor na próxima etapa.

6.9.3 Código da etapa

Você pode consultar o código final desta etapa nos seguintes arquivos:

Código fonte /etapa8/src/lingua-do-i.c[code/lingua-do-i/etapa8/src/lingua-do-i.c]

Código fonte /etapa8/src/lingua-do-i-test.c[code/lingua-do-i/etapa8/src/lingua-do-i-test.c]

Código fonte /etapa8/src/lingua-do-i-core.c[code/lingua-do-i/etapa8/src/lingua-do-i-core.c]

Código fonte /etapa8/src/lingua-do-i-core.h[code/lingua-do-i/etapa8/src/lingua-do-i-core.h]

Código fonte /etapa8/src/musica-trecho.txt[code/lingua-do-i/etapa8/src/musica-trecho.txt]

Código fonte /etapa8/src/Makefile[code/lingua-do-i/etapa8/src/Makefile]

6.10 Etapa 10: Tratando texto com acentos

Na etapa anterior percebemos que a utilização de acentos provocou falha no nosso teste.

Para compreender o problema precisamos entender sobre **codificação** de arquivos. Enquanto caracteres ASCII são facilmente codificados, pois cada carácter possui um único byte (que corresponde ao tamanho de um `char`), outras codificações, como UTF-8, podem possuir caracteres com tamanhos diferentes. Além disso, pode existir mais de uma codificação para o mesmo carácter.

Por enquanto nós iremos utilizar o Algoritmo do Avestruz para resolução de problemas: nós iremos ignorar o problema e continuar nossa implementação ignorando o tratamento de acentos. Depois, quando tivermos conhecimento suficiente para tratar este problema poderemos voltar a ele.

**Dica**

Você conhecer mais sobre o **Algoritmo do Avestruz** em http://pt.wikipedia.org/wiki/Algoritmo_do_avestruz.

6.10.1 Desfazer teste com acentos

Para manter nossos testes passando vamos manter o carácter á no string `TRADUCAO_ESPERADA2`:

```
char* MENSAGEM_ORIGINAL2="Oh! Deus, será que o senhor se zangou";
char* TRADUCAO_ESPERADA2="Ih! Diis, sirá qii i sinhir si zingii";❶
```

❶ Substituição do í por á. Por enquanto nosso programa não irá tratar acentos.

6.10.2 Desfazer implementação de core que tratava acentos

Em seguida, precisamos remover de `traduzCaracterParaLinguaDoI` os case com caracteres acentuados:

```
char traduzCaracterParaLinguaDoI(char original) {
    char resposta;

    switch (original) {
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u':
            resposta = 'i';
            break;
        case 'A':
        case 'E':
        case 'I':
        case 'O':
        case 'U':
            resposta = 'I';
            break;
        default:
            resposta = original;
    }

    return resposta;
}
```

Após atualizar a função, vamos re-executar os testes:

Re-execução dos testes após remoção de tratamento de acentos

```
$ make clean && make && make test_all
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i.c lingua-do-i-core.c -o lingua-do-i
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o
lingua-do-i-test
./lingua-do-i-test
```

Ótimo, como esperado, nossos testes estão passando novamente!

6.10.3 Mantendo registro de novas funcionalidades no TODO

Para não esquecer que temos uma pendência no projeto, vamos criar um arquivo como nome `TODO.txt` que manterá registro de atividades que gostaríamos de realizar no futuro:

Código fonte /etapa10/src/lingua-do-i.c[code/lingua-do-i/etapa10/TODO.txt]

Criação do arquivo TODO.txt

Arquivo para manter registro de atividades que gostaríamos de realizar no futuro:

- Conversão de vogais com acento para seus correspondentes na lingua do i
Exemplos: á -> í, õ -> ã, à -> ì etc.



Dica

A criação de um arquivo **TODO** possibilita comunicar para as demais pessoas (envolvidas ou que irão se envolver no projeto) onde elas podem contribuir para o projeto.

6.10.4 Código da etapa

Você pode consultar o código final desta etapa nos seguintes arquivos:

Código fonte /etapa10/src/lingua-do-i.c[code/lingua-do-i/etapa10/src/lingua-do-i.c]

Código fonte /etapa10/src/lingua-do-i-test.c[code/lingua-do-i/etapa10/src/lingua-do-i-test.c]

Código fonte /etapa10/src/lingua-do-i-core.c[code/lingua-do-i/etapa10/src/lingua-do-i-core.c]

Código fonte /etapa10/src/lingua-do-i-core.h[code/lingua-do-i/etapa10/src/lingua-do-i-core.h]

Código fonte /etapa10/src/musica-trecho.txt[code/lingua-do-i/etapa10/src/musica-trecho.txt]

Código fonte /etapa10/src/Makefile[code/lingua-do-i/etapa10/src/Makefile]

6.11 Etapa 11: Salvando conteúdo em arquivo

Além de traduzir mensagens nosso programa também precisa salvar seu conteúdo em arquivo. Nesta etapa vamos atualizar nossa implementação para possibilitar o salvamento.

6.11.1 Adicionar teste de salvamento de conteúdo

Mas uma vez, antes de iniciar a implementação da nossa funcionalidade iremos escrever seu teste:

```
char* CONTEUDO_QUALQUER = "abracadabra";
void testSalvaConteudoEmArquivo() {
    FILE * arquivoDestino = criaArquivoTemporario(); ❶
    salvaConteudo(arquivoDestino, CONTEUDO_QUALQUER); ❷
    verificaConteudoFoiSalvo(arquivoDestino, CONTEUDO_QUALQUER); ❸

    fclose(arquivoDestino); ❹
}
```

- ❶ Criação de um arquivo temporário para o teste, o conteúdo do teste será escrito nele. Se estivéssemos utilizando um arquivo normal, deveríamos nos certificar de que o arquivo não existia antes, em decorrência de execuções anteriores. Arquivos temporários são bons para os testes pois quando fechados são automaticamente excluídos.
- ❷ Esta função será implementada no core, será responsável por salvar um string em um FILE.
- ❸ Função para verificar que o conteúdo escrito no arquivo corresponde ao conteúdo passado como parâmetro (CONTEUDO_QUALQUER).
- ❹ Fechamento do arquivo temporário.

Percebam que estamos definindo que a função `salvaConteudo` irá receber um `FILE*` em vez de um `char*`, correspondente ao nome do arquivo, diferente de como fizemos com a função `lerConteudoDoArquivo` (no Primeiro teste [101]). Isto foi intencional, pois pretendemos, no futuro, salvar o conteúdo também na saída padrão (que é do tipo `FILE*`).

6.11.2 Função para criar arquivo temporário

Precisamos criar a função responsável pela criação do arquivo temporário que foi definida no teste:

Função para criar arquivos temporários

```
FILE * criaArquivoTemporario() {
    return tmpfile(); ❶
}
```

- ❶ Neste função apenas invocamos a função `tmpfile()`.



Nota

Decidimos criar a função `criaArquivoTemporario` apenas para aumentar a legibilidade do código para os leitores em português. Na prática, não haveria a necessidade de criá-la.

6.11.3 Verificando conteúdo salvo no arquivo

A outra função, para verificar que o conteúdo salvo no arquivo corresponde ao que foi escrito nele, foi implementada como a seguir:

Função para verificar conteúdo salvo em arquivo

```
void verificaConteudoFoiSalvo( FILE* arquivo,
    char* conteudo_esperado ){
    char* conteudo = lerConteudoDeArquivoArberto(arquivo); ❶
    assert( strcmp(conteudo, conteudo_esperado) == 0
        && "Conteúdo do arquivo não corresponde ao esperado"); ❷
}
```

- ❶ A função `lerConteudoDeArquivoArberto` foi implementada em `lingua-do-i-core.c`, mas ainda não está visível aqui, precisaremos adicioná-la ao cabeçalho mais adiante.
- ❷ Código similar a `verificaConteudosSaoIguais`, mas com outra mensagem.

Percebam que na segunda linha, poderíamos ter invocado `verificaConteudosSaoIguais(conteudo, conteudo_esperado)`, mas quando esta função fosse falhar receberíamos a mesma mensagem das falhas anteriores, então optamos por exibir uma nova mensagem: “Conteúdo do arquivo não corresponde ao esperado”.

6.11.4 Atualização do arquivo de cabeçalho do core

Sabemos que temos algumas alterações pendentes para serem realizadas no arquivo de cabeçalho do core, vamos realizá-las agora:

Atualização de `lingua-do-i-core.h`

```
void salvaConteudo(FILE* arquivoDestino, char* conteudo); ❶
char* lerConteudoDeArquivoArberto(FILE* arquivo); ❷
```

- ❶ Principal função do teste, que desejamos implementar no core.
- ❷ Declaração de função já implementada em core mas que não estava visível externamente ao arquivo.

6.11.5 Implementação vazia para possibilitar teste falhar

Antes de compilar e executar o teste, precisamos ainda criar uma implementação vazia de `salvaConteudo`:

Implementação vazia em `lingua-do-i-core.c`

```
void salvaConteudo(FILE* arquivoDestino, char* conteudo){
}
```

6.11.6 Verificando o teste falhando

Com a implementação da função realizada, vamos compilar os arquivos e executar os testes:

Compilação e execução dos testes

```
$ make clean && make && ./lingua-do-i-test
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i.c lingua-do-i-core.c -o lingua-do-i
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o lingua-do-i-test
lingua-do-i-test: lingua-do-i-test.c:28: verificaConteudoFoiSalvo:
Assertion `strcmp(conteudo, conteudo_esperado) == 0 && "Conteúdo do
arquivo não corresponde ao esperado"' failed.
Abortado (imagem do núcleo gravada)
```

Como esperado, o teste falhou exatamente porque a função que deveria escrever o conteúdo está vazia. Repare que a mensagem da falha corresponde à mesma da verificação que adicionamos nesta etapa.

6.11.7 Fazendo o teste passar

Desta vez não há uma forma simples de fazer o teste passar, precisamos realmente escrever o conteúdo no arquivo para que o teste passe.

Implementamos a função `salvaConteudo` da seguinte forma:

Implementação para escrever conteúdo no arquivo

```
void salvaConteudo(FILE* arquivoDestino, char* conteudo) {
    fprintf(arquivoDestino, "%s", conteudo);
}
```



Dica

A função `fprintf()` [176] é similar a `printf()` [176], a única diferença é que `printf` escreve sempre na saída padrão, enquanto que `fprintf` recebe como parâmetro qual o arquivo (`FILE*`) onde será escrito o conteúdo.

Após a implementação, vamos verificar os testes:

Execução dos testes

```
$ make clean && make && ./lingua-do-i-test
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i.c lingua-do-i-core.c -o lingua-do-i
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o lingua-do-i-test
```

Ótimo, nossos testes estão passando novamente!

6.11.8 Código da etapa

Você pode consultar o código final desta etapa nos seguintes arquivos:

Código fonte /etapa11/src/lingua-do-i.c[code/lingua-do-i/etapa11/src/lingua-do-i.c]

Código fonte /etapa11/src/lingua-do-i-test.c[code/lingua-do-i/etapa11/src/lingua-do-i-test.c]

Código fonte /etapa11/src/lingua-do-i-core.c[code/lingua-do-i/etapa11/src/lingua-do-i-core.c]

Código fonte /etapa11/src/lingua-do-i-core.h[code/lingua-do-i/etapa11/src/lingua-do-i-core.h]

Código fonte /etapa11/src/musica-trecho.txt[code/lingua-do-i/etapa11/src/musica-trecho.txt]

Código fonte /etapa11/src/Makefile[code/lingua-do-i/etapa11/src/Makefile]

6.12 Etapa 12: Determinando entrada do aplicativo

Nesta etapa vamos definir como o programa irá determinar qual a entrada será processada, se utilizará a entrada padrão ou a partir de um arquivo.

Nesta funcionalidade, caso o aplicativo seja invocado sem nenhum parâmetro, ele irá ler da entrada padrão, caso seja especificado o nome de um arquivo, ele será utilizado.

Invocando sem parâmetro: ler da entrada padrão

```
lingua-do-i
```

Invocando com parâmetro: ler do arquivo

```
lingua-do-i meu-arquivo-de-entrada.txt
```



Importante

Lembre-se que estamos construindo uma aplicação, embora estivemos implementando vários testes, aos poucos as funcionalidades do nosso programa estão sendo implementadas em `lingua-do-i-core.c`. Mais adiante iremos alterar `lingua-do-i.c` para invocar as funções do core.

6.12.1 Testes para especificar a entrada

No código a seguir estão definidos 3 testes, que utilizam os parâmetros passados pelo sistema operacional à função `main()` [169], para determinar a entrada:

Testes para determinar a entrada do programa

```
int ARGV_SEM_PARAMETROS = 1;
const char* ARGV_SEM_PARAMETROS[] = {"lingua-do-i"};
void testDefinirEntradaPadrao() {
    FILE* entrada = determinaEntrada(ARGV_SEM_PARAMETROS,
    ARGV_SEM_PARAMETROS);
    verificaEntradaFoiEntradaPadrao(entrada);
}
```

```
int ARGV_COM_1_PARAMETRO = 2;
const char* ARGV_ARQUIVO_VALIDO[] =
    {"lingua-do-i", "musica-trecho.txt"};
const char* ARGV_ARQUIVO_INEXISTENTE[] =
    {"lingua-do-i", "inexistente.xyz"};
void testDefinirEntradaDeArquivo() {
    FILE* entrada = determinaEntrada(ARGV_COM_1_PARAMETRO,
        ARGV_ARQUIVO_VALIDO);
    verificaEntradaFoiArquivoValido(entrada);
}

void testDefinirEntradaDeArquivoInexistente() {
    FILE* entrada = determinaEntrada(ARGV_COM_1_PARAMETRO,
        ARGV_ARQUIVO_INEXISTENTE);
    verificaEntradaFoiInvalida(entrada);
}
```

Os três testes estão invocando a função `determinaEntrada`, que ainda será implementada no core, mas que terá a responsabilidade de determinar a entrada do aplicativo a partir dos parâmetros passados à função `main()` [169]:

testDefinirEntradaPadrao

Neste teste estamos definindo que se o programa for executado sem nenhum parâmetro, ele irá ler da entrada padrão.

testDefinirEntradaDeArquivo

Caso o programa seja invocado passando o nome do arquivo, ele será utilizado como entrada.

testDefinirEntradaDeArquivoInexistente

Caso seja passando o nome do arquivo, mas ele não exista, então o programa não poderá processar a entrada.



Dica

Perceba como os nomes das funções facilitam na compreensão do código. É preferível que você crie várias funções pequenas com responsabilidades bem definidas do que uma poucas funções com muitas responsabilidades.

Por último, também é necessário invocá-los a partir do `main`:

Invocando os testes a partir do main

```
int main(void) {
    ...
    testDefinirEntradaPadrao();
    testDefinirEntradaDeArquivo();
    testDefinirEntradaDeArquivoInexistente();

    return EXIT_SUCCESS;
}
```


6.12.2 Fazendo os testes falharem

Uma vez definido os testes, nosso próximo passo é fazê-los compilarem e falharem.

Para fazer os testes compilarem precisamos da ajuda do compilador, utilizaremos as notificações de problemas de compilação para descobrir o próximo passo:

Invocando o compilador para descobrir o próximo passo

```
$ make clean && make && make test_all
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i.c lingua-do-i-core.c -o lingua-do-i
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o lingua-do-i-test
lingua-do-i-test.c: Na função 'testDefinirEntradaPadrao':
lingua-do-i-test.c:54:2: aviso: implicit declaration of function 'determinaEntrada' [-Wimplicit-function-declaration]
...
```

O compilador nos informou que a função `determinaEntrada` ainda não existe, precisamos defini-la no cabeçalho e criá-la no core:

Adicionando `determinaEntrada` em `lingua-do-i-core.h`

```
FILE* determinaEntrada(int argc, const char* argv[]);
```

Em seguida precisamos criar a função no core:

Implementando `determinaEntrada` no core

```
FILE* determinaEntrada(int argc, const char* argv[]){
    return NULL;
}
```

Vamos invocar a compilação novamente para ter uma sugestão do próximo passo:

```
$ make clean && make && make test_all
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i.c lingua-do-i-core.c -o lingua-do-i
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o lingua-do-i-test
lingua-do-i-test.c: Na função 'testDefinirEntradaPadrao':
lingua-do-i-test.c:56:2: aviso: implicit declaration of function 'verificaEntradaFoiEntradaPadrao' [-Wimplicit-function-declaration]
...
```

Descobrimos que o próximo passo é criar a função `verificaEntradaFoiEntradaPadrao`:

```
void verificaEntradaFoiEntradaPadrao(FILE* entrada){
    assert(entrada == stdin &&
           "Entrada deveria ser Entrada Padrão");
}
```

Compilando novamente para descobrir o próximo passo:

```
$ make clean && make && make test_all
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i.c lingua-do-i-core.c -o lingua-do-i
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o lingua-do-i-test
lingua-do-i-test.c: Na função 'testDefinirEntradaDeArquivo':
lingua-do-i-test.c:69:2: aviso: implicit declaration of function 'verificaEntradaFoiArquivoValido' [-Wimplicit-function-declaration]
...
```

O erro anterior foi corrigido, agora o compilador nos informa que precisamos implementar a função `verificaEntradaFoiArquivoValido`:

Implementação de `verificaEntradaFoiArquivoValido` no teste

```
void verificaEntradaFoiArquivoValido(FILE* entrada){
    assert(entrada != NULL && entrada!= stdin &&
           "Entrada deveria ser um arquivo válido");❶
}
```

Mais uma vez, compilando para descobrir o próximo passo:

```
$ make clean && make && make test_all
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i.c lingua-do-i-core.c -o lingua-do-i
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o lingua-do-i-test
lingua-do-i-test.c: Na função
'testDefinirEntradaDeArquivoInexistente':
lingua-do-i-test.c:80:2: aviso: implicit declaration of function 'verificaEntradaFoiInvalida' [-Wimplicit-function-declaration]
...
```

Novamente, precisamos implementar a função `verificaEntradaFoiInvalida`:

```
void verificaEntradaFoiInvalida(FILE* entrada){
    assert(entrada == NULL && "Entrada deve ser inválida");
}
```

Compilando para descobrir o próximo passo:

```
$ make clean && make && make test_all
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i.c lingua-do-i-core.c -o lingua-do-i
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o lingua-do-i-test
```

```
./lingua-do-i-test
lingua-do-i-test: lingua-do-i-test.c:52: ↵
    verificaEntradaFoiEntradaPadrao: Assertion `entrada == stdin ↵
    && "Entrada deveria ser Entrada Padrão"' failed.
make: *** [test_all] Abortado (arquivo core criado)
...
```

Ótimo! O código já está compilando e nosso primeiro teste está falhando.

6.12.3 Certificando-se que todos os novos testes estão falhando

Para verificar os outros testes falhando, basta comentar a linha de invocação do teste na função main:

Comentando teste para verificar que os demais estão falhando

```
int main(void) {
    testLerConteudoDoArquivo();
    testSalvaConteudoEmArquivo();
    testTraducaoParaLinguaDoI();

    //testDefinirEntradaPadrao();
    testDefinirEntradaDeArquivo();
    testDefinirEntradaDeArquivoInexistente();

    return EXIT_SUCCESS;
}
```

Verificando o próximo teste falhando:

```
$ make clean && make && make test_all
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i.c lingua-do-i-core.c -o lingua ↵
-do-i
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o ↵
lingua-do-i-test
./lingua-do-i-test
lingua-do-i-test: lingua-do-i-test.c:57: ↵
    verificaEntradaFoiArquivoValido: Assertion `entrada != ((void ↵
    *)0) && entrada!= stdin && "Entrada deveria ser um arquivo ↵
    válido"' failed.
make: *** [test_all] Abortado (arquivo core criado)
```

Mais um teste falhando, falta apenas verificar o último.

```
int main(void) {
    testLerConteudoDoArquivo();
    testSalvaConteudoEmArquivo();
    testTraducaoParaLinguaDoI();

    //testDefinirEntradaPadrao();
    //testDefinirEntradaDeArquivo();
    testDefinirEntradaDeArquivoInexistente();
}
```

```
    return EXIT_SUCCESS;
}
```

Tentando verificar o último teste falhando:

```
$ make clean && make && make test_all
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i.c lingua-do-i-core.c -o lingua ↵
    -do-i
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o ↵
    lingua-do-i-test
./lingua-do-i-test
```

O último teste não está falhando. Verificar que o teste está falhando é uma etapa muito importante², vamos modificar a implementação de `determinaEntrada` para garantir isso:

Modificação em core para todos os testes falharem

```
FILE* determinaEntrada(int argc, const char* argv[]){
    FILE* resultadoParaFalhar;
    if (argc == 1){
        resultadoParaFalhar = stdout;
    }else{
        resultadoParaFalhar = stdin;
    }
    return resultadoParaFalhar;
}
```

Vamos re-executar o teste:

```
$ make clean && make && make test_all
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i.c lingua-do-i-core.c -o lingua ↵
    -do-i
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o ↵
    lingua-do-i-test
./lingua-do-i-test
lingua-do-i-test: lingua-do-i-test.c:62: ↵
    verificaEntradaFoiInvalida: Assertion 'entrada == ((void *)0) ↵
    && "Entrada deve ser inválida"' failed.
make: *** [test_all] Abortado (arquivo core criado)
```

Após esta modificação todos os testes falham, estamos prontos para avançar para a próxima etapa.



Nota

Perceba que todas as mensagens de falha dos testes foram diferentes, para ficar evidente a causa da falha.

²Para saber mais sobre os testes falhando consulte <http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd>.

6.12.4 Código da etapa

Você pode consultar o código final desta etapa nos seguintes arquivos:

Código fonte /etapa12/src/lingua-do-i.c[code/lingua-do-i/etapa12/src/lingua-do-i.c]

Código fonte /etapa12/src/lingua-do-i-test.c[code/lingua-do-i/etapa12/src/lingua-do-i-test.c]

Código fonte /etapa12/src/lingua-do-i-core.c[code/lingua-do-i/etapa12/src/lingua-do-i-core.c]

Código fonte /etapa12/src/lingua-do-i-core.h[code/lingua-do-i/etapa12/src/lingua-do-i-core.h]

Código fonte /etapa12/src/musica-trecho.txt[code/lingua-do-i/etapa12/src/musica-trecho.txt]

Código fonte /etapa12/src/Makefile[code/lingua-do-i/etapa12/src/Makefile]

6.13 Etapa 13: Fazendo os testes de entrada passarem

6.13.1 Fazendo os testes passarem

Antes de fazer o primeiro teste passar, vamos verificar o teste falhando novamente:

Verificando teste da entrada padrão falhando

```
$ make clean && make lingua-do-i-test && make test_all
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o ↵
    lingua-do-i-test
./lingua-do-i-test
lingua-do-i-test: lingua-do-i-test.c:53: ↵
    verificaEntradaFoiEntradaPadrao: Assertion `entrada == stdin ↵
    && "Entrada deveria ser Entrada Padrão"' failed.
make: *** [test_all] Abortado (arquivo core criado)
```

Para o teste passar, precisamos retornar `stdin` quando o aplicativo for invocado sem parâmetros:

```
FILE* determinaEntrada(int argc, const char* argv[]){
    FILE* entrada;
    if (argc == 1){
        entrada = stdin; ❶
    }else{
        entrada = stdin;
    }
    return entrada;
}
```

❶, ❶ Quando `argc == 1`, então não há nenhum parâmetro e devemos retornar a entrada padrão.

Após esta modificação, vamos executar os testes novamente:

Re-execução dos testes

```
$ make clean && make lingua-do-i-test && make test_all
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o ↵
    lingua-do-i-test
./lingua-do-i-test
lingua-do-i-test: lingua-do-i-test.c:58: ↵
    verificaEntradaFoiArquivoValido: Assertion `entrada != ((void ↵
        *)0) && entrada!= stdin && "Entrada deveria ser um arquivo ↵
        válido"' failed.
make: *** [test_all] Abortado (arquivo core criado)
```

Ótimo, nosso primeiro teste passou, e a falha do segundo foi apresentada.

O próximo passo será fazer o segundo teste passar. Este teste espera que caso seja passado o nome do arquivo como parâmetro, então a entrada deve utilizá-lo para leitura:

Ajuste para fazer o segundo teste passar

```
FILE* determinaEntrada(int argc, const char* argv[]){
    FILE* entrada;
    if (argc == 1){
        entrada = stdin;
    }else{
        entrada = fopen(argv[1], "r"); ❶
    }
    return entrada;
}
```

- ❶ Utiliza `fopen()` [172] para abrir arquivo para leitura, utilizando o nome do arquivo passado como parâmetro.

Após esta modificação vamos re-executar os testes:

```
$ make clean && make lingua-do-i-test && make test_all
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o ↵
    lingua-do-i-test
./lingua-do-i-test
```

Com esta última modificação todos os nossos testes estão passando!

Na próxima etapa iremos unir a implementação de `core` no `main()` para gerar o aplicativo capaz de traduzir para língua do `i`.

6.13.2 Código da etapa

Você pode consultar o código final desta etapa nos seguintes arquivos:

Código fonte /etapa13/src/lingua-do-i.c[[code/lingua-do-i/etapa13/src/lingua-do-i.c](#)]

Código fonte /etapa13/src/lingua-do-i-test.c[[code/lingua-do-i/etapa13/src/lingua-do-i-test.c](#)]

Código fonte /etapa13/src/lingua-do-i-core.c[[code/lingua-do-i/etapa13/src/lingua-do-i-core.c](#)]

Código fonte /etapa13/src/lingua-do-i-core.h[[code/lingua-do-i/etapa13/src/lingua-do-i-core.h](#)]

Código fonte /etapa13/src/musica-trecho.txt[[code/lingua-do-i/etapa13/src/musica-trecho.txt](#)]

Código fonte /etapa13/src/Makefile[[code/lingua-do-i/etapa13/src/Makefile](#)]

6.14 Etapa 14: Implementando aplicação da lingua-do-i

6.14.1 Construindo o main

Agora que já possuímos várias funções implementadas vamos incorporá-las ao `main` da aplicação.

```
#include <stdio.h>
#include <stdlib.h>
#include "lingua-do-i-core.h" ❶

int main(int argc, const char* argv[]) {❷

    FILE* entrada = determinaEntrada(argc, argv);

    if (entrada) {❸
        char* conteudo=lerConteudoDeArquivoArberto(entrada);❹
        char* mensagem = traduzParaLingaDoI(conteudo);❺
        salvaConteudo(stdout, mensagem);❻
    } else {
        fprintf(stderr, "Problema ao abrir arquivo: %s\n",
                argv[1]);❼
        exit(EXIT_FAILURE);❽
    }

    return EXIT_SUCCESS;
}
```

- ❶ Nosso primeiro passo é incluir o cabeçalho de `core`, para poder utilizar as funções definidas lá.
- ❷ Atualizamos o cabeçalho do `main`, adicionando os parâmetros que serão atribuídos pelo sistema operacional ao invocar a aplicação.
- ❸ Em C é comum a checagem de parâmetros desta forma, caso o parâmetro seja diferente de zero ou `NULL`, então é o valor válido.
- ❹, ❺ Ler conteúdo da entrada e traduz para língua do `i`
- ❻ Salva a mensagem traduzida na saída padrão. O usuário poderá redirecionar para um arquivo se desejar.
- ❼, ❽ Caso o arquivo não exista, a aplicação finaliza com mensagem na **saída de erro**.

A seguir você pode conferir a compilação com sucesso da aplicação:

Compilação de `lingua-do-i`

```
$ make clean && make && make test_all
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i.c lingua-do-i-core.c -o lingua-do-i
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o lingua-do-i-test
./lingua-do-i-test
```

6.14.2 Executando lingua-do-i

Primeiro vamos testar a tradução lendo de um arquivo passado como parâmetro:

Lendo de arquivo passado como parâmetro

```
$ ./lingua-do-i musica-trecho.txt
Ih! Diis, pirdii isti pibri ciitidi
```

Em seguida, vamos testar ler da entrada padrão, direcionada de um arquivo:

```
$ ./lingua-do-i < musica-trecho.txt
Ih! Diis, pirdii isti pibri ciitidi
```

Ainda faltamos testar se foi passado um arquivo que não existe:

```
$ ./lingua-do-i arquivo-que-nao-existe.txt
Problema ao abrir arquivo: arquivo-que-nao-existe.txt
```

Nosso aplicativo foi capaz de ler de um arquivo, passado como parâmetro ou sendo direcionado pela entrada padrão. Mas será que ele é capaz de ler a partir da entrada padrão sem ser direcionado de um arquivo? Vamos fazer o teste!

Tentando ler da entrada padrão:

```
$ cat musica-trecho.txt | ./lingua-do-i
```



Nota

O comando `cat`, disponível no Linux, ler o conteúdo de um arquivo e o imprime na saída padrão. Ele é equivalente ao comando `type` no Windows.

O resultado esperado não foi satisfatório, ele deveria ser capaz de ler da entrada padrão, o que está errado com a nossa implementação? Vamos analisá-la na próxima etapa.

6.14.3 Código da etapa

Você pode consultar o código final desta etapa nos seguintes arquivos:

Código fonte /etapa14/src/lingua-do-i.c[[code/lingua-do-i/etapa14/src/lingua-do-i.c](#)]

Código fonte /etapa14/src/lingua-do-i-test.c[[code/lingua-do-i/etapa14/src/lingua-do-i-test.c](#)]

Código fonte /etapa14/src/lingua-do-i-core.c[[code/lingua-do-i/etapa14/src/lingua-do-i-core.c](#)]

Código fonte /etapa14/src/lingua-do-i-core.h[[code/lingua-do-i/etapa14/src/lingua-do-i-core.h](#)]

Código fonte /etapa14/src/musica-trecho.txt[[code/lingua-do-i/etapa14/src/musica-trecho.txt](#)]

Código fonte /etapa14/src/Makefile[[code/lingua-do-i/etapa14/src/Makefile](#)]

6.15 Etapa 15: Processando entrada por fluxo

Nossa aplicação não foi capaz de traduzir uma mensagem ao recebê-la por um fluxo na entrada padrão.

Nós sabemos que a Função lerTamanhoDoArquivo [113] vai ao final do arquivo, depois retorna para o início. Mas isto não é possível quando estamos processando fluxos.

Para possibilitar traduzir através de fluxos, vamos iniciar escrevendo um teste para isso.

6.15.1 Incluindo teste para tradução de fluxo

Neste teste, estamos desejando implementar uma função `traduzFluxoDeEntradaNaSaida` que irá traduzir a mensagem de entrada e salvar sua tradução na saída.

Teste de tradução de fluxo

```
char* MUSICA_COMPLETA = "musica-completa.txt";
char* MUSICA_COMPLETA_TRAUDIZADA = "musica-completa-traduzida.txt";
void testTraduzFluxoDeEntrada() {
    FILE* entrada = fopen(MUSICA_COMPLETA, "r");
    FILE* saida = tmpfile();
    traduzFluxoDeEntradaNaSaida(entrada, saida);

    verificaMusicaFoiTraduzidaCorretamenteNaSaida(saida,
        MUSICA_COMPLETA_TRAUDIZADA);

    fclose(entrada);
    fclose(saida);
}
```

Para garantir a tradução correta, criamos um arquivo temporário para servir como saída da função, em seguida, `verificaMusicaFoiTraduzidaCorretamenteNaSaida` irá ler o conteúdo escrito em saída e comparar com o conteúdo do arquivo `MUSICA_COMPLETA_TRAUDIZADA`.

A função para verificar a tradução foi implementada da seguinte forma:

```
void verificaMusicaFoiTraduzidaCorretamenteNaSaida(
    FILE* fluxo, char* arquivoComTraducaoCorreta) {

    rewind(fluxo); // volta para ler do início do fluxo
    char* conteudo=lerConteudoDeArquivoArberto(fluxo);
    char* traducao=lerConteudoDoArquivo(arquivoComTraducaoCorreta);

    assert( strcmp(conteudo, traducao) == 0
        && "Arquivo deve ser traduzido corretamente");
}
```

6.15.2 Salvando a letra da música e sua tradução

Para execução do teste também é preciso criar os arquivos com a música e a tradução:

Música completa

```
Oh! Deus, perdoe este pobre coitado
Que de joelhos rezou um bocado
Pedindo pra chuva cair sem parar

(...)
```

```
Desculpe eu pedir a toda hora pra chegar o inverno
Desculpe eu pedir para acabar com o inferno
Que sempre queimou o meu Ceará
```

Música traduzida

```
Ih! Diis, pirdii isti pibri ciitidi
Qii di jiilhis rizii im bicidi
Pidindi pri chivi ciir sim pirir

(...)
```

```
Discilpi ii pidir i tidi hiri pri chigir i invirni
Discilpi ii pidir piri icibir cim i infirni
Qii simpri qiiimii i mii Ciirá
```



Nota

Ainda não estamos tratando vogais acentuadas, portanto o arquivo traduzido ainda possui as vogais acentuadas.

6.15.3 Fazendo o teste falhar

Vamos invocar o compilador diversas vezes para descobrir os próximos passos até o teste falhar:

Invocando o compilador

```
$ make clean && make lingua-do-i-test && make test_all
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o ↵
    lingua-do-i-test
lingua-do-i-test.c: Na função 'testTraduzFluxoDeEntrada':
lingua-do-i-test.c:107:2: aviso: implicit declaration of ↵
    function 'traduzFluxoDeEntradaNaSaida' [-Wimplicit-function- ↵
    declaration]
/tmp/ccPNDB7P.o: In function 'testTraduzFluxoDeEntrada':
/home/santana/livro/capitulos/code/lingua-do-i/etapa15/src/ ↵
    lingua-do-i-test.c:107: undefined reference to ` ↵
    traduzFluxoDeEntradaNaSaida'
```

```
collect2: ld returned 1 exit status
make: ** [lingua-do-i-test] Erro 1
```

O compilador nos avisou que precisamos criar a função `traduzFluxoDeEntradaNaSaida`:

Inclusão de `traduzFluxoDeEntradaNaSaida` no cabeçalho

```
void traduzFluxoDeEntradaNaSaida(FILE* entrada, FILE* saida);
```

Criando função `traduzFluxoDeEntradaNaSaida` vazia no core

```
void traduzFluxoDeEntradaNaSaida(FILE* entrada, FILE* saida){
}
```

Compilando e executando os testes

```
$ make clean && make lingua-do-i-test && make test_all
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o ↵
    lingua-do-i-test
./lingua-do-i-test
lingua-do-i-test: lingua-do-i-test.c:98: ↵
    verificaMusicaFoiTraduzidaCorretamenteNaSaida: Assertion `↵
    strcmp(conteudo, traducao) == 0 && "Arquivo deve ser ↵
    traduzido corretamente"' failed.
make: *** [test_all] Abortado (arquivo core criado)
```

Ótimo, nosso teste está falhando!

6.15.4 Fazendo o teste passar

Uma vez que nosso teste está falhando, o próximo passo é fazê-lo passar, com o esforço mínimo. Para isso, vamos atualizar a função `traduzFluxoDeEntradaNaSaida` para escrever na saída o valor que esperamos:

Fazendo o teste passar com esforço mínimo

```
void traduzFluxoDeEntradaNaSaida(FILE* entrada, FILE* saida){
    char* traducao = lerConteudoDoArquivo("musica-completa-traduzida.txt ↵
    ");
    int tamanhoDaTraducao = strlen(traducao);
    fwrite(traducao, 1, tamanhoDaTraducao, saida);
}
```

Esta implementação é temporária, servirá apenas para verificarmos o teste passando:

Verificação do teste passando

```
$ make clean && make lingua-do-i-test && make test_all
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o ↵
    lingua-do-i-test
./lingua-do-i-test
```

Ótimo! Nossos testes estão passando novamente. Agora precisamos substituir a implementação da função por outra que continue passando.

6.15.5 Implementando tradução por fluxo

Quando estamos tratando um fluxo, precisamos lê-lo por partes. Só podemos ler uma outra parte após tratar a anterior. Em nossa implementação, decidimos dividir o fluxo por linhas:

Tradução do fluxo por linhas

```
void traduzFluxoDeEntradaNaSaida(FILE* entrada, FILE* saida){
    bool chegouNoFinalDoArquivo = false;

    while (!chegouNoFinalDoArquivo){
        char* linha = lerLinhaDaEntrada(entrada);
        if (tentouLerAposFinalDoArquivo(linha)) {
            chegouNoFinalDoArquivo = true;
        } else {
            char* traducao = traduzParaLinhaDoI(linha);
            salvaConteudoNaSaida(traducao, saida);
        }
    }
}
```

A lógica da função `traduzFluxoDeEntradaNaSaida` consiste em ler a entrada por linhas. A função entra em loop enquanto não chegou ao final do arquivo: lendo uma linha da entrada por vez, traduz e em seguida salva a tradução na saída. A condição de parada do loop só ocorre quando última linha foi traduzida e o final do arquivo foi atingido.

A condição de parada é implementada através da variável `chegouNoFinalDoArquivo`, que inicia com valor `false` e só será modificada quando `tentouLerAposFinalDoArquivo` retornar verdadeiro.

A função `lerLinhaDaEntrada` será responsável por ler do fluxo de entrada uma linha. E `salvaConteudoNaSaida` será responsável por salvar a tradução na saída.

Estas três funções ainda não existem, precisaremos implementá-las a seguir.

6.15.5.1 Lendo conteúdo da entrada com `fgets`

Vamos iniciar a implementação com a função `lerLinhaDaEntrada`, responsável por ler uma linha da entrada:

```
int TAMANHO_MAXIMO_DA_LINHA = 2048; ❶
char* lerLinhaDaEntrada(FILE* entrada){
    char* linha = calloc(1, TAMANHO_MAXIMO_DA_LINHA); ❷
    return fgets(linha, TAMANHO_MAXIMO_DA_LINHA, entrada); ❸
}
```

- ❶ Definindo tamanho máximo de leitura de linha (buffer).
- ❷ Alocando espaço suficiente para caber uma linha.
- ❸ Lendo da entrada e salvando o conteúdo lido no buffer (linha).

Para compreender esta função é preciso entender que `fgets()` [174] irá ler até:

1. Encontrar o fim de linha,
2. Ou até chegar ao final do fluxo
3. Ou até ler a quantidade de caracteres referenciada por `TAMANHO_MAXIMO_DA_LINHA`.

6.15.5.2 Sobre as funções `fgets` e `feof`

O propósito da função `feof()` [180] é indicar se em leituras anteriores o final do arquivo foi *atingido*. Para que isto ocorra, as funções de leitura, como `fgets`, devem registrar na estrutura `FILE` quando detectarem o final do fluxo.

A função `fgets()` [174] só poderá indicar que chegou ao final do arquivo quando ela *atingir* o final do arquivo. A Figura 6.1 [142] ilustra o funcionamento da função `fgets`, estamos evidenciando dois casos:

Conteúdo do arquivo	Na 3ª chamada a <code>fgets</code>		Na 4ª chamada a <code>fgets</code>	
	Retorno	Encontrou final de arquivo	Retorno	Encontrou final de arquivo
Arquivo terminando\n sem final\n de linha EOF	"de linha"	Sim	NULL	Sim
Arquivo terminando\n com final\n de linha\n EOF	"de linha\n"	Não	NULL	Sim

Figura 6.1: Ilustração de leitura com `fgets`

Quando arquivo termina sem final de linha

Neste primeiro caso (ver Figura 6.1 [142]), na terceira chamada de `fgets` os bytes são lidos até quando tentou ler o próximo byte e identificou que chegou no final do arquivo. Neste momento a função irá retornar todos os bytes lidos e registrar que o final do arquivo foi *atingido*. Chamadas subsequentes a `feof` saberão que o final do arquivo foi encontrado.

Quando arquivo termina com final de linha

No segundo caso, a terceira chamada de `fgets` retornará todos os caracteres da última linha, inclusive o `\n`. Ao encontrar o final de linha (`\n`) a leitura da linha é finalizada. Não houve nenhum registro de que se chegou ao final do arquivo. Somente na quarta leitura que a função `fgets` irá perceber que chegou ao final do arquivo e então irá registrar o ocorrido. Apenas na 4ª chamada à `fgets` que seu retorno será `NULL`, indicando que não houve nenhum conteúdo lido.

6.15.5.3 Condição de parada tentouLerAposFinalDoArquivo

Para garantir que todas as linhas lidas sejam traduzidas, não basta ler até atingir o final do arquivo, precisamos nos certificar que mesmo atingido o final, a última linha seja traduzida. Portanto, a condição de parada será ler até que `fgets` retorne `null`:

```
bool tentouLerAposFinalDoArquivo(char* linha){
    return linha == NULL;
}
```

6.15.5.4 Salvando o conteúdo com a função fwrite

Finalmente, precisamos salvar o conteúdo traduzido na saída:

Escrevendo conteúdo na saída

```
void salvaConteudoNaSaida(char* conteudo, FILE* saida){
    int tamanhoDaMensagem = strlen(conteudo);
    fwrite(conteudo, 1, tamanhoDaMensagem, saida);
}
```

6.15.6 Verificando que os testes continuam passando

Após estas atualizações, vamos verificar se nossos testes continuam passando:

Verificando os testes passando após as modificações

```
$ make clean && make lingua-do-i-test && make test_all
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o ↵
    lingua-do-i-test
./lingua-do-i-test
```

Ótimo, nossos testes estão passando novamente!

6.15.7 Atualizando aplicação para processar fluxos

Após os testes passarem, nosso próximo passo será a atualização da aplicação para utilizar a nova implementação:

Atualizando main em lingua-do-i.c para processar fluxos

```
int main(int argc, const char* argv[]) {
    FILE* entrada = determinaEntrada(argc, argv);
    if (entrada){
        traduzFluxoDeEntradaNaSaida(entrada, stdout);
    }else{
        fprintf(stderr, "Problema ao abrir arquivo: %s\n", argv[1]);
        exit(EXIT_FAILURE);
    }
    return EXIT_SUCCESS;
}
```

Após atualização, vamos compilar a aplicação:

Compilando a aplicação

```
$ make clean && make lingua-do-i
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i.c lingua-do-i-core.c -o lingua-do-i
```

Verificando que a aplicação está processando a entrada por fluxo:

Utilizando aplicação para traduzir fluxo

```
$ cat musica-trecho.txt | ./lingua-do-i
Ih! Diis, pirdii isti pibri ciitidi
```

Ótimo! Nossa aplicação agora é capaz de processar o fluxo de entrada também.

6.15.8 Código da etapa

Você pode consultar o código final desta etapa nos seguintes arquivos:

Código fonte /etapa15/src/lingua-do-i.c[[code/lingua-do-i/etapa15/src/lingua-do-i.c](#)]

Código fonte /etapa15/src/lingua-do-i-test.c[[code/lingua-do-i/etapa15/src/lingua-do-i-test.c](#)]

Código fonte /etapa15/src/lingua-do-i-core.c[[code/lingua-do-i/etapa15/src/lingua-do-i-core.c](#)]

Código fonte /etapa15/src/lingua-do-i-core.h[[code/lingua-do-i/etapa15/src/lingua-do-i-core.h](#)]

Código fonte /etapa15/src/musica-trecho.txt[[code/lingua-do-i/etapa15/src/musica-trecho.txt](#)]

Código fonte /etapa15/src/Makefile[[code/lingua-do-i/etapa15/src/Makefile](#)]

6.16 Etapa 16: Tratando acentos

Revendo nosso arquivo `TODO.txt`, lembramos que ainda está faltando processar mensagens com acentos.

6.16.1 Fazendo o teste de tradução de vogais acentuadas falhar

Nós temos dois testes relacionados à tradução para lingua do i. Por enquanto eles foram implementados para não traduzirem vogais acentuadas e estão passando deste jeito. Para fazê-los falharem nós precisaremos atualizá-los da seguinte forma:

`testTraducaoParaLinguaDoI`

Atualizar o string `TRADUCAO_ESPERADA2`, substituindo á por í:

```
// ANTES
char* TRADUCAO_ESPERADA2= "Ih! Diis, sirá qii i sinhir si zingii";
// DEPOIS
char* TRADUCAO_ESPERADA2= "Ih! Diis, sirí qii i sinhir si zingii";
```

`testTraduzFluxoDeEntrada`

Neste teste utilizamos o arquivo `musica-completa-traduzida.txt` que contém o resultado esperado da tradução. Você pode encontrar o arquivo referido já traduzido na pasta do código desta etapa: `/etapa16/src/musica-completa-traduzida.txt`

Após estas modificações, executamos os testes e verificamos que primeira falha está relacionada ao nosso primeiro teste, devido a nossa alteração em `TRADUCAO_ESPERADA2`.

Rexecutando os testes após as modificações com vogais acentuadas

```
$ make clean && make lingua-do-i-test && make test_all
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o ↵
    lingua-do-i-test
./lingua-do-i-test
lingua-do-i-test: lingua-do-i-test.c:11: ↵
    verificaConteudosSaoIguais: Assertion `strcmp(conteudo, ↵
    esperado) == 0 && "conteúdo deve ser igual ao esperado"' ↵
    failed.
make: *** [test_all] Abortado (arquivo core criado)
```

6.16.2 Entendendo o que é necessário para fazer o teste passar

Para fazer o teste passar, precisamos transformar “á” em “í”. Sabemos até agora que estes caracteres são representados com mais de um byte. Vamos verificar sua representação em decimal:

Utilizando comando `od` para ver representação dos caracteres

```
$ echo áéíóú | od -t d1
00000000  -61  -95  -61  -87  -61  -83  -61  -77  -61  -70   10
0000013
```



Dica

O comando `od`, disponível no Linux possui várias opções para exibição de conteúdos binários. Neste caso estamos utilizando para imprimir o valor de um byte com sinal, que equivale ao valor de `signed char`.

Descobrimos que todos estes caracteres possuem dois bytes, onde o primeiro tem valor `-61` e o segundo depende da vogal utilizada, confira as representações na tabela a seguir:

Tabela 6.1: Representação de vogais acentuados em UTF-8

Carácter	Primeiro Byte	Segundo Byte
á	-61	-95
é	-61	-87
í	-61	-83
ó	-61	-77
ú	-61	-70

**Nota**

O que precisamos fazer é implementar uma tradução que identifique dois bytes consecutivos desta tabela e escreva na saída os dois bytes do carácter *í*.

6.16.3 Fazendo o primeiro teste de tradução acentuada passar

Decidimos modificar a função `traduzParaLinguaDoI`, atualmente responsável por traduzir uma linha para a língua do *i*:

```
char* traduzParaLinguaDoI(char* mensagemOriginal) {
    int tamanhoDaMensagem = strlen(mensagemOriginal);
    char* traducao = calloc(1, tamanhoDaMensagem+1);

    for(int i=0; i<tamanhoDaMensagem; i++){
        char caracterDaMensagemOriginal = mensagemOriginal[i];
        if (podeSerVogalAcentuada(caracterDaMensagemOriginal)) { ❶
            int incremento = traduzCaracteresAcentuados(
                mensagemOriginal, i, traducao); ❷
            i+=incremento; ❸
        } else { ❹
            traducao[i] = traduzCaracterParaLinguaDoI(❺
                caracterDaMensagemOriginal); ❻
        }
    }
    return traducao;
}
```

- ❶, ❹, ❺, ❻ Nossa primeira modificação consiste em verificar se o primeiro byte lido corresponde ao início de uma vogal acentuada (`podeSerVogalAcentuada`). Se ele **não** for, então seguirá o fluxo normal, sendo traduzida pela função que traduz caracteres de um byte (`traduzCaracterParaLinguaDoI`), e adicionada ao String que corresponde à tradução. **Ainda precisamos criar a função `podeSerVogalAcentuada`.**
- ❷ Caso o primeiro byte lido seja o início de vogal acentuada, chamaremos a função `traduzCaracteresAcentuados` que deverá ler os bytes restantes necessários para representar o carácter e salvá-los na posição indicada por *i* no String `traducao`. Teoricamente um carácter acentuado poderia possuir mais de 2 bytes, portanto esta função precisará retornar um incremento para ser utilizado em *i*, para garantir a consistência da posição de leitura. **A função `traduzCaracteresAcentuados` ainda não existe**, será criada mais adiante.
- ❸ Atualizamos o valor de *i* com o incremento necessário. É importante perceber que no comando `for`, a variável já é incrementada (`i++`), portanto não precisamos incrementá-la se apenas um byte for lido em `traduzCaracteresAcentuados`.

**Dica**

Certifique-se de ter entendido o código apresentado antes de continuar.

Dando sequência, vamos criar as funções que desejamos utilizar, iniciaremos com `podeSerVogalAcentuada`:

```
bool podeSerVogalAcentuada(char caracterInicial){  
    return caracterInicial == -61; ❶  
}
```

- ❶ A implementação desta função é muito simples, conforme resumido na Tabela 6.1 [145], o primeiro byte de todas as vogais com acento agudo tem o valor `-61`. A função simplesmente compara o byte lido com este valor, retornando o resultado da comparação.

Em seguida, vamos implementar a última função que ficou faltando:

```
int traduzCaracteresAcentuados(char* mensagem, int i,  
    char* traducao){  
    int incremento;  
    char primeiroByte = mensagem[i];  
    char s = mensagem[i+1]; // segundo byte ❶  
  
    assert(primeiroByte == -61); // prefixo dos acentos ❷  
    traducao[i] = primeiroByte; // passa o primeiro byte ❸  
  
    if (s == -95 or s == -87 or s == -77 or s == -70){ // áéóú ❹  
        traducao[i+1] = -83; // í ❺  
    }else{  
        traducao[i+1] = mensagem[i+1]; ❻  
    }  
    incremento = 1; ❼  
    return incremento;  
}
```

- ❶ Iniciamos a função buscando os valores dos dois primeiros bytes. É importante observar aqui os nomes das variáveis. A variável `primeiroByte` possui um nome bastante representativo, não há necessidade de comentários. Já o segundo byte lido, estamos nomeando-o apenas por `s`, para diminuir o tamanho do código. No entanto, estamos adicionando um comentário explicando qual o propósito desta variável, esta é uma boa prática!
- ❷ A função `assert` está indicando que o pré-requisito desta função é ser chamada quando o primeiro byte possuir o valor `-61`, caso contrário seu comportamento não está definido. Aqui o comentário mais uma vez é importante, sem ele o número em questão poderia ser considerado um Número Mágico.³ Para utilização do `assert`, também foi necessário incluir no início do código: `#include <assert.h>`.
- ❸ `traducao[i]` corresponde a posição onde deveremos escrever a tradução do carácter lido. Neste trecho, escrevemos o primeiro byte lido na posição correspondente.
- ❹ Neste `if` estamos comparando o segundo byte lido com o segundo byte dos caracteres `áéóú`, conforme indicado no comentário.

³Você pode saber mais sobre Número Mágico em informática no seguinte link: [http://pt.wikipedia.org/wiki/Número_mágico_\(informática\)](http://pt.wikipedia.org/wiki/Número_mágico_(informática))

- 5 Caso o segundo byte lido corresponda ao segundo byte dos caracteres indicados, sua tradução corresponderá ao segundo byte de í, que possui o valor -83. Salvamos a tradução na posição `traducao[i+1]`.
- 6 Caso contrário, o segundo byte é escrito conforme lido. Este é o caso inclusive do carácter í, que sua tradução corresponde a ele mesmo.
- 7 Por fim, retornamos 1 como incremento, indicando que só foi necessário ler um byte após o primeiro. Também seria possível implementar o retorno da seguinte forma: `return 1;`, mas a intenção não fica bem transmitida.

Após as alterações em `traduzCaracteresAcentuados`, vamos re-executar os testes:

Re-execução dos testes após modificações

```
$ make clean && make lingua-do-i-test && make test_all
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o ↵
    lingua-do-i-test
./lingua-do-i-test
lingua-do-i-test: lingua-do-i-test.c:98: ↵
    verificaMusicaFoiTraduzidaCorretamenteNaSaida: Assertion ` ↵
    strcmp(conteudo, traducao) == 0 && "Arquivo deve ser ↵
    traduzido corretamente" failed.
make: *** [test_all] Abortado (arquivo core criado)
```

Perceba que a falha agora é outra, nosso primeiro testes passou! A falha ocorre em `verificaMusicaFoiTraduzidaCorretamenteNaSaida` que corresponde ao segundo teste, devido as atualizações que fizemos no arquivo `musica-completa-traduzida.txt`. Isto significa que nossa implementação conseguiu traduzir corretamente á para í. Mas será que ela traduz também é para í? Nós esperamos que sim, mas não temos nenhum teste comprovando isso. Vamos fazer isto a seguir.

6.16.4 Criando teste de tradução específico para vogais acentuadas

Vamos criar um novo teste com o propósito específico para traduzir vogais acentuadas:

Teste específico para vogais acentuadas

```
char* VOGAIS_ACENTUADAS= "áéíóúÁÉÍÓÚ-âêîôûÂÊÎÔÛ-àèìòùÀÈÌÒÙ";
char* ACENTOS_ESPERADOS= "íííííííííí-îîîîîîîîîî-ïïïïïïïïïï";
void testTraduzVogaisAcentuadas() {
    char* traducao = traduzParaLingaDoI(VOGAIS_ACENTUADAS);
    verificaConteudosSaoIguais(traducao, ACENTOS_ESPERADOS);
}
```

O teste é bastante simples, todas as vogais no String `VOGAIS_ACENTUADAS` deverão ser traduzidas por suas correspondentes em `ACENTOS_ESPERADOS`. Na última linha verificaremos se os conteúdos são idênticos.

Um detalhe importante, é que este teste deve ser chamado antes de `testTraduzFluxoDeEntrada`, que ainda está falhando na tradução da música:

Posição da evocação do teste `testTraduzFluxoDeEntrada` no main

```
int main(void) {
    testLerConteudoDoArquivo();
    testSalvaConteudoEmArquivo();
    testTraducaoParaLinguaDoI();

    testDefinirEntradaPadrao();
    testDefinirEntradaDeArquivo();
    testDefinirEntradaDeArquivoInexistente();

    testTraduzVogaisAcentuadas(); ❶
    testTraduzFluxoDeEntrada();

    return EXIT_SUCCESS;
}
```

❶ Precisa ser evocado antes de `testTraduzFluxoDeEntrada`.

Vamos executar os testes novamente, esperamos que a falha seja diferente da anterior:

```
$ make clean && make lingua-do-i-test && make test_all
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o ↵
    lingua-do-i-test
./lingua-do-i-test
lingua-do-i-test: lingua-do-i-test.c:11: ↵
    verificaConteudosSaoIguais: Assertion `strcmp(contenido, ↵
    esperado) == 0 && "conteúdo deve ser igual ao esperado"' ↵
    failed.
make: *** [test_all] Abortado (arquivo core criado)
```

Ótimo! Como podemos ver, temos uma falha diferente devido a inclusão do nosso novo teste. O próximo passo será fazer o teste passar.

6.16.5 Fazendo o teste específico das vogais acentuadas passar

Nós já temos toda a *infraestrutura* necessária para tradução das vogais acentuadas, precisamos apenas descobrir os bytes que representam todas estas letras e atualizar a função `traduzCaracteresAcentuados`.

Invocaremos o comando `od` novamente para descobrir os bytes dos caracteres:

```
$ echo áéíóú ÁÉÍÓÚ âêîôû ÂÊÎÔÛ àèìòù ÀÈÌÒÙ | od -t d1
0000000 -61 -95 -61 -87 -61 -83 -61 -77 -61 -70 32 ↵
    -61 -127 -61 -119 -61
0000020 -115 -61 -109 -61 -102 32 -61 -94 -61 -86 -61 ↵
    -82 -61 -76 -61 -69
0000040 32 -61 -126 -61 -118 -61 -114 -61 -108 -61 -101 ↵
    32 -61 -96 -61 -88
0000060 -61 -84 -61 -78 -61 -71 32 -61 -128 -61 -120 ↵
    -61 -116 -61 -110 -61
```

```
0000100 -103 10
0000102
```

Sabendo que espaço corresponde ao carácter 32, percebemos que todas as nossas vogais começam com o valor do primeiro byte igual a **-61**. Vamos construir outra tabela para auxiliar na tradução:

Tabela 6.2: Representação das vogais acentuadas em UTF-8

Char	2º Byte	Char	2º Byte	Char	2º Byte	Char	2º Byte	Char	2º Byte	Char	2º Byte
á	-95	Á	-127	â	-94	Â	-126	à	-96	À	-128
é	-87	É	-119	ê	-86	Ê	-118	è	-88	È	-120
í	-83	Í	-115	î	-82	Î	-114	ì	-84	Ì	-116
ó	-77	Ó	-109	ô	-76	Ô	-108	ò	-78	Ò	-110
ú	-70	Ú	-102	û	-69	Û	-101	ù	-71	Ù	-103

A atualização da implementação é trivial, basta comparar o segundo byte do carácter com os valores da tabela, substituindo-o pelo segundo byte da vogal i com o acento correspondente:

Atualizando para tradução dos caracteres acentuados

```
int traduzCaracteresAcentuados(char* mensagem, int i,
    char* traducao){
    int incremento;
    char primeiroByte = mensagem[i];
    char s = mensagem[i+1]; // segundo byte

    assert (primeiroByte == -61); // prefixo dos acentos
    traducao[i] = primeiroByte; // passa o primeiro byte

    if (s== -94 or s== -86 or s== -76 or s== -69){ // âêôû
        traducao[i+1] = -82; // î
    }else if (s== -95 or s== -87 or s== -77 or s== -70){ // áéóú
        traducao[i+1] = -83; // í
    }else if (s== -96 or s== -88 or s== -78 or s== -71){ // àèòù
        traducao[i+1] = -84; // ì
    }else if (s== -126 or s== -118 or s== -108 or s== -101){ // ÂÊÔÛ
        traducao[i+1] = -114; // Î
    }else if (s== -127 or s== -119 or s== -109 or s== -102){ // ÁÉÓÚ
        traducao[i+1] = -115; // Í
    }else if (s== -128 or s== -120 or s== -110 or s== -103){ // ÀÈÒÙ
        traducao[i+1] = -116; // Ì
    }else{
        traducao[i+1] = mensagem[i+1];
    }
    incremento = 1;
    return incremento;
}
```

Após nossas modificações, vamos executar os testes novamente:

Executando o teste após as modificações

```
$ make clean && make lingua-do-i-test && make test_all
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o ↵
    lingua-do-i-test
./lingua-do-i-test
lingua-do-i-test: lingua-do-i-test.c:98: ↵
    verificaMusicaFoiTraduzidaCorretamenteNaSaida: Assertion `↵
    strcmp(conteudo, traducaao) == 0 && "Arquivo deve ser ↵
    traduzido corretamente"' failed.
make: *** [test_all] Abortado (arquivo core criado)
```

Estranho! Nosso teste passou e a falha ocorrida foi na função `verificaMusicaFoiTraduzidaCorretamenteNaSaida`. Mas o estranho é que teoricamente, uma vez que nossa função de tradução estivesse corretada, ela deveria traduzir a música também!

6.16.6 Fazendo o teste de tradução da música passar com os acentos

A causa da falha é mais fácil de ser encontrada quando sabemos depurar um programa. Após uma depuração (não mostrada aqui), descobrimos que o teste falhou pois o programa não conseguiu traduzir ã para ã.

Realmente, nosso programa não estava traduzindo os caracteres com ~. Vamos descobrir suas representações:

```
$ echo "ã ~e ã õ ã Ã ~E ã ã ã" | od -t d1
0000000 -61 -93 32 -31 -70 -67 32 -60 -87 32 -61 ↵
    -75 32 -59 -87 32
0000020 32 -61 -125 32 -31 -70 -68 32 -60 -88 32 ↵
    -61 -107 32 -59 -88
0000040 10
```

Importante



Infelizmente tivemos um problema técnico na impressão do carácter e acentuado com o ~, estaremos representando o carácter acentuado através do prefixo ~. Então ~e e ~E representam os caracteres acentuados em minúsculo e maiúsculo, o equivalente com a vogal a seriam: ã e Ã.

Desta vez, nos deparamos com uma novidade! Os caracteres ~e e ~E são codificados com 3 bytes e também não começam com -61! Vamos construir nossa tabela para resumir a representação dos caracteres:

Tabela 6.3: Representação de vogais com ~

Carácter	Bytes	Carácter	Bytes
ã	-61 -93	Ã	-61 -125

Tabela 6.3: (continued)

Carácter	Bytes	Carácter	Bytes
~e	-31 -70 -67	~E	-31 -70 -68
ĩ	-60 -87	Ĩ	-60 -88
õ	-61 -75	Õ	-61 -107
ũ	-59 -87	Ũ	-58 -88

Com a tabela em mãos, precisamos atualizar nossa função que detecta uma possível vogal acentuada:

Atualização da detecção do primeiro byte de uma vogal acentuada

```
bool podeSerVogalAcentuada(char caracter) {
    return caracter == -61 or caracter == -31 or caracter == -60
        or caracter == -59;
}
```

Em seguida, atualizamos a regra de tradução na função `traduzCaracteresAcentuados` da seguinte forma:

Atualização da função para incluir tradução com tio

```
int traduzCaracteresAcentuados(char* mensagem, int i,
    char* traducao){
    int incremento = 1; // por padrão ler dois bytes
    char primeiroByte = mensagem[i];
    char s = mensagem[i+1]; // segundo byte

    assert (primeiroByte == -61); // prefixo dos acentos
    traducao[i] = primeiroByte; // passa o primeiro byte por padrão

    if (primeiroByte == -61){
        if (s== -94 or s== -86 or s== -76 or s== -69){ // âêôû
            traducao[i+1] = -82; // î
        }else if (s== -95 or s== -87 or s== -77 or s== -70){ // áéóú
            traducao[i+1] = -83; // í
        }else if (s== -96 or s== -88 or s== -78 or s== -71){ // àèòù
            traducao[i+1] = -84; // ï
        }else if (s== -126 or s== -118 or s== -108 or s== -101){ // ÂÊÔÛ
            traducao[i+1] = -114; // Î
        }else if (s== -127 or s== -119 or s== -109 or s== -102){ // ÁÉÓÚ
            traducao[i+1] = -115; // Í
        }else if (s== -128 or s== -120 or s== -110 or s== -103){ // ÀÈÒÙ
            traducao[i+1] = -116; // Ï
        }else if (s== -93 or s== -75){ // ãõ
            traducao[i] = -60; // ã - substitui primeiro byte
            traducao[i+1] = -87; // õ
        }else{
            traducao[i+1] = mensagem[i+1];
        }
    }else if (primeiroByte == -31){
        char terceiro = mensagem[i+2];
```

```
if (s == -70 and terceiro == -67){ // &#x1ebd;
    traducao[i] = -60; // ã
    traducao[i+1] = -87; // ã
    incremento = 2; // descartamos o terceiro byte
}else{
    traducao[i] = primeiroByte;
    traducao[i+1] = s;
    incremento = 1;
}
}else if (primeiroByte == -59){
    if ( s == -87 ) { // ã
        traducao[i] = -60; // ã
        traducao[i+1] = -87; // ã
    }else{
        traducao[i] = primeiroByte;
        traducao[i+1] = s;
    }
}
}else{
    traducao[i] = primeiroByte;
    traducao[i+1] = s;
}
return incremento;
}
```

Nossa implementação ficou grande,⁴ mas vamos verificar se ela está correta executando os testes:

Executando os testes após as modificações

```
$ make clean && make lingua-do-i-test && make test_all
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o ↵
    lingua-do-i-test
./lingua-do-i-test
```

Ótimo! Nossos testes estão passando novamente! Nosso programa está traduzindo corretamente para língua do i!⁵

6.16.7 Código da etapa

Você pode consultar o código final desta etapa nos seguintes arquivos:

Código fonte /etapa16/src/lingua-do-i.c[[code/lingua-do-i/etapa16/src/lingua-do-i.c](#)]

Código fonte /etapa16/src/lingua-do-i-test.c[[code/lingua-do-i/etapa16/src/lingua-do-i-test.c](#)]

Código fonte /etapa16/src/lingua-do-i-core.c[[code/lingua-do-i/etapa16/src/lingua-do-i-core.c](#)]

Código fonte /etapa16/src/lingua-do-i-core.h[[code/lingua-do-i/etapa16/src/lingua-do-i-core.h](#)]

Código fonte /etapa16/src/musica-trecho.txt[[code/lingua-do-i/etapa16/src/musica-trecho.txt](#)]

Código fonte /etapa16/src/musica-trecho.txt[[code/lingua-do-i/etapa16/src/musica-completa.txt](#)]

⁴Alguns estilos de código defendem que se o código de uma função não couber na tela, ela está grande demais e deveria delegar responsabilidades para outras funções.

⁵Realmente, um programa está correto até que alguém encontre um erro ou *bug*.

Código fonte `/etapa16/src/musica-trecho.txt`[code/lingua-do-i/etapa16/src/musica-completa-traduzida.txt]

Código fonte `/etapa16/src/Makefile`[code/lingua-do-i/etapa16/src/Makefile]



Feedback sobre o capítulo

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

Para compreender melhor como feedbacks funcionam consulte o guia do curso.

Apêndice A

Depuração

A **Depuração** de um programa consiste em acompanhar a execução do programa, parando em pontos específicos do código fonte e verificando os valores das variáveis do programa.

A.1 Requisitos da depuração

Para depurar um programa é necessário que o programa seja compilado com instruções para depuração. No `gcc` isto é possível através do parâmetro `-g`.

A.2 Depurando o programa `lingua do i`

Como exemplo de depuração nós vamos utilizar o programa que estava sendo desenvolvido na Seção 6.8 [119].

A.2.1 Checando o pré-requisito para depuração

Vamos verificar o `Makefile` do projeto:

Código fonte `/etapa7/src/Makefile`[`code/lingua-do-i/etapa7/src/Makefile`]

```
CC=gcc
CFLAGS=-Wall -g -std=c1x

all: lingua-do-i lingua-do-i-test

lingua-do-i: lingua-do-i.c
    $(CC) $(CFLAGS) lingua-do-i.c lingua-do-i-core.c -o lingua-do-i

lingua-do-i-test: lingua-do-i-test.c
    $(CC) $(CFLAGS) lingua-do-i-test.c lingua-do-i-core.c -o lingua-do-i-test ↵

test_all:
    ./lingua-do-i-test
```

```
clean:
    rm -rf lingua-do-i lingua-do-i-test
```

Percebam que o parâmetro `-g` está presente nas flags de compilação:

```
CFLAGS=-Wall -g -std=c1x
```

A.2.2 Última execução do programa

Vamos relembrar o resultado da última execução do programa:

Última execução de lingua-do-i-test

```
$ make clean && make lingua-do-i-test && ./lingua-do-i-test
rm -rf lingua-do-i lingua-do-i-test
gcc -Wall -g -std=c1x lingua-do-i-test.c lingua-do-i-core.c -o ↵
    lingua-do-i-test
lingua-do-i-test: lingua-do-i-test.c:11: ↵
    verificaConteudosSaoIguais: Assertion `strcmp(conteudo, ↵
    esperado) == 0 && "conteúdo deve ser igual ao esperado"' ↵
    failed.
Abortado (imagem do núcleo gravada)
```

O código fonte utilizado na execução foi:

Código fonte `/etapa7/src/lingua-do-i-test.c`[\[code/lingua-do-i/etapa7/src/lingua-do-i-test.c\]](#)

Código fonte para os testes

```
1  #include <assert.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include "lingua-do-i-core.h"
6
7  void verificaConteudosSaoIguais(char* conteudo, char* esperado){
8      assert(conteudo != NULL && "conteúdo não pode ser NULL");
9
10     assert( strcmp(conteudo, esperado) == 0
11             && "conteúdo deve ser igual ao esperado");
12 }
13
14 char* NOME_DO_ARQUIVO = "musica-trecho.txt";
15 char* CONTEUDO_ESPERADO = "Oh! Deus, perdoe este pobre coitado";
16 void testLerConteudoDoArquivo(){
17     char* conteudo = lerConteudoDoArquivo(NOME_DO_ARQUIVO);
18     verificaConteudosSaoIguais(conteudo, CONTEUDO_ESPERADO);
19 }
20
21 char* MENSAGEM_ORIGINAL="Minhas vogais, tudo aqui.";
22 char* TRADUCAO_ESPERADA="Minhis vigiis, tidi iqui.";
23 void testTraducaoParaLinguaDoI(){
24     char* mensagemTraduzida = traduzParaLinguaDoI(MENSAGEM_ORIGINAL);
```

```
25     verificaConteudosSaoIguais(mensagemTraduzida, TRADUCAO_ESPERADA);
26 }
27
28 int main(void) {
29     testLerConteudoDoArquivo();
30     testTraducaoParaLinguaDoI();
31
32     return EXIT_SUCCESS;
33 }
```

A.2.3 Escolha do ponto de parada

Com base no desenvolvimento do projeto e com a mensagem de erro nós temos algumas opções de ponto de parada:

1. Na linha do erro

```
assert( strcmp(contenido, esperado) == 0
        && "conteúdo deve ser igual ao esperado");
```

Este é um ponto de parada bastante intuitivo, uma vez que a mensagem de erro indicou-a. A desvantagem deste ponto é que a função `verificaConteudosSaoIguais` é chamada mais de uma vez no arquivo, então teríamos que descobrir a interação que ocorreu o erro. Embora neste caso ela só foi chamada uma única vez, em aplicações maiores talvez seja trabalhoso identificar a iteração onde ocorrerá o erro.

2. No método de tradução

```
char* mensagemTraduzida = traduzParaLinguaDoI(MENSAGEM_ORIGINAL);
```

Nós estávamos elaborando a função `traduzParaLinguaDoI` antes do teste falhar, então talvez seja conveniente adicionar um ponto de parada nesta linha e acompanhar a execução. Uma desvantagem é que dentro desta função existe um loop com várias iterações, talvez seja difícil encontrar a iteração com erro.

3. Antes de verificação

```
verificaConteudosSaoIguais(mensagemTraduzida, TRADUCAO_ESPERADA);
```

Inicialmente este parece ser um ótimo ponto de parada para compreender o problema, pois teríamos como comparar os valores de `mensagemTraduzida` com `TRADUCAO_ESPERADA`. Dependendo da diferença poderemos ter uma dica do problema.



Nota

Nós vamos escolher a **linha 24** como nosso ponto de parada. O número da linha será utilizado no comando `break` para definir o ponto de parada na próxima seção.

A.2.4 Realizando a depuração com o gdb



Dica

O `gdb` é o depurador que utilizaremos para inspecionar nosso programa. Para saber mais detalhes sobre o `gdb` recomendamos a leitura do livro “Debugging with gdb”, disponível em <http://www.gnu.org/software/gdb/documentation/> ele está disponível na versão on-line ou em PDF.

Depuração do programa

```
$ gdb ./lingua-do-i-test
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/ ↵
  gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show ↵
  copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Lendo símbolos de /home/santana/asciibook/linguagem-de-programacao-i- ↵
  livro/livro/capitulos/code/lingua-do-i/etapa7/src/lingua-do-i-test ↵
  ...concluído.
(gdb) break 24 ❶
Ponto de parada 1 at 0x80485f2: file lingua-do-i-test.c, line 24.
(gdb) run ❷
Starting program: /home/santana/asciibook/linguagem-de-programacao-i- ↵
  livro/livro/capitulos/code/lingua-do-i/etapa7/src/lingua-do-i-test

Breakpoint 1, testTraducaoParaLinguaDoI () at lingua-do-i-test.c:24
24      char* mensagemTraduzida = traduzParaLinguaDoI(MENSAGEM_ORIGINAL ↵
  );
(gdb) next ❸
25      verificaConteudosSaoIguais(mensagemTraduzida, ↵
  TRADUCAO_ESPERADA);
(gdb) s ❹
verificaConteudosSaoIguais (conteudo=0x804b008 "Minhis vigiis, tidi ↵
  iqii.",
      esperado=0x80489ba "Minhis vigiis, tidi iqii.") at lingua-do-i- ↵
  test.c:8
8      assert(conteudo != NULL && "conteúdo não pode ser NULL");
(gdb) next ❺
10      assert( strcmp(conteudo, esperado) == 0
(gdb) print conteudo ❻
$1 = 0x804b008 "Minhis vigiis, tidi iqii."
(gdb) print esperado ❼
$2 = 0x80489ba "Minhis vigiis, tidi iqii."
(gdb) print strcmp (conteudo , esperado) ❽
$3 = -1
(gdb) print strncmp (conteudo , esperado,15) ❾
```

```
$4 = 0
(gdb) print strcmp (conteudo , esperado,20) ❶
$5 = 0
(gdb) print strcmp (conteudo , esperado,25) ❷
$6 = -1
(gdb) print strcmp (conteudo , esperado,21) ❸
$7 = 0
(gdb) print strcmp (conteudo , esperado,22) ❹
$8 = 0
(gdb) print strcmp (conteudo , esperado,23) ❺
$9 = -1
(gdb) print conteudo[22] ❻
$10 = 105 'i'
(gdb) print esperado[22] ❼
$11 = 117 'u'
(gdb) print esperado
$12 = 0x80489ba "Minhis vigiis, tidi iqui."
(gdb) quit ❽
A debugging session is active.

    Inferior 1 [process 13644] will be killed.

Quit anyway? (y or n) y
```

- ❶ Antes de iniciar a execução do aplicativo, nós estabelecemos nosso ponto de parada (*break point*), através deste comando: `break 24`, definindo que a depuração irá iniciar na linha 24.
- ❷ Em seguida solicitamos a execução do programa através do comando `run`. O programa então é executado até chegar no ponto de parada, quando então imprime número da linha, seguido do código contido nesta linha.
- ❸ Neste momento temos duas escolhas, ou podemos entrar na função `traduzParaLinguaDoI` através do comando `s` ou podemos pular para próxima linha através do comando `next`. Antes de inspecionar o funcionamento da função, decidimos primeiro inspecionar porque a verificação estava falhando.
- ❹ Diferente do comando `next` que avança para a próxima linha, decidimos entrar na função `verificaConteudosSaoIguais` através do comando **step** (`s`). O depurador entra na função imprimindo os valores que foram passados como parâmetro, seguido da linha e o código atual.
- ❺ Nós sabemos que conteúdo não é `NULL`, então avançamos para a próxima linha.
- ❻, ❼ Neste momento estamos utilizando o comando `print` para exibir os conteúdos das variáveis. Até agora **não** percebemos a razão do teste está falhando.
- ❽ Solicitamos a invocação da função em C que realiza comparação de strings (`strcmp()` [171]) passando os dois strings com parâmetros. A resposta indica que eles realmente são diferentes, mas ainda não percebi a diferença.
- ❾, ❿, ⓫, ⓬, ⓭, ⓮ Outra função que realiza comparação é a `strncmp()` [171], que possui um último parâmetro para indicar o tamanho da comparação. Realizamos diversas inspeções para identificar aonde os strings são diferentes. Percebemos que os strings diferem no 23º carácter.

- 15, 16 Imprimimos o 23º carácter dos dois strings. Neste momento percebemos que há algo de errado com o string esperado. O programa língua do i converte todas as vogais para **i**, não deveria nenhuma outra vogal, inclusive o **u**. Em seguida percebemos que o erro estava na palavra *iqui*, na frase que utilizamos no teste como resultado esperado.
- 17 Pressionamos CTRL+D para sair do programa.

A.2.5 Resultado da depuração

Após realizamos a depuração fomos capazes de identificar o erro, que estava no nosso arquivo de teste, na palavra *iqui*:

```
char* TRADUCAO_ESPERADA= "Minhis vigiis, tidi iqui.";
```

O correto seria:

```
char* TRADUCAO_ESPERADA= "Minhis vigiis, tidi iqii.";
```

A correção do arquivo de teste pode ser visto na Seção 6.8.2 [119].



Nota

Embora você pode ter percebido este erro muito antes, frequentemente estamos tão envolvidos com o código fonte que não conseguimos perceber os erros mesmo estando evidentes para terceiros. Por isso a depuração é tão importante, ela nos permite perceber estes erros.

Apêndice B

Conteúdo extra: Arquivos especiais

Arquivo e Diretório são um conceitos abordados em disciplinas de programação, no entanto, somente em disciplinas de Sistemas Operacionais é que estes conceitos são aprofundados.

Arquivos especiais são arquivos que não armazenam conteúdo. As seções a seguir apresentam como são os arquivos especiais no Windows e no Linux.



Importante

Embora este assunto não seja exigido numa disciplina de Programação, este conhecimento possibilitará escrever programas que leiam e escrevam em dispositivos.

B.1 Arquivos especiais que representam dispositivos

Alguns arquivos podem representar dispositivos, de tal forma que escrever ou ler nesses arquivos corresponde a enviar ou ler dados no dispositivo.

Existem arquivos que representam os discos, partições, impressoras, placa de rede, placa de som, microfone entre outros dispositivos.

B.1.1 No Linux

Como exemplo da utilização deste tipo de arquivo especial, temos o seguinte comando no Linux, que o ler do arquivo `/dev/cdrom` (que representa o CD-ROM) e salva a imagem do CD-ROM no arquivo `cdrom_image.iso`:

Utilização do comando `dd` para criar uma imagem de `cd`

```
$ dd if=/dev/cdrom of=cdrom_image.iso
```

Neste comando o parâmetro `if` (*input file*) especifica o arquivo de entrada, que neste caso é um arquivo especial que representa o CD-ROM (`/dev/cdrom`). Enquanto que `of` (*output file*) especifica o arquivo de saída que será criado: `cdrom_image.iso`.

B.1.2 No Windows

No Windows os seguintes arquivos representam dispositivos:

CON

O console do sistema. É uma combinação do teclado (quando utilizado como leitura) e a tela (durante a escrita).

PRN

Representa a impressora padrão.

LPT1 à LPT4

Representa as portas paralelas. Os computadores modernos não costumam mais possuir esta entrada.

COM1 à COM4

Representa as portas seriais.

NUL

Tudo que for enviado para este arquivo é descartado.

CLOCK\$

Representa o dispositivo do relógio do sistema. Este arquivo não funciona nos sistemas modernos.

B.1.2.1 Testando arquivos de dispositivos no Windows

Para testar os arquivos especiais no Windows, abra o **Prompt de Comando do MS-DOS** e digite os comandos informados a seguir.

Neste primeiro teste vamos comparar a escrita em arquivo normal e a escrita no arquivo especial NUL, no Windows:

Testando arquivo especial NUL do Windows

```
> echo "oi" > saida.txt ❶  
> type saida.txt ❷  
oi ❸  
> echo "oi" > nul ❹  
> type nul ❺  
❻
```

- ❶ Invoca o comando `echo` para escrever a mensagem `oi` e redireciona a saída ao arquivo `saida.txt` que será criado ou sobrescrito, caso exista.
- ❷ Utiliza o comando `type` para ler o conteúdo do arquivo.
- ❸ Exibição do conteúdo do arquivo lido.
- ❹, ❺ Faz o mesmo, mas redireciona o conteúdo para o arquivo especial NUL.
- ❻ Nada é exibido, pois tudo que foi enviado para NUL foi descartado.

Neste segundo teste, vamos comparar a criação de um diretório com a tentativa de criar um novo diretório com um nome de arquivo reservado (especial):

Testando arquivo especial CON no Windows

```
> mkdir novodir ❶  
> mkdir con ❷  
O nome de pasta é inválido ❸
```

- ❶ Utiliza o comando `mkdir` para criar um diretório chamado `novodir`. O diretório é criado com sucesso.
- ❷ Tenta criar um diretório utilizando o nome de um arquivo especial: `con`.
- ❸ Mensagem de erro ao tentar criar diretório com o nome reservado.

Dica



Como desafio, deixamos um último teste para você. O que iria acontecer se utilizássemos o comando `TYPE` para ler o conteúdo do arquivo `CON`? Reflita um pouco sobre o resultado esperado depois execute:

Lendo do arquivo especial CON no Windows

```
> TYPE CON
```

B.2 Arquivos especiais que criam um Pipe em memória

Um *Pipe* é um recurso que possibilita dois processos transmitirem dados entre eles. Enquanto um processo *produz* bytes o segundo *consome* os bytes produzidos.

Você pode imaginar que o sistema operacional utiliza o arquivo para criar meio de comunicação entre os dois processos, os bytes que forem escritos no arquivo pelo primeiro processo estarão disponíveis para leitura no segundo processo. O arquivo continua existindo antes e depois da execução dos processos, no entanto ele não possui conteúdo salvo no disco, diferente de um arquivo regular (não especial).



Dica

No Apêndice C [165] demonstramos a criação e utilização deste tipo de arquivo no Linux. Recomendamos esta leitura caso deseje conhecer a utilização deste tipo de arquivo.

B.3 Arquivos especiais devido ao conteúdo

Alguns arquivos são especiais devido ao conteúdo que apresentam durante sua leitura.

No Linux, quando lemos o arquivo `/proc/uptime` o sistema retorna quanto tempo (em segundos) o sistema está ligado:

Exibindo o conteúdo do arquivo `/proc/uptime`

```
$ cat /proc/uptime  
9913.84 38485.65
```

Neste caso podemos ver que o computador permaneceu ligado por 9913.84 segundos, que equivale a aproximadamente 2,7 horas.¹

**Nota**

O tipo de arquivo *especial devido ao conteúdo* não existe no Windows, todos os arquivos especiais neste sistema estão relacionados aos dispositivos.

¹O segundo número é a soma da quantidade de segundos que os *cores* permaneceram ociosos.

Apêndice C

Experimento com mkfifo para demonstrar arquivos especiais

Neste experimento iremos demonstrar a utilização do tipo de arquivo especial *Named Pipe*, que pode ser utilizados para transmitir dados entre dois aplicativos em execução no sistema operacional Linux.

C.1 Imprimindo na saída padrão

Para imprimir na saída padrão, vamos utilizar o comando `cat` para ler linhas da entrada padrão e envia-las para a saída padrão. Para sair, pressionamos CTRL+D numa linha em branco. Para facilitar a identificação das linhas, iremos invocar o comando com o parâmetro `-n`, que irá numerar as linhas na saída:

Invocando comando `cat` para imprimir linhas numeradas na saída padrão

```
$ cat -n
As linhas serão impressas na saída padrão,
  1  As linhas serão impressas na saída padrão,
que neste caso é a tela.
  2  que neste caso é a tela.
CTRL+D numa linha em branco para sair.
  3  CTRL+D numa linha em branco para sair.
```

Para direcionar a saída padrão para um arquivo, basta invocar o comando `cat` direcionando a saída com `>`, indicando o arquivo de destino:

Redirecionando saída padrão para um arquivo

```
$ cat -n > saida.txt
Com a saída padrão direcionada para um arquivo
as linhas deixaram de serem impressas na tela.
Para encerrar pressionamos CTRL+D em nova linha
```

A invocação `cat` a seguir imprime o conteúdo do arquivo na saída padrão, que neste caso foi a tela:

```
$ cat saida.txt
  1  Com a saída padrão direcionada para um arquivo
  2  as linhas deixaram de serem impressas na tela.
```

3 Para encerrar pressionamos CTRL+D em nova linha

Na próxima seção, vamos aprender como criar um arquivo especial.

C.2 Criando um arquivo especial com mkfifo

O comando `mkfifo` cria um arquivo do tipo especial (*Named Piped*). Todo o conteúdo que for escrito nele será mantido em um Buffer na memória, até que outro processo leia do arquivo.

Criando um arquivo especial com o mkfifo

```
$ mkfifo arquivo-especial.fifo
```

Vamos listar os arquivos para comparar os tamanhos:

Verificando os tamanhos dos arquivos

```
$ du -ab
165 ./saida.txt
0   ./arquivo-especial.fifo
```

Percebam que o arquivo `saida.txt` ocupa 165 bytes, enquanto `arquivo-especial.fifo` ocupa 0 byte.

C.3 Utilizando o arquivo especial

Neste experimento vamos utilizar o arquivo especial criado para possibilitar que dois processos transmitam dados entre si.

Vamos começar abrindo um terminal (`term1`) e utilizando o comando `cat` para escrever conteúdo no arquivo:

Primeiro terminal, enviando duas linhas para o buffer em memória

```
term1$ cat -n > arquivo-especial.fifo
Primeira linha digitada
Segunda linha digitada
```

As linhas digitadas foram escritas no arquivo, no entanto, como ele é um arquivo especial o sistema operacional está armazenando o que foi escrito num Buffer em memória, aguardando que um outro aplicativo *abra o arquivo para leitura*. O terminal não finaliza, fica aguardando que novas linhas sejam digitadas.



Nota

Percebam que caso houvesse algum erro no envio da primeira linha para a saída, o `cat` encerraria e não seria possível digitar a segunda linha.

Nosso próximo passo será abrir um novo terminal (`term2`), e tentar ler o conteúdo do arquivo. Para isto invocamos o comando `cat`, passando no nome do arquivo:

Segundo terminal, utilizando o cat para ler do arquivo

```
term2$ cat arquivo-especial.fifo
1  Primeira linha digitada
2  Segunda linha digitada
```

No segundo terminal o `cat` foi capaz de ler as linhas que foram digitadas e salvas num buffer, antes de sua execução. Ele também não finaliza, permanece aberto, aguardando que novas linhas sejam disponibilizadas através do arquivo.



Nota

Como a segunda aplicação foi capaz de ler o que foi escrito anteriormente pela primeira aplicação isto comprova que um Buffer realmente existiu, guardando as duas linhas digitadas para leitura posterior.

Em seguida vamos digitar mais uma linha no primeiro terminal. Assim que pressionamos `ENTER`, o segundo terminal recebe a linha digitada pelo arquivo, demonstrando a conexão entre os dois processos:

Primeiro terminal: digitando a terceira linha

```
term1$ cat -n > arquivo-especial.fifo
Primeira linha digitada
Segunda linha digitada
Terceira linha digitada
```

Segundo terminal: recebendo a terceira linha rapidamente

```
term2$ cat arquivo-especial.fifo
1  Primeira linha digitada
2  Segunda linha digitada
3  Terceira linha digitada
```



Nota

No momento em que a segunda aplicação leu o conteúdo do Buffer, ele deixou de existir. Tudo que for escrito no arquivo pelo primeiro processo será direcionado para o segundo, que está lendo dele. Você pode imaginar que ouve uma conexão entre os dois processos, como se existisse um cano (*pipe*) ligando a saída de um processo na entrada do outro.

Agora nós temos duas alternativas, finalizar a comunicação através do primeiro terminal ou do segundo, vamos ver as duas opções:

Enviando o final do arquivo (EOF)

Caso pressionarmos `CTRL+D` estaremos enviando o sinal de final de arquivo, que finaliza a entrada do primeiro terminal e a leitura do segundo. O resultado é o seguinte:

Primeiro terminal: entrada finalizada enviando EOF

```
term1$ cat -n > arquivo-especial.fifo
Primeira linha digitada
Segunda linha digitada
Terceira linha digitada
term1$
```

Segundo terminal: finaliza após encontrar final do arquivo

```
term2$ cat arquivo-especial.fifo
 1 Primeira linha digitada
 2 Segunda linha digitada
 3 Terceira linha digitada
$
```

Interrompendo a aplicação de leitura

Quando interrompemos o segundo terminal com CTRL+C a *leitura do arquivo é fechada* e a aplicação é finalizada. A primeira aplicação, no entanto, não sabe que a *conexão* que o sistema operacional montou entre os dois processos, através do arquivo especial, foi interrompida. Ela continua aguardando que o usuário digite a próxima linha. Quando uma nova linha for digitada, ela tentará escrever no arquivo e não conseguirá, pois tanto a conexão como o Buffer deixaram de existir:

Finalizando o segundo terminal com CTRL+C

```
term2$ cat arquivo-especial.fifo
 1 Primeira linha digitada
 2 Segunda linha digitada
 3 Terceira linha digitada
^C
term2$
```

Terminal finaliza tentando escrever no arquivo com conexão desfeita

```
term1$ cat -n > arquivo-especial.fifo
Primeira linha digitada
Segunda linha digitada
Terceira linha digitada
Quarta linha digitada
~/temp/fifo$
```

Importante



Perceba que podemos abstrair que estamos utilizando um arquivo e imaginar que estamos lendo ou escrevendo os dados em um fluxo (*stream*) de entrada ou de saída. Poderemos ler dados do fluxo até encontrar o final da fluxo (EOF). Na linguagem C os arquivos (FILE) possuem esta mesma abstração.

Apêndice D

Bibliotecas

D.1 Padrão C1X da Linguagem C

O último padrão adotado para a Linguagem C foi publicado pela ISO em 12/8/2011 no documento: ISO/IEC 9899:2011. Este padrão é conhecido como C11 ou C1X, você pode consultar gratuitamente a última versão pública do documento em: <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf>.



Nota

Durante o livro iremos referenciar este documento apenas por “Padrão C1X”.



Importante

Nas seções a seguir nós resumimos a documentação das funções e bibliotecas utilizadas neste livro. No entanto, nós recomendamos a consulta da documentação original (em inglês).

D.2 `main()`

TODO: Explicar os parâmetros da função `main`.

D.3 `assert.h`



Dica

Mais informações sobre esta função pode ser consultada na seção “7.2 Diagnostics <assert.h>” no padrão C1X da linguagem C. Os seguintes sites também podem ajudá-lo a compreender esta função: http://www.acm.uiuc.edu/webmonkeys/book/c_guide/2.1.html, <http://pt.wikipedia.org/wiki/Assert.h> e <http://www.cplusplus.com/reference/cassert/assert/>.

D.3.1 assert()

```
void assert ( int expressao )
```

Esta função costuma ser utilizado

Ajuda online: <http://www.cplusplus.com/reference/cassert/assert/>

D.4 stdlib.h

D.4.1 exit()

```
void exit( int codigoDeErro )
```

Termina o programa indicando o código de erro. Ver os códigos de erro EXIT_SUCCESS [170] e EXIT_FAILURE [170].

D.4.2 EXIT_SUCCESS

Constante para indicar que não houve erro.

D.4.3 EXIT_FAILURE

Constante para indicar que houve erro na execução do arquivo.

D.4.4 Gerência de memória

D.4.4.1 malloc()

```
#include <stdlib.h>
void *malloc(size_t tamanho);
```

Aloca um espaço de tamanho bytes, o conteúdo do espaço alocado é indefinido.

Retorna NULL se houve erro na alocação ou um ponteiro para o espaço alocado.

D.4.4.2 calloc()

```
void *calloc(size_t quantidade, size_t tamanho);
```

A função calloc, além de alocar espaço na memória, inicializa todo o espaço alocado com 0 (zeros).

D.4.4.3 free()

```
void free(void *ponteiro);
```

A função free causa a desalocação do ponteiro. Se ponteiro for NULL nada acontece. Se ponteiro não for um ponteiro válido ou já foi desalocado anteriormente, o comportamento não é definido.

D.4.4.4 realloc()

```
void *realloc(void *ponteiro, size_t tamanho);
```

Desaloca `ponteiro` e aloca um novo espaço com o tamanho especificado por `tamanho`.

D.5 string.h

D.5.1 strlen()

```
size_t strlen(const char *string);
```

Computa o tamanho de um string.

Retorna tamanho do string, até encontrar o carácter `\0`.

D.5.2 strcmp()

```
int strcmp(const char *primeiroString, const char *segundoString);
```

Compara dois strings.

Retorna um valor maior do que zero se `primeiroString > segundoString`, zero se `primeiroString==segundoString` e um valor negativo de `primeiroString<segundoString`.

D.5.3 strncmp()

```
int strncmp(const char *primeiroString,  
            const char *segundoString, size_t n);
```

Compara dois strings até `n` posições.

D.5.4 strcpy()

```
char *strcpy(char * destino, const char * origem);
```

Copia o conteúdo de um string para outro.

A função retorna `destino`.

D.5.5 strncpy()

```
char *strncpy(char * destino, const char * origem,  
              size_t n);
```

Copia o conteúdo de `origem` para `destino` até no máximo `n` caracteres.

A função retorna `origem`.

D.6 stdbool.h

Inclusão da biblioteca `stdbool.h` [172], que define o tipo `bool`. Este tipo pode ser utilizado como retorno de expressões lógicas. Além disso também estão definidos macro `true` com o valor 1 e `false` com o valor 0.

D.7 stdio.h



Nota

A documentação sobre as funções de leitura de arquivo estão contidas na seção “7.21 Input/output <stdio.h>” do padrão C1X.

D.7.1 fopen()

```
#include <stdio.h>
FILE *fopen(const char * nomeDoArquivo,
            const char * modo);
```

A função `fopen` retorna um ponteiro para `FILE` se conseguir abrir o arquivo, caso contrário retorna `NULL`.

nomeDoArquivo

nome do arquivo que será aberto

mode

Modo de abertura do arquivo.

r

abre arquivo de texto para leitura

w

abre arquivo de texto para escrita

wx

cria arquivo de texto para escrita

a

adiciona ao final; o indicador de posição de arquivo é posicionado no final do arquivo

rb

abre arquivo binário para leitura

wb

abre arquivo binário para escrita

ab

abre arquivo binário para escrita, no final do arquivo

D.7.2 fclose()

```
int fclose(FILE *arquivo);
```

Uma chamada realizada com sucesso invoca o `fflush()` [175] e fecha o arquivo.

A função retorna zero caso o arquivo foi fechado com sucesso, ou `EOF` caso houve erro no fechamento.

D.7.3 fgetc()

```
int fgetc(FILE *arquivo);
```

Ler um carácter do arquivo.

Caso em caso de erro ou não houver mais caracteres, retorna `EOF`.

D.7.4 getchar()

```
#include <stdio.h>
int getchar(void);
```

Ler um carácter da entrada padrão, equivale a `fgetc(stdin)`:

Ver `fgetc()` [173].

D.7.5 fputc()

```
#include <stdio.h>
int fputc(int character, FILE *arquivo);
```

Escreve um carácter no arquivo.

Retorna o carácter escrito. Se houve erro, o indicador de erro é setado e retorna `EOF`.

D.7.6 putchar()

```
#include <stdio.h>
int putchar(int character);
```

Escreve um carácter na saída padrão.

Retorna o carácter escrito. Se houve erro, o indicador de erro é setado e retorna `EOF`.

- Ver `fputc()` [173].

D.7.7 fgetc

```
#include <stdio.h>
char *fgetc(char * string, int n, FILE * arquivo);
```

Ler um string de arquivo e salva o conteúdo em `string`.

Retorna `string` se realizado com sucesso. Se o final do arquivo foi encontrado e não leu nenhum carácter então `string` não é alterado e `NULL` é retornado. Se ocorreu algum erro o valor de `string` é indeterminado e retorna `NULL`.

D.7.8 fputc

```
#include <stdio.h>
int fputc(const char * string, FILE * arquivo);
```

Escreve `string` na posição atual de arquivo. O carácter nulo de término não é escrito.

Retorna `EOF` se houve erro na escrita; caso contrário retorna um valor maior ou igual a zero.

Ver `fgetc()` [174].

D.7.9 fread()

```
#include <stdio.h>
size_t fread(void * ponteiro, size_t tamanho, size_t
    quantidade, FILE * arquivo);
```

A função ler para o buffer `ponteiro` até `quantidade` quantidade de elementos, de tamanho `tamanho` do arquivo `arquivo`. O indicador de posição é avançado de acordo com a quantidade de caracteres lidos.

A função retorna o número de elementos lidos, que pode ser menor do que `quantidade` caso encontrou o final do arquivo ou houve erro. Se `quantidade` ou `tamanho` for zero, o conteúdo de `ponteiro` não é alterado.

D.7.10 fwrite()

```
#include <stdio.h>
size_t fwrite(const void * ponteiro, size_t tamanho, size_t
    quantidade, FILE * arquivo);
```

A função escreve na posição apontada por `ponteiro`, até a `quantidade` quantidade de elementos do tamanho `tamanho` no arquivo `arquivo`. O indicador de posição é incrementado de acordo com a quantidade de bytes escritos.

A função retorna o número de elementos escritos, que pode ser menor do que `quantidade` caso houve erro. Se `quantidade` ou `tamanho` for zero, nada é escrito no arquivo.

D.7.11 fflush()

```
#include <stdio.h>
int fflush(FILE *arquivo);
```

Causa a escrita de qualquer dado que ainda não foi escrito no arquivo.

Retorna EOF se houve erro na escrita ou zero se a escrita foi realizada com sucesso.

D.7.12 fseek()

Move o indicador de posição do arquivo.

```
#include <stdio.h>
int fseek(FILE *arquivo, long int deslocamento, int whence);
```

Ver <http://www.cplusplus.com/reference/cstdio/fseek/>.

D.7.13 ftell()

Retorna a posição atual no arquivo.

```
#include <stdio.h>
long int ftell(FILE *arquivo);
```

Ver <http://www.cplusplus.com/reference/cstdio/ftell/>, [fseek\(\)](#) [175].

D.7.14 rewind()

```
#include <stdio.h>
void rewind(FILE *stream);
```

Retroce o indicador de posição para o início do arquivo. É equivalente a:

```
(void) fseek(arquivo, 0L, SEEK_SET)
```

D.7.15 fscanf()

```
#include <stdio.h>
int fscanf(FILE * arquivo,
    const char * formato, ...);
```

D.7.16 scanf()

Ver [fscanf\(\)](#) [175].

D.7.17 printf()

Ver fprintf() [176].

D.7.18 fprintf

```
#include <stdio.h>
int fprintf(FILE * arquivo,
    const char * formato, ...);
```

D.7.18.1 Flags

–

O resultado é justificado a esquerda, se não for especificado é justificado a direita.

+

O resultado da conversão sempre será prefixado com o sinal do número (positivo ou negativo).

espaço

Se o resultado for prefixado com *espaço* então ele será impresso.

0

Imprime zeros a esquerda do número.

#

O Resultado é apresentado utilizando um *formato alternativo*.

D.7.18.2 Modificadores de tamanho

Os modificadores de tamanho especificam qual o tamanho será utilizado na conversão. O tamanho independe do sinal (*signed* ou *unsigned*).

Modificador	Tamando do tipo
hh	char
h	short
l	long
ll	long long
L	long double

D.7.18.3 Especificadores de Conversão de tipos

d, i (tipo decimal com sinal)

O argumento int é convertido para um decimal com sinal no estilo [-]dddd. A precisão especifica o número mínimo de dígitos a aparecer; se o valor convertido pode ser representado com menos dígitos, ele é expandido com zeros. A precisão padrão é 1. O resultado da conversão de Zero com a precisão de Zero é nenhum carácter.

u, o, x, X (tipo decimal sem sinal, octal, ou hexadecimal)

O argumento `unsigned int` é convertido para `unsigned octal (o)`, `unsigned decimal (u)`, ou notação `unsigned hexadecimal (x or X)` no estilo `dddd`; as letras `abcdef` (de hexadecimal) são usadas na conversão com `x` e `ABCDEF` para conversão com `X`. A precisão especifica o número mínimo de dígitos a aparecer; se o valor a ser convertido pode ser representado com menos dígitos, ele é expandido com zeros a esquerda. A precisão padrão é 1. O resultado da conversão de Zero com a precisão de Zero é nenhum carácter.

f, F (tipo double)

O argumento `double` representando o número de ponto flutuante é convertido para a notação decimal no estilo `[-]ddd.ddd`, em que o número de dígitos depois do carácter ponto é igual a especificação da precisão. Se a precisão não for especificada, é assumido 6; se a precisão for zero e a flag `#` não for utilizada, o ponto não irá aparecer. Se o ponto foi especificado então pelo menos um dígito é impresso antes dele. O valor é arredondado para o número de dígitos apropriados.

c (tipo carácter)

O argumento inteiro é convertido para `unsigned char`, e o carácter resultante é escrito.

s (tipo string)

O argumento deve um ponteiro para o início de um array de `char`. Os caracteres do array serão impressos até encontrar o carácter null `\0` (que não será impresso).

`%`

O carácter `%` é escrito. A especificação completa deve ser `“%%”`.

D.7.18.4 Resumo e utilização do formato

A Figura D.1 [178] resume o formato utilizado nas funções `fprintf()` [176] e `printf()` [176].

**Nota**

Na plataforma em que esta figura foi criada os tipos `long int` e `int` possuem o mesmo tamanho.

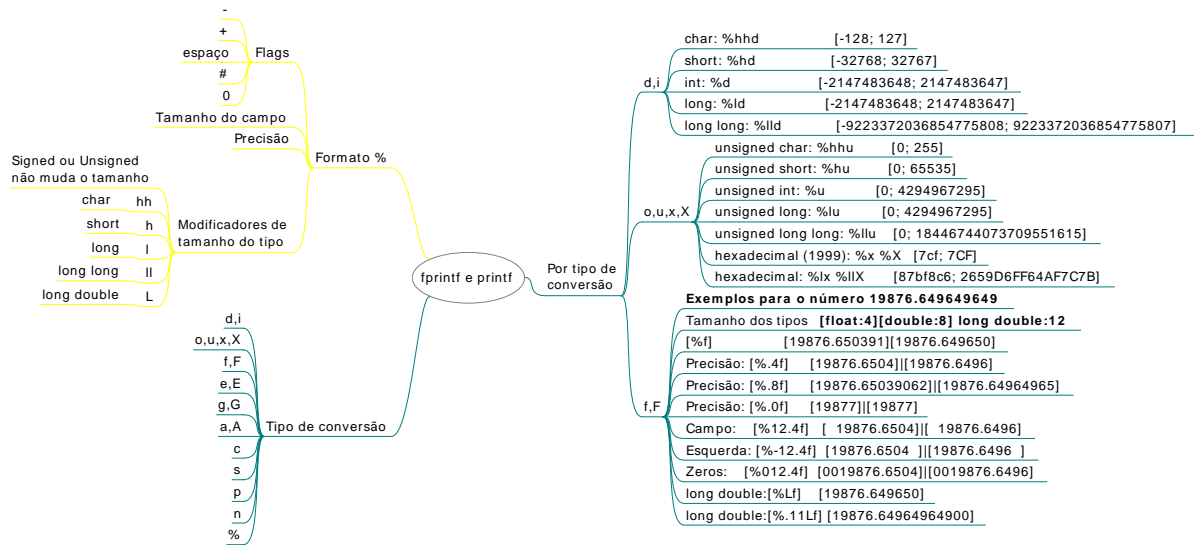


Figura D.1: Resumo do formato de fprintf e printf

Exemplo D.1 Exemplo de utilizações do formato de printf**Código fonte** code/bibliotecas/aprendendo_printf.c

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

int main() {
    long double meuLD = 19876.649649649;
    double meuDouble = (double) meuLD;
    float meuFloat = meuDouble;

    printf(
        "Tamanho dos tipos\t float:%d\t double:%d\t long double:%d\n",
        sizeof (float), sizeof (double), sizeof (long double));

    printf("[%f] [%f]\n", meuFloat, meuDouble);
    printf("Precisão: [%4f] [%4f]\n", meuFloat,
        meuDouble);
    printf("Precisão: [%8f] [%8f]\n", meuFloat,
        meuDouble);
    printf("Precisão: [%0f] [%0f]\n", meuFloat,
        meuDouble);

    printf("Campo: [%12.4f] [%12.4f]\n", meuFloat,
        meuDouble);
    printf("Esquerda: [%-12.4f] [%-12.4f]\n", meuFloat,
        meuDouble);
    printf("Zeros: [%012.4f] [%012.4f]\n", meuFloat,
        meuDouble);
}
```

```

printf("long double:[%Lf]      [%Lf]\n", meuLD);
printf("long double:[%.11Lf] [%.11Lf]\n", meuLD);

// Inteiros

printf("Tamanho dos tipos\tchar:%d\tshort:%d\tint:%d\tlong:%d\t ←
      tlong long:%d\n", sizeof (char), sizeof (short), sizeof (int), ←
      sizeof (long), sizeof (long long));
printf("Inteiros\n");
printf("char: %hhd          [%hhd; %hhd]\n",
      SCHAR_MIN, SCHAR_MAX);
printf("short: %hd          [%hd; %hd]\n",
      SHRT_MIN, SHRT_MAX);
printf("int: %d             [%d; %d]\n",
      INT_MIN, INT_MAX);
printf("long: %ld           [%ld; %ld]\n",
      LONG_MIN, LONG_MAX);
printf("long long: %lld      [%lld; %lld]\n",
      LLONG_MIN, LLONG_MAX);

printf("unsigned char: %hhu   [%hhu; %hhu]\n",
      (unsigned char)0, UCHAR_MAX);
printf("unsigned short: %hu   [%hu; %hu]\n",
      (unsigned short)0, USHRT_MAX);
printf("unsigned int: %u       [%u; %u]\n",
      (unsigned int)0, UINT_MAX);
printf("unsigned long: %lu     [%lu; %lu]\n",
      0UL, ULONG_MAX);
printf("unsigned long long: %llu [%llu; %llu]\n",
      0ULL, ULLONG_MAX);

printf("hexadecimal (1999): %x %X [%x; %X]\n", 1999, 1999);
printf("hexadecimal: %lx %lX [%lx; %lX]\n",
      142342342ul, 2763476238762736763ul);

return EXIT_SUCCESS;
}

```

Saída da execução

```

Tamanho dos tipos    float:4      double:8     long double:12
[%f]                 [19876.650391][19876.649650]
Precisão: [% .4f]     [19876.6504]| [19876.6496]
Precisão: [% .8f]     [19876.65039062]| [19876.64964965]
Precisão: [% .0f]     [19877]| [19877]
Campo: [%12.4f]       [ 19876.6504]| [ 19876.6496]
Esquerda: [%-12.4f]   [19876.6504  ]| [19876.6496  ]
Zeros: [%012.4f]      [0019876.6504]| [0019876.6496]
long double:[%Lf]     [19876.649650]
long double:[%.11Lf]  [19876.64964964900]
Tamanho dos tipos    char:1  short:2 int:4   long:4   long long:8
Inteiros
char: %hhd           [-128; 127]

```

short: %hd	[-32768; 32767]
int: %d	[-2147483648; 2147483647]
long: %ld	[-2147483648; 2147483647]
long long: %lld	[-9223372036854775808; ↵ 9223372036854775807]
unsigned char: %hhu	[0; 255]
unsigned short: %hu	[0; 65535]
unsigned int: %u	[0; 4294967295]
unsigned long: %lu	[0; 4294967295]
unsigned long long: %llu	[0; 18446744073709551615]
hexadecimal (1999): %x %X	[7cf; 7CF]
hexadecimal: %lx %llX	[87bf8c6; 2659D6FF64AF7C7B]

D.7.19 feof

```
#include <stdio.h>
int feof(FILE *arquivo);
```

Testa se o indicador do final de arquivo de `arquivo`.

Retorna um valor não zero se o indicador de final de arquivo foi setado em `arquivo`.

D.8 math.h

Funções matemáticas.

D.8.1 M_PI

```
#include <math.h>
#define M_PI 3.14159265358979323846
```

`M_PI` é uma macro que contém uma definição de π .

D.8.2 fabs()

```
#include <math.h>
double fabs(double x);
float fabsf(float x);
long double fabsl(long double x);
```

Calcula módulo ou valor absoluto de `x`: $|x|$.

D.8.3 pow()

```
#include <math.h>
double pow(double x, double y);
float powf(float x, float y);
long double powl(long double x, long double y);
```

Retorna x elevado a y: x^y .

D.8.4 sqrt()

```
#include <math.h>
double sqrt(double x);
float sqrtf(float x);
long double sqrtl(long double x);
```

Calcula a raiz quadrada de x: \sqrt{x} .

Apêndice E

Respostas das atividades

Nesta capítulo apresentamos as respostas de algumas atividades.



Nota

Você pode contribuir para elaboração desta seção enviando suas respostas.



Dica

Na sala de aula, nos fóruns e mensagens recomendamos a utilização do site <https://gist.github.com> para compartilhar códigos e tirar dúvidas sobre eles.

E.1 Capítulo 1

4

O programa precisa ler um CPF, salvá-lo numa variável do tipo `long long` e em seguida imprimir seu conteúdo.

Código fonte

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

/** O propósito deste programa é demonstrar a leitura e impressão
 * de um CPF, salvando-o em variável do tipo 'long long'.
 */

int main(void) {
    long long cpf; // variável long long, onde será salvo o cpf

    printf("Digite um CPF, somente os números: ");
    scanf("%lld", &cpf); // ler o cpf, utilizando o formato %lld

    // Imprime o cpf, incluindo zeros a esquerda, se necessário.
```

```
printf("CPF lido: %011lld \n", cpf);

return EXIT_SUCCESS;
}
```

Saída da execução

Digite um CPF, somente os números: 05814840536
CPF lido: 05814840536

E.2 Capítulo 2

3d

Não existe resposta única. Nesta resposta optamos por criar dois registros, um para organizar o **Agendamento** e outro para agrupar os campos do **Horário**. As tabelas a seguir mostram o resultado da análise.

Novo tipo	Campo	Tipo do campo
Compromisso	codigo ★	Numérico
	titulo	Textual
	local	Textual
	horario	Horario
	detalhes	Textual

Exemplos de outros campos que poderiam existir são: tipo do compromisso, recorrência do compromisso, participantes, horário de termino etc.

No tipo de registro **Compromisso**, nossa primeira tentativa de definir um campo identificador foi de utilizar `titulo`, mas percebemos que poderia haver mais de um compromisso com o mesmo título. Outro possível campo seria o `horario`, no entanto é possível existir dois compromissos com o mesmo horário. Portanto criamos um campo `codigo` para identificar os registros.

Novo tipo	Campo	Tipo do campo
Horario	ano ★	Numérico
	mes ★	Numérico
	dia ★	Numérico
	hora ★	Numérico
	minuto ★	Numérico

No tipo **Horario** percebemos que todos os campos juntos identificam um registro. Caso dois registros possuam os mesmos valores então eles representam um mesmo **Horario**.

Código fonte

Tipos dos registros **Compromisso** e **Horario**.

```
typedef struct {
    short ano;
    short mes;
```

```
    short dia;  
    short hora;  
    short minuto;  
} Horario;  
  
typedef struct {  
    int codigo;  
    char titulo[100];  
    char local[100];  
    char detalhes[255];  
    Horario horario;  
} Compromisso;
```

Capítulo 7

Índice Remissivo

A

abrindo, 83
Algoritmo do Avestruz, 123
Algoritmos, 2
algoritmos, 2
Alocação dinâmica, 68
alocação dinâmica, 4, 66
alocação estática, 4, 66
Arquivo, 82
 abrindo, 83
 binário, 81
 especial, 81
 fechando, 85
 final de arquivo, 180
 indicador de erro, 86
 indicador final de arquivo, 87
 indicadores, 86
 posição atual, 175
 retroceder, 175
 somente texto, 81
Arquivos especiais, 161
assert, 108, 170
assert.h, 169

B

binário, 81

C

calloc, 141, 170
Campo Identificador, 22
codificação, 107, 122

D

Depuração, 155
Depurar, 119

E

escopo de uma função, 56
especial, 81

estático, 66

estruturas de controle, 5
estruturas de dados, 4
exit, 170
EXIT_FAILURE, 170
EXIT_SUCCESS, 170

F

fabs, 180
fclose, 85, 173
fechando, 85
feof, 180
fflush, 93, 175
fgetc, 87, 173
fgets, 89, 141, 174
final de arquivo, 180
flags de compilação, 156
Fluxo de dados, 81
fopen, 83, 172
fprintf, 94, 176, 177
fputc, 87, 173
fputs, 89, 174
fread, 91, 174
free, 68, 170
fscanf, 94, 175
fseek, 95, 175
ftell, 94, 175
função, 56
fwrite, 91, 174

G

getchar, 87, 173

I

implementação inocente, 100
indicador de erro, 86
indicador de posição, 94
indicador final de arquivo, 87
indicadores, 86

indireção, 52

L

Lingua do i, 99

M

módulo, 180

main, 169

Makefile, 155

malloc, 68, 170

math.h, 180

memória dinâmica, 66

O

operador de endereço, 52, 53, 55

operador de indireção, 53, 55

P

parâmetros por referência, 59

passagem de parâmetros por valor, 59

passagem de valores por referência, 60

passagem por parâmetro, 59

passagem por valor, 58

persistência, 21

Pi, 180

pilha de execução, 56, 57

Pipe, 163

Ponteiro, 52

 indireção, 52

ponteiro, 55

ponteiros, 60

posição atual, 175

potência, 181

pow, 181

printf, 94, 176, 177

protótipo da função, 61

putchar, 87, 173

R

raiz quadrada, 181

realloc, 68, 171

Registro, 20

 sintaxe, 22

relacionamento, 28

retroceder, 175

rewind, 94, 175

S

scanf, 94, 175

sintaxe, 22

somente texto, 81

sqrt, 181

stdbool.h, 172

stderr, 85

stdin, 85

stdio.h, 172

stdlib.h, 170

stdout, 85

strcmp, 171

strcpy, 171

string.h, 171

strlen, 171

strncmp, 171

strncpy, 171

T

Tipo de dado, 22

Tipos de dados, 5

V

variáveis, 5

variáveis locais, 56

variável local, 57