

---

## Programação de Computadores

---



Reitor

Prof. Dr. Targino de Araújo Filho

Pró-Reitora de Graduação

Profa. Dra. Emília Freitas de Lima



Coordenação UAB-UFSCar

Prof. Dr. Daniel Mill

Profa. Dra. Denise Abreu-e-Lima

Profa. Dra. Valéria Sperduti Lima

Profa. Dra. Joice Lee Otsuka

Coordenação do curso de Sistemas de Informação

Profa. Dra. Sandra Abib (Coordenadora)

Profa. Dra. Wanda Hoffmann (Vice-Coordenadora)

---

Ricardo Rodrigues Ciferri

---

# Programação de Computadores

---



São Carlos, 2009

© 2009, Ricardo Rodrigues Ciferri. Todos os Direitos Reservados.  
reimpressão em 06/04/2009

Livro destinado ao curso de Bacharelado em Sistemas de Informação, UAB-UFSCar, EaD (Educação a Distância).

Concepção e Produção Editorial  
Prof. Dr. Daniel Mill

Responsáveis pela Preparação e Revisão  
Ms. Gislaine Cristina Micheloti Rosales (designer/projetista)  
Douglas H. Perez Pino (revisor)

Arte da Capa, Designer e Imagens  
Jorge Oliveira

Editoração, diagramação eletrônica  
Rodrigo Rosalis

UAB-UFSCar  
Universidade Federal de São Carlos  
Rodovia Washington Luís, km 235  
13.565-905 - São Carlos - São Paulo - Brasil  
Tel.: (16) 3351 8420 - [www.uab.ufscar.br](http://www.uab.ufscar.br)

---

Todos os direitos reservados. Nenhuma parte desta obra pode ser reproduzida ou transmitida por qualquer forma e/ou quaisquer meios (eletrônicos ou mecânicos, incluindo fotocópia e gravação) ou arquivada em qualquer sistema de banco de dados sem permissão escrita do titular do direito autoral.

# SUMÁRIO

---

Ficha da disciplina .....	7
Unidade 1: Introdução à Disciplina	
1.1 Orientações Gerais .....	13
1.2 Conceitos Básicos .....	18
1.2.1 Organização Básica de um Computador .....	18
1.2.2 Sistemas Operacionais .....	20
1.2.3 Linguagens de Programação .....	22
1.3 Ambiente de Desenvolvimento Integrado Dev-C++ .....	24
1.4 Lista de Compiladores C .....	38
1.5 Conceitos Básicos de Programação .....	41
1.5.1 Estrutura Geral de um Programa .....	41
1.5.2 Exemplo de Programa 1 .....	44
1.5.3 Exemplo de Programa 2 .....	54
1.5.4 Explicações Adicionais .....	57
1.5.5 Mapeamento de Algoritmo para Programa C .....	61
1.5.6 Observações Fianis sobre a Linguagem C .....	65
Unidade 2: Bases Numéricas, Expressões e Comandos Condicionais	
2.1 Bases Numéricas .....	69
2.1.1 Mudança de Base .....	71
2.1.2 Representação na Linguagem C .....	74
2.2 Expressões .....	75
2.2.1 Expressões e Operadores Aritméticos .....	75
2.2.2 Expressões e Operadores Relacionais .....	82
2.2.3 Expressões e Operadores Lógicos .....	88
2.2.4 Explicações Adicionais .....	94
2.2.5 Mapeamento de Algoritmo para Programa C .....	102
2.3 Comandos Condicionais .....	107
2.3.1 Comando Condicional if-else .....	108
2.3.2 Comando switch .....	115
2.3.3 Explicações Adicionais .....	118
2.3.4 Mapeamento de Algoritmo para Programa C .....	122
Unidade 3: Comandos de Repetição	
3.1 Comandos de Repetição .....	131
3.1.1 Comando while .....	133
3.1.2 Comando do-while .....	138
3.1.3 Comando for .....	143

3.2 Explicações Adicionais .....	148
3.3 Mapeamento de Algoritmo para Programa C .....	154

#### Unidade 4: Ponteiros, Registros e Funções

4.1 Ponteiros .....	161
4.2 Registros .....	167
4.3 Funções .....	176
4.4 Explicações Adicionais .....	194
4.5 Mapeamento de Algoritmo para Programa C .....	209

#### Unidade 5: Vetores, Matrizes e Arranjos de Registros

5.1 Arranjos Unidimensionais .....	227
5.2 Arranjos Bidimensionais .....	236
5.3 Arranjos de Estruturas .....	243
5.4 Explicações Adicionais .....	248
5.5 Mapeamento de Algoritmo para Programa C .....	254

## Ficha da disciplina

---

### Professor responsável pela disciplina

Ricardo Rodrigues Ciferri é docente do Departamento de Computação da Universidade Federal de São Carlos desde fevereiro de 2006. Obteve seu título de Doutor em Ciência da Computação pela Universidade Federal de Pernambuco em 2002. Entre março de 1996 e fevereiro de 2006 foi docente do Departamento de Informática da Universidade Estadual de Maringá. No ensino de graduação, o Prof. Ricardo já lecionou diversas disciplinas, dentre as quais pode-se citar: Algoritmos e Estruturas de Dados, Programação de Computadores, Sistemas Operacionais, Banco de Dados e Organização e Recuperação da Informação. Quanto à pesquisa, o Prof. Ricardo participou de projetos financiados pelo CNPq, MCT e Fundação Araucária. As suas áreas de atuação na pesquisa são: Bancos de Dados, Sistemas de Informações Geográficas, Data Warehousing e Bioinformática.

### Objetivos de aprendizagem:

A disciplina tem por objetivo geral capacitar o aluno a resolver problemas de programação com o auxílio de uma linguagem de programação estruturada que seja adequada para fins didáticos. Ao final da disciplina, o aluno deverá estar apto a usar as principais características de uma linguagem de programação estruturada para resolver problemas de programação. Mais especificamente, a disciplina tem por objetivo fornecer ao aluno conhecimentos relativos à estrutura básica de um computador e à codificação de algoritmo para programa com uso das principais funcionalidades de uma linguagem de programação estruturada, a saber: tipos de dados, constantes, variáveis, expressões, comandos de entrada e saída, comando de atribuição, comandos condicionais, comandos de repetição, funções básicas, sub-rotinas, estruturas compostas de dados e ponteiros. A linguagem de pro-

gramação estruturada escolhida para esta disciplina é a linguagem C.

**Ementário:**

A disciplina tem por objetivo abordar problemas de programação de diferentes complexidades com o uso de uma linguagem de programação cujas características obedeçam a fins didáticos. Além disso, a disciplina visa capacitar o aluno a implementar na linguagem escolhida os algoritmos desenvolvidos na disciplina "Construção de Algoritmos".

**Visão geral da disciplina:**

Esta disciplina é de vital importância no processo de formação do aluno, desde que o aluno terá que programar para resolver problemas na maioria das disciplinas subseqüentes do Curso. Esta disciplina também é importante no sentido de permitir a fixação do conteúdo apresentado na disciplina "Construção de Algoritmos". A disciplina será desenvolvida ao longo do período na forma de unidades seqüenciais. Cada unidade possuirá um conteúdo que abordará um assunto específico da disciplina. Dependendo da complexidade do assunto apresentado, uma unidade poderá ser coberta em várias semanas, ou mesmo algumas unidades sendo abordadas em um intervalo de tempo mais reduzido. Cada unidade corresponderá a uma aula virtual. Os alunos devem seguir a seqüência das unidades e somente poderão avançar para uma unidade posterior caso as atividades propostas tenham sido cumpridas.

**Conteúdo da disciplina:**

Esta disciplina terá o seguinte conteúdo:

- Apresentação dos alunos, professor e tutores virtuais;
- Introdução e apresentação da disciplina;
- Conceitos básicos de computadores, linguagens de programação, ambiente de programação, editores de texto e sistemas operacionais;



- Conceito de programa e introdução à linguagem de programação estruturada;
- Conceitos de tipos de dados, constantes, constantes simbólicas e variáveis;
- Estruturação básica de programas: seqüência de comandos, comandos de entrada e saída e comando de atribuição;
- Introdução ao uso de funções predefinidas;
- Expressões aritméticas, literais, relacionais e lógicas;
- Conceito de bases numéricas;
- Introdução ao uso de bibliotecas;
- Comandos condicionais simples e completos;
- Comandos condicionais aninhados;
- Comando switch (estrutura de seleção múltipla);
- Comandos de repetição;
- Ponteiros e endereçamento;
- Conceitos de dados heterogêneos: declaração, uso e acesso a campos de registros;
- Sub-rotinas: funções. Conceitos de escopo, variáveis locais e globais. Conceitos de passagem de parâmetros por valor e por referência. Conceito de valor de retorno;
- Arranjos e matrizes; e
- Arranjos e matrizes de registros.



# Unidade 1

---

## Introdução à Disciplina

---



## 1.1 Orientações Gerais

A disciplina “Programação de Computadores” está organizada em cinco **ciclos de aprendizagem**. O primeiro ciclo é uma introdução à disciplina. Neste ciclo são descritos conceitos básicos de programação na linguagem C. O segundo ciclo enfoca a utilização de expressões e comandos condicionais, além de discutir brevemente o conceito de bases numéricas. O terceiro ciclo é voltado totalmente para o aprendizado do uso de comandos de repetição. O quarto ciclo trata de conceitos mais avançados da linguagem C, a saber: ponteiros, registros (estruturas) e funções (sub-rotinas). A disciplina encerra-se no quinto ciclo com o estudo de arranjos unidimensionais, bidimensionais e arranjos de estruturas. Após o período regular de cinco ciclos de aprendizagem, haverá um período dedicado especialmente para a **recuperação de aprendizagem**.

Cada ciclo de aprendizagem é constituído por quatro **macro-atividades**. Haverá macro-atividades **obrigatórias** para todos os alunos. Haverá também macro-atividades **opcionais**, nas quais a participação do aluno é facultativa, mas fortemente recomendada. Haverá ainda uma macro-atividade voltada para a **recuperação de aprendizagem** dentro do próprio ciclo. Macro-atividades de recuperação serão obrigatórias apenas para os alunos que não alcançarem um desempenho satisfatório no aprendizado dos conceitos apresentados no ciclo (i.e., não tirarem nota média 6,0 nas atividades AA descritas posteriormente neste texto). A Tabela 1 lista as macro-atividades presentes em cada ciclo de aprendizagem.

número	macro-atividade	tipo
	Material Básico: apostila da disciplina.	
1	O professor poderá marcar aulas de web-conferência em função das dúvidas dos alunos para complementar o aprendizado da apostila.	Leitura obrigatória.
2	Material Complementar: livro-texto da disciplina e demais livros da bibliografia.	Esta macro-atividade terá tanto atividades obrigatórias quanto atividades opcionais. O tipo de cada atividade estará indicado na agenda do ciclo no ambiente virtual de aprendizagem Moodle.
3	Atividades no Moodle: fórum de discussão, questionário, chat (bate-papo), lista de exercícios de programação, wikis, glossários, etc.	Esta macro-atividade terá tanto atividades obrigatórias quanto atividades opcionais. O tipo de cada atividade estará indicado na agenda do ciclo no ambiente virtual de aprendizagem Moodle.
4	Atividades no Moodle: exercícios de recuperação	Recuperação de Aprendizagem

Tabela 1. Macro-atividades.

A **apostila da disciplina** terá um capítulo para cada ciclo de aprendizagem. A apresentação do conteúdo nos capítulos será efetuada da seguinte forma. Inicialmente, por meio de um ou mais programas de exemplo, conceitos serão ilustrados e explicados. Para facilitar o acompanhamento da explicação, os programas estarão numerados por linha. Em seguida, os principais conceitos serão reforçados com uma explicação adicional acompanhada de exemplos de trechos de programas. Adicionalmente, exemplos de

mapeamento de algoritmos para programas na linguagem C serão descritos e explicados. Em uma próxima etapa, o aluno será direcionado para realizar leituras complementares em livros e links da Web. Após, o aluno realizará atividades no Moodle para avaliar o seu aprendizado. Algumas destas atividades serão pontuadas e comporão a nota final do aluno na disciplina. Por fim, para os alunos com aproveitamento insuficiente nas atividades do ciclo de aprendizagem, exercícios de recuperação estarão disponíveis no Moodle.

Cada macro-atividade será composta por um conjunto de atividades. O aluno deve realizar primeiramente as atividades do ciclo 1 e somente após finalizar estas atividades deverá realizar as atividades do ciclo 2 e assim sucessivamente com relação às atividades dos ciclos 3, 4 e 5. Portanto, os ciclos devem ser estudados na sequência. Dentro de cada ciclo, o aluno deve realizar as atividades em ordem crescente de numeração, iniciando com a atividade 1 (Atv.01) e terminando na última atividade (exemplo, Atv. 14).

As atividades seguirão a seguinte nomenclatura:

- **Atividade Avaliativa (AA):** são as atividades nas quais o aluno será avaliado pela correção (i.e., se a resposta está correta) e qualidade de suas respostas. Estas atividades terão peso no cálculo da média final. Estas atividades são **obrigatórias** para todos os alunos;
- **Atividade de Participação Avaliativa (APA):** são as atividades nas quais não será observado se o aluno respondeu corretamente ou não, somente se ele participou. Estas atividades valem nota de participação e portanto terão peso no cálculo da média final. Estas atividades são **obrigatórias** para todos os alunos;
- **Atividade Presencial (AP):** são as provas escritas presenciais no pólo. Estas atividades terão grande peso no cálculo da média final. Estas atividades são **obrigatórias** para todos os alunos;
- **Atividade Teórica (AT):** são as atividades de leitura de material, as atividades de assistir arquivos multimídia (e.g., plano de ensino, vídeo de apresentação do professor, apostila, livro-texto da disciplina e demais livros da bibliografia) e aulas de web-confe-

rência. Estas atividades não irão compor o cálculo da média final. Algumas atividades serão obrigatórias e outras serão opcionais;

- **Atividade Suplementar (AS):** são as atividades extras que o aluno poderá enviar para o tutor e solicitar a correção. Porém, estas atividades não irão compor o cálculo da média final. Estas atividades são **opcionais**; e

- **Atividade de Recuperação Paralela (AR):** são as atividades propostas ao final de cada ciclo de aprendizagem e que servirão como recuperação integral somente das atividades AA.

- **Atividades Substitutivas:** são atividades que podem ser escolhidas para substituir outras atividades (AA, AR ou AS). Por exemplo, para a atividade 12 do ciclo 2, o aluno poderá escolher entre a atividade regular (i.e., resolver exercícios do livro-texto “Primeiro Curso de Programação em C” de Edson Senne) e a atividade substitutiva (i.e., fazer um programa que verifique se uma data é válida). O aluno deve resolver apenas uma das atividades (atividade regular ou atividade substitutiva). O local para submeter a resposta no Moodle é o mesmo, ou seja, no mesmo link (Atv.xx).

Com relação aos prazos para cumprir as atividades dos ciclos, o aluno deve notar que os prazos são bastante flexíveis e extensos. Porém, o aluno deve se organizar em função de sua disponibilidade de tempo para cumprir as atividades nos prazos estipulados.

As únicas atividades com data e horário fixos são:

- **Chat (atividade APA):** atividade realizada semanalmente em dia da semana e horário previsto no plano de ensino.

- **Aula de web-conferência (atividade AT):** atividade que poderá ser agendada em dia e horário específico para tratar de assuntos de cada um dos ciclos de aprendizagem.

- **Atividade Presencial (atividade AP):** provas escritas nos pólos com datas e horários fixos. A prova 1 abordará os conceitos estudados nos ciclos 1, 2 e 3. A prova 2 abordará os conceitos estudados nos ciclos 4 e 5. A prova REC abordará todo o conteúdo da apostila (ciclos 1 a 5).

Para que os tutores virtuais possam corrigir as atividades AA, AS e AR, o aluno deve enviar uma mensagem, usando o correio



interno do Moodle, para o tutor com o assunto “ciclo x Atv.yy realizada” (exemplo: “ciclo 1 Atv.11 realizada”). Uma vez recebida a mensagem, o tutor corrigirá a atividade e depois colocará a nota e os comentários da correção no link da atividade no Moodle. Uma vez finalizadas todas as atividades AA de um ciclo, o tutor informará o aluno, pelo correio interno do Moodle, se o aluno obteve nota média ou se terá que realizar a atividade AR do ciclo.

Por fim, esta apostila tem por objetivo oferecer um suporte básico para a disciplina “Programação de Computadores” do curso de Bacharelado em Ciência da Informação da Universidade Aberta do Brasil – UFSCar. O material contido nesta apostila é **básico** e tem como objetivo apenas guiar o aluno pelos vários assuntos que serão abordados durante o decorrer da disciplina. Os tópicos são cobertos aqui de modo mais simplificado e, muitas vezes, incompleto. Portanto, a apostila **não deve** ser usada como material único para os seus estudos, mas sim como uma primeira leitura para o aprendizado dos conceitos, conforme constante na agenda de cada ciclo.

Os alunos que chegam a um curso universitário possuem, geralmente, formação heterogênea. Alguns alunos já tiveram contato com computação e com alguma linguagem de programação, enquanto outros alunos não tiveram nenhum contato. Esta disciplina é voltada a **todos os alunos** e, portanto, cada aluno usará o material da apostila de forma diferente, de acordo com os seus conhecimentos prévios. Assume-se que o aluno não precisa ter conhecimento algum para entender os conceitos da apostila.

## 1.2 Conceitos Básicos

Nesta seção, você aprenderá sobre os seguintes **conceitos introdutórios**: organização básica de um computador, sistemas operacionais e linguagens de programação.

### 1.2.1 Organização Básica de um Computador

Atualmente, você encontrará em uma loja de informática diversas configurações e marcas de computador. Você possivelmente encontrará desde computadores mais simples e baratos, por exemplo computadores voltados ao processamento de dados em um escritório, até computadores mais sofisticados e caros, por exemplo computadores projetados especificamente para o mercado de jogos e processamento gráfico. Em algumas lojas, você também encontrará configurações de alto desempenho, tais como servidores de banco de dados.

O entendimento do funcionamento completo de um computador disponível em uma loja de informática é muito complexo e está fora do escopo desta disciplina. Ao invés disso, você aprenderá sobre o funcionamento de um computador simplificado, cujo princípio de funcionamento é similar ao funcionamento de computadores reais. Formalmente, nós chamaremos este computador simplificado de **modelo de computador**. A Figura 1 ilustra o modelo de computador que usaremos nesta disciplina, o qual possui os seguintes componentes: processador, memória primária (também conhecida como memória principal, RAM – random access memory ou simplesmente memória), dispositivos de entrada/saída de dados e a interface de entrada/saída de dados.

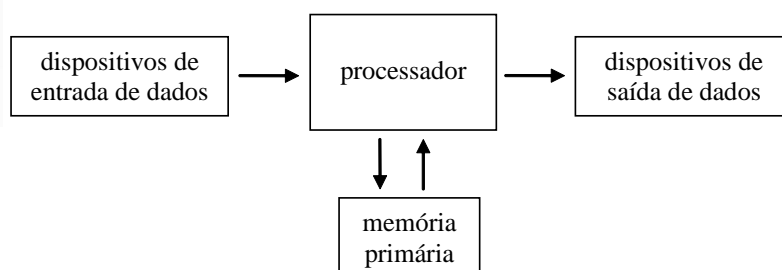


Figura 1. Modelo de Computador.

O processador de um computador é responsável pela transformação de **dados de entrada** em **dados de saída**. Portanto, este componente possui capacidade de processamento de dados. O processador é considerado o cérebro do computador e permite a computação de, por exemplo,  $10 + 20 = 30$ . Um processador pode realizar milhões e até bilhões de operações por segundo.

Para realizar qualquer processamento ou cálculo, o processador precisa ler dados de entrada (por exemplo, os valores 10 e 20) e escrever dados de saída (por exemplo, a soma 30). Estes dados de entrada e de saída ficam armazenados dentro do computador na memória primária. Esta memória possui quatro características básicas. A primeira característica refere-se à sua capacidade de armazenamento. A memória primária de um computador desktop (i.e., de um computador de mesa usado em ambientes de escritório e doméstico) possui uma capacidade de armazenamento entre 256 MB (i.e., milhões de bytes) e 4 GB (i.e., bilhões de bytes). Um byte corresponde a menor unidade de armazenamento na memória. Um caractere ocupa tipicamente 1 byte na memória primária. A segunda característica refere-se à sua volatilidade. Os dados da memória primária são perdidos na ausência de energia elétrica, ou seja, os dados são perdidos depois que o computador é desligado. A terceira característica é a sua velocidade ou tempo de acesso a memória. O tempo de acesso é medido em nanossegundos (ns) que corresponde a  $10^{-9}$  segundos. O tempo de acesso da memória primária é relativamente rápido quando comparado com o tempo de acesso de outros meios de armazenamento, tais como discos magnéticos e dispositivos óticos (CD e DVD). A quarta característica da memória primária é que o tempo de acesso a qualquer posição da memória é o mesmo, ou seja, é constante. Esta última característica simplifica o planejamento do acesso aos dados, pois pode-se armazenar em qualquer posição sem que isto afete o desempenho na recuperação dos dados.

Para você conseguir usar um computador e depois visualizar os resultados produzidos, é necessário utilizar dispositivos de entrada/saída de dados. Os dados são tipicamente passados para um computador pelo teclado, o qual é considerado o dispositivo

de entrada de dados padrão. No entanto, os dados de entrada também podem ser obtidos a partir, por exemplo, do mouse ou de um scanner. Já os dados produzidos por um computador são comumente visualizados em um monitor de vídeo, o qual é considerado o dispositivo de saída de dados padrão. Outro dispositivo de saída de dados bastante utilizado é a impressora. Em particular, discos magnéticos podem atuar tanto como dispositivos de entrada de dados quanto como dispositivos de saída de dados. Discos magnéticos (também conhecidos como hard disks ou winchesters) são usados para armazenamento persistente de dados (i.e., dados não voláteis) e são classificados como memória secundária.

Por fim, a interface de entrada/saída de dados possibilita a comunicação entre os dispositivos de entrada/saída de dados e o processador. Na Figura 1, esta interface é representada pela seta dos dispositivos de entrada de dados para o processador e pela seta do processador para os dispositivos de saída de dados.

### 1.2.2 Sistemas Operacionais

O gerenciamento dos **recursos computacionais** (por exemplo, processador, memória primária e discos magnéticos) é realizado por um programa especial chamado **sistema operacional**. O sistema operacional é o primeiro programa a ser executado após o computador ser ligado. Ele literalmente assume o controle do computador para permitir que os recursos computacionais possam ser usados corretamente e compartilhados por diversos outros programas, tais como programas aplicativos e utilitários, ambientes de desenvolvimento integrado, compiladores, interpre-tadores e jogos.

O uso do sistema operacional para controlar o computador apresenta várias vantagens. A primeira vantagem refere-se ao correto funcionamento do computador. Sistemas operacionais são desenvolvidos por equipes especializadas, garantindo que o computador funcione sem erros e de forma segura. Possíveis erros (também chamados de bugs do sistema), assim que identificados são corrigidos pelo fabricante do sistema operacional e a correção é

disponibilizada para os usuários por meio de pacotes de atualização (i.e., comumente via Web, tal como feito no Windows Update). A segunda vantagem do uso de um sistema operacional consiste na rapidez do gerenciamento dos recursos, desde que os componentes de um sistema operacional são customizados para este fim.

Uma terceira vantagem do uso de um sistema operacional é que o computador torna-se mais fácil de usar pelos usuários finais e também pelos programadores. Por exemplo: acessar e interagir diretamente com um disco magnético é uma atividade tediosa, sujeita a erros e com muitos detalhes dependentes do dispositivo físico (i.e., troca-se o modelo de disco magnético, alteram-se estes detalhes). Ao invés disso, o sistema operacional oferece serviços para ler e escrever em um disco magnético. O uso desses serviços pelos programadores é muito mais fácil. Os detalhes da comunicação com o disco magnético são resolvidos de forma transparente pelo sistema operacional. Outro exemplo é o uso de um computador por um usuário leigo, ou seja, um usuário que possui pouca familiaridade com computadores. Para um usuário leigo, é muito mais fácil interagir com o computador por meio de uma interface gráfica constituída por janelas, botões e ícones, do que interagir por meio de uma linha de comandos (i.e., prompt para digitação de comandos em um interpretador de comandos ou shell). Nenhuma interface gráfica é fornecida pelo computador. Cada sistema operacional disponibiliza a sua própria interface gráfica (ou diversas opções de interface gráfica) para interação com os usuários.

Existem diversos sistemas operacionais no mercado. Na plataforma de computadores PC (Personal Computer), os sistemas operacionais mais usados pelos usuários são os sistemas da família Windows da Microsoft (Windows 98, Windows 98SE, Windows XP, Windows NT e Windows Vista), da família Linux (distribuições SUSE da Novell, red hat e Fedora da Red Hat, Inc., Ubuntu da Canonical Ltd., CentOS da The Community ENTERprise Operating System e Debian da Public Interest, Inc.) e da família Mac OS da Apple (atualmente na versão Mac OS X 10.4.1). Há também sistemas operacionais específicos para outras plataformas de compu-

tadores, por exemplo z/OS para os servidores System z da IBM. Nesta disciplina usaremos computadores da plataforma PC com um sistema operacional da família Windows, preferencialmente o sistema operacional Windows XP.

### 1.2.3 Linguagens de Programação

Um computador processa instruções escritas apenas em linguagem de máquina. Uma instrução escrita em linguagem de máquina é composta por uma seqüência de zeros e uns (i.e., seqüência de bits) que tem significado para o computador. Por exemplo, a instrução formada por 00001100 11001100 00110011 pode significar em um computador hipotético SOME 10 e 20 (obs.: note no exemplo o agrupamento em conjuntos de 8 bits, que formam um byte).

É fácil perceber que desenvolver uma solução para um problema complexo usando linguagem de máquina não é simples tampouco intuitivo para nós seres humanos. Ao invés de usar diretamente a linguagem de máquina, programadores usam linguagens de mais alto nível desenvolvidas por fabricantes de soft ware ou disponibilizadas gratuitamente por organizações de padronização ou de open-source. Estas linguagens de mais alto nível possuem construtores que tornam mais direta a concepção do raciocínio lógico da solução. Exemplos de linguagens de mais alto nível são: C (linguagem que será usada nesta disciplina), C++ (extensão da linguagem C que provê suporte ao paradigma de orientação a objetos e não será tratada nesta disciplina), Java, COBOL, Pascal, Delphi, Perl, C# e PHP.

Neste momento, você provavelmente estará se perguntando sobre a seguinte dúvida. Se o computador não entende outra linguagem a não ser a linguagem de máquina, como o computador irá processar instruções escritas em uma linguagem de mais alto nível ? Como o computador irá entender instruções escritas na linguagem C? A resposta é: o computador não irá entender e não irá processar diretamente instruções escritas em uma linguagem de mais alto nível, tal como na linguagem C. Para que isto seja possí-

vel, é usado um **processo de tradução** da linguagem de mais alto nível para a linguagem de máquina. Assim, instruções na linguagem C são traduzidas em um conjunto de instruções equivalentes em linguagem de máquina, antes de ser processadas pelo computador.

O processo de tradução pode ocorrer de duas formas básicas. A primeira forma chama-se **interpretação**. Na interpretação, cada instrução do código-fonte é interpretada e traduzida por um programa chamado **interpretador** em um conjunto de instruções equivalentes em linguagem de máquina. Logo em seguida, o interpretador executa este conjunto de instruções e produz algum resultado. O uso da interpretação apresenta como principal vantagem o fato de que a escrita de programas torna-se mais interativa. Por exemplo, não é necessário escrever todo o programa para obter resultados parciais. Além disso, no início da aprendizagem de uma linguagem de programação, você provavelmente cometerá muitos erros na escrita das instruções. Na interpretação, os erros são identificados à medida que as instruções são escritas, novamente sem a necessidade de se ter o programa completo. Em teoria, isto ajuda no aprendizado em estágios iniciais. Na prática, porém, o uso de interpretadores apresenta sérias desvantagens, tais como visibilidade do código-fonte (i.e., não protege-se que outras pessoas vejam o que está escrito no programa), perda de desempenho (i.e., o código gerado e executado é muito mais lento quando comparado ao processo de compilação. Por exemplo, se uma mesma instrução é executada 500 vezes, o interpretador tem que interpretá-la e depois executá-la 500 vezes) e necessidade de se ter o programa interpretador em cada computador que for executar o programa. Um bom interpretador para a linguagem C, chamado “Embedded Ch”, está disponível para download gratuito em:

<http://www.softintegration.com/>.

Outra forma de se realizar o processo de tradução chama-se **compilação**. Na compilação, o código-fonte deve estar totalmente escrito e sem erros. Um programa chamado **compilador** traduz todas as instruções do código-fonte, gerando o código-

objeto. Este código é chamado de programa-objeto, possui extensão .OBJ e apesar de já estar em linguagem de máquina ainda não pode ser executado pelo computador. Em seguida, um outro programa, chamado **link-editor**, adiciona bibliotecas ao código-objeto que permitem que este código possa ser carregado em memória primária e depois executado. O código resultante é chamado de programa executável e possui extensão .EXE na plataforma Windows. De forma contrária ao processo de interpretação, o processo de compilação esconde o código-fonte, produz um programa mais rápido e não necessita de nenhum programa adicional para executar o programa gerado no processo de compilação. Apesar do processo de compilação envolver alguns passos e programas diferentes (e.g. compilador e link-editor), em geral estes passos e programas estão contidos em um único ambiente, chamado ambiente de desenvolvimento integrado (IDE – Integrated Development Environment). Um IDE possui, além de um compilador e um link-editor, um editor de texto customizado para a escrita do código-fonte (por exemplo, o editor pode usar uma cor específica para representar as palavras reservadas da linguagem), ferramentas de depuração de erros, ferramentas de ajuda, dentre outras. Nesta disciplina, nós utilizaremos o IDE Bloodshed Dev-C++ 5. Na próxima seção, você aprenderá mais sobre o Dev-C++.

Por fim, entende-se por **código-fonte** um programa escrito em uma linguagem de mais alto nível usando um editor de texto. Este programa (chamado **programa-fonte**) é comumente armazenado em um arquivo texto em um disco magnético ou em outras mídias de armazenamento persistente (tais como floppy-disk, CD, DVD e pen-drive). O código-fonte de um programa C possui por convenção a extensão .C (como exemplo, “programa1.c”).

### 1.3 Ambiente de Desenvolvimento Integrado Dev-C++

Na disciplina IntProg será utilizado o ambiente de desenvolvimento integrado Dev-C++ versão 5. Esta seção descreve como **instalar** o Dev-C++ em um computador. Além disso, esta seção en-



sina como usar o Dev-C++ para **digitar** um programa-fonte na linguagem C e depois **compilar** e **executar** o programa.

Relembrando que a **compilação** de um programa tem por objetivo **traduzir** um programa-fonte em um programa executável. Um programa-fonte é mais fácil de ser codificado e compreendido por nós humanos, desde que este programa é escrito em uma linguagem de programação de mais alto nível (por exemplo, escrito na linguagem C), a qual utiliza termos já conhecidos em nossa linguagem (em geral os termos são da língua inglesa, como exemplo os termos begin, end, for e while). Porém, um programa-fonte não pode ser diretamente executado por um computador. De forma contrária, um programa executável possui código binário que pode ser carregado e processado diretamente por um computador. Porém, o uso de código binário na escrita de programas dificulta a elaboração e a organização da lógica que resolverá um certo problema. Desta forma, programadores escrevem código em um programa-fonte que depois será traduzido por um **compilador** em um programa executável.

Um ambiente de desenvolvimento integrado, tal como Dev-C++ 5, possui um compilador e outras ferramentas úteis para a escrita de um programa (e.g., editor de texto adaptado para a linguagem de programação, ferramenta de depuração de programas que auxilia na identificação e remoção de erros de programação, um conjunto de bibliotecas com código que pode ser reaproveitado e a documentação da linguagem).

A **compilação** de um programa somente acontece se o programa-fonte estiver sem erros. Os erros podem ser léxicos, sintáticos ou semânticos. Erros léxicos estão relacionados com o uso indevido de caracteres ou termos que não são permitidos pela linguagem de programação. Como exemplo, o uso do caractere @ em um programa C. Erros sintáticos acontecem quando usamos uma seqüência de termos que não é aceita pela linguagem de programação. Por exemplo, enquanto na linguagem C é permitida a seqüência `< int nome1, nome2 ; >`, não é permitida a seqüência `< int nome1; nome2 ; >`. A troca da primeira vírgula após nome1 por ponto-e-vírgula torna a seqüência incorreta e produz um erro sintático. Erros semânticos, por sua vez, ocorrem quando a asso-

ciação de termos não tem significado ou viola uma restrição da linguagem de programação. Por exemplo, suponha que um termo é declarado para receber apenas valores inteiros (e.g., valor 7). Se o programa tiver um comando de atribuição deste termo com um valor real (e.g, **termo** = 5,8 , sendo que “=” significa comando de atribuição), então ocorrerá um erro semântico. Erros semânticos são mais difíceis de se identificar e corrigir.

### Como obter o Dev-C++ 5 ?

O ambiente de desenvolvimento integrado Dev-C++ versão 5 está disponível para download na URL:

<http://www.bloodshed.net/devcpp.html> da empresa Blood shed Software. Use um navegador (tal como Mozilla Firefox, Internet Explorer ou Netscape) para acessar esta URL.

A página inicial do Dev-C++ possui na parte central superior o título **Dev-C++ 5 (currently beta)**. Desça nesta página inicial até encontrar **Downloads** e clique com o mouse no link imediatamente à frente chamado **Go to Download Page**. Na próxima página, The Dev-C++ Resource Site, localize novamente **Downloads** (em letras bem grandes) e logo abaixo localize o título **Dev-C++ 5.0 beta 9.2 (4.9.9.2) (9.0 MB) with Mingw/GCC 3.4.2**. Para fazer o download do Dev-C++ clique com o mouse no link **SourceForge**. Neste momento, uma janela irá se abrir. Nesta janela, clique no botão **Save File**. Agora, uma nova janela, chamada Enter name of file to save to, irá se abrir. Não mude o nome do arquivo sugerido (i.e., devcpp-4.9.9.2\_setup.exe). Escolha um diretório para salvar este arquivo e anote-o. Clique no botão **Salvar**. Pronto! O download do Dev-C++ foi feito com sucesso.

Uma forma alternativa e mais simples para se fazer o download do ambiente Dev-C++ é acessar diretamente a URL [http://prdownloads.sourceforge.net/dev-cpp/devcpp-4.9.9.2\\_setup.exe](http://prdownloads.sourceforge.net/dev-cpp/devcpp-4.9.9.2_setup.exe). O professor da disciplina também irá disponibilizar o Dev-C++ na sua área da UFSCar na URL [http://www.dc.ufscar.br/~ricardo/download/UAB/devcpp-4.9.9.2\\_setup.exe](http://www.dc.ufscar.br/~ricardo/download/UAB/devcpp-4.9.9.2_setup.exe). Para ambas as URLs, uma janela será aberta e os passos descritos no parágrafo anterior devem ser seguidos para concluir o download do Dev-C++. Por fim, o Dev-C++ também estará disponível para download no site da dis-

ciplina IntProg no ambiente virtual de aprendizagem Moodle (ambiente do seu pólo). Por exemplo, para o pólo de Itapevi, acesse a URL <http://ead.uab.ufscar.br/course/view.php?id=52>. No site da disciplina, localize o título **Programação** e clique no link **Download do Dev-C++ 5**.

### Como instalar o Dev-C++ 5 ?

Primeiramente, usando uma janela de diretórios do Windows (por exemplo, Meu computador), localize o diretório no qual o arquivo **devcpp-4.9.9.2\_setup.exe** foi armazenado (**Passo 1**, Figura 2). Em seguida, clique com o botão esquerdo do mouse duas vezes sobre o arquivo **devcpp-4.9.9.2\_setup.exe**. Uma janela de boas-vindas, com a mensagem “Welcome to Dev-C++ install program.”, irá se abrir. Clique no botão **OK** (**Passo 2**, Figura 3).

O **passo 3** consiste na escolha do idioma que será usado na instalação do Dev-C++. Escolha o idioma Português e clique no botão **OK** (Figura 4). Após, leia o contrato de licença do Dev-C++ na janela do **Contrato de Licença** (**Passo 4**, Figura 5). Se concordar com os termos do contrato, clique no botão **Aceito**.

O **passo 5** consiste na escolha dos componentes do Dev-C++ que serão instalados (Figura 6). Na opção “Escolha o tipo de instalação”, selecione **Full**. Para prosseguir com a instalação, clique no botão **Seguinte >**. Na próxima janela (**Passo 6**), o local de instalação do Dev-C++ será escolhido (Figura 7). Neste momento, será indicado em qual diretório os arquivos do Dev-C++ serão armazenados em seu computador (i.e., Pasta de Destino). O diretório padrão indicado na instalação é **C:\Dev-Cpp**. Sugere-se manter este diretório. Para finalizar a instalação, clique no botão **Instalar**. Se o sistema operacional Windows XP estiver configurado com vários usuários, uma janela aparecerá perguntando se a instalação deve ser aplicada para todos os usuários (i.e., “Do you want to install Dev-C++ for all users on this computer ?”). Clique no botão **Sim** para continuar. Em seguida, a janela “Concluindo o Assistente de Instalação do Dev-C++ 5 beta 9” deverá aparecer indicando uma instalação com sucesso (**Passo 7**, Figura 8). Clique no botão **Terminar**. O programa Dev-C++ será executado pela primeira vez.

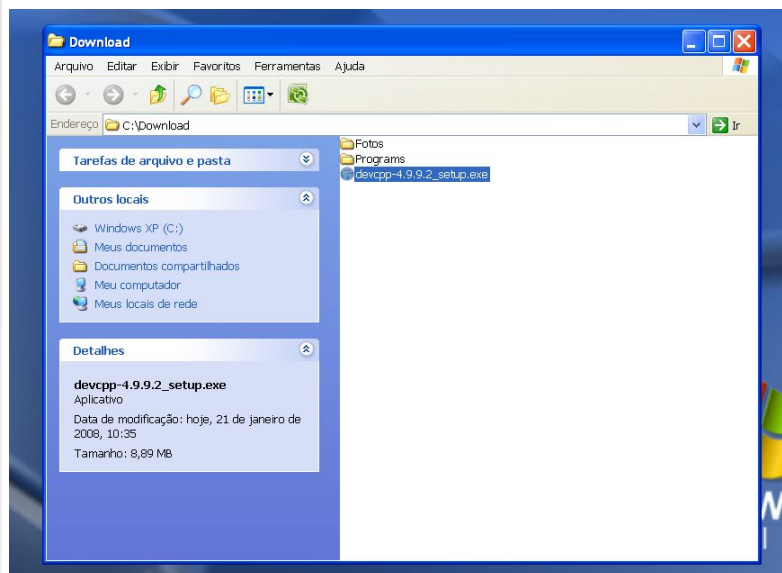


Figura 2. Primeiro passo da instalação do Dev-C++ 5.

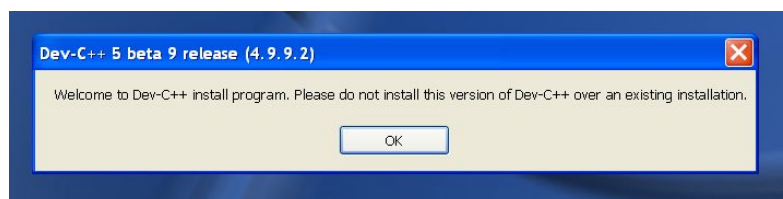


Figura 3. Segundo passo da instalação do Dev-C++ 5.



Figura 4. Terceiro passo da instalação do Dev-C++ 5.

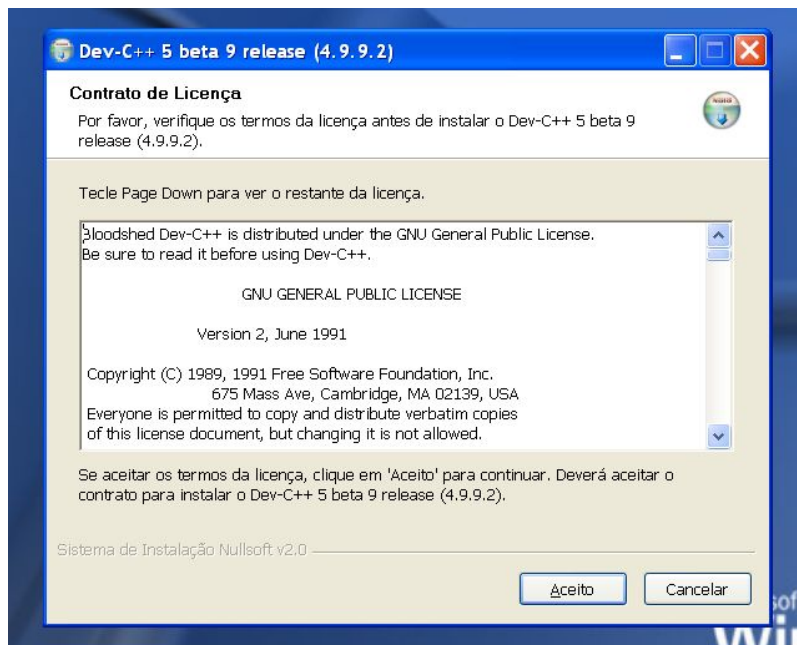


Figura 5. Quarto passo da instalação do Dev-C++ 5.

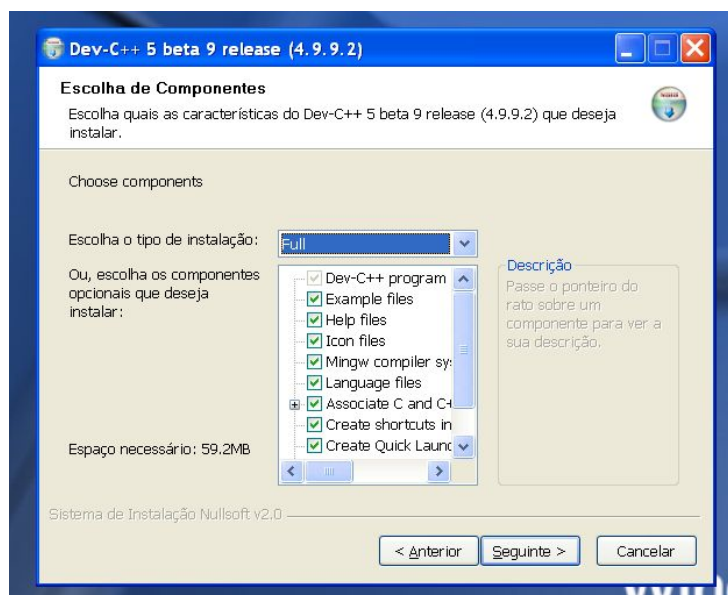


Figura 6. Quinto passo da instalação do Dev-C++ 5.

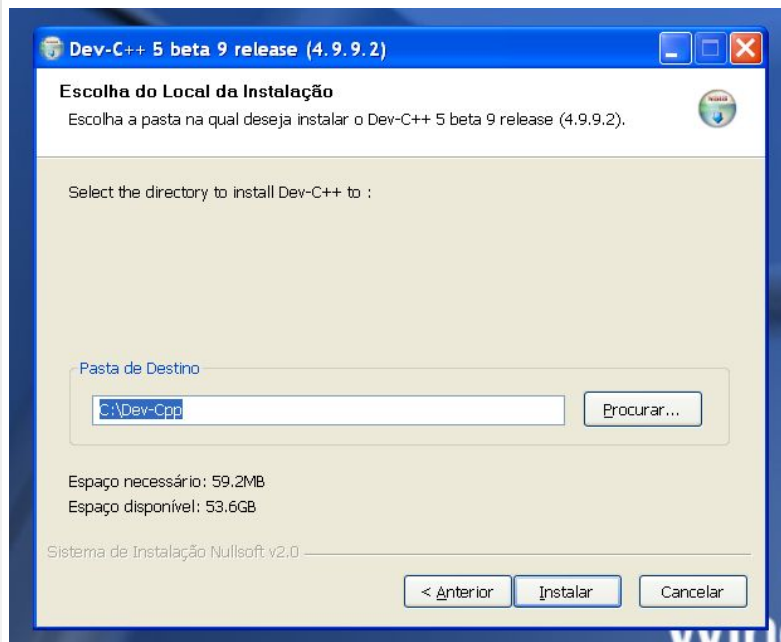


Figura 7. Sexto passo da instalação do Dev-C++ 5.

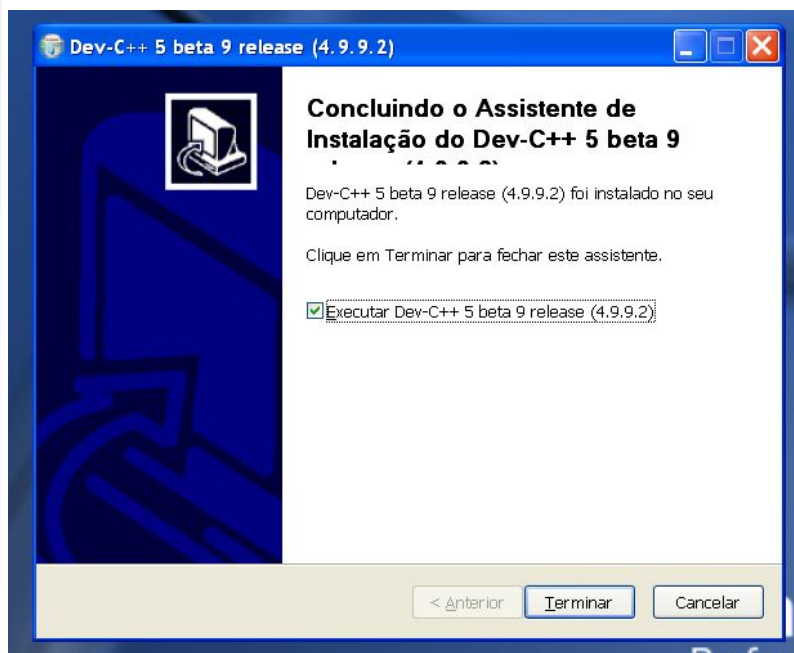


Figura 8. Concluir a instalação.

### Como usar o Dev-C++ 5 ?

Em primeiro lugar, deve-se executar o programa Dev-C++. Uma das formas de se fazer isto consiste em localizar o programa Dev-C++ no **Menu Iniciar** do Windows. A Figura 9 ilustra este passo. Note, porém, que a localização do Dev-C++ no Menu Iniciar do seu computador pode ligeiramente diferir da Figura 9. Neste caso, tente localizar no Menu Iniciar o item **Bloodshed Dev-C++**.

Após a instalação, o Dev-C++ será automaticamente executado. Na primeira execução do programa você terá que configurar algumas opções do ambiente. Neste caso, a janela **Beta version Notice** aparecerá com informações sobre a versão atual do Dev-C++, com instruções de como fazer update do Dev-C++ para novas versões (i.e., como verificar se existe uma nova versão e como mudar para uma nova versão do Dev-C++) e com informações sobre a localização dos arquivos de configuração. Clique no botão **OK** para continuar. Em seguida, aparecerá a janela **Dev-C++ first time configuration**. Nesta janela, selecione a linguagem **Portuguese (Brazil)** em **Select your language**. Clique no botão **Next** para continuar. A próxima janela irá perguntar se o Dev-C++ deve recuperar informações automaticamente dos arquivos. Recomenda-se escolher a opção **Yes, I want to use this feature**. Não escolha esta opção apenas se o computador não tiver muita memória primária (RAM) e secundária (Disco), por exemplo, 512 MB de RAM e 60 GB de disco. Clique no botão **Next** para continuar. A próxima janela irá perguntar se a ferramenta de auto-completar código deve ser usada. Para isso, é necessário criar uma cache. Escolha a opção **Yes, create the cache now**. Clique no botão **Next** para continuar. Isto pode demorar alguns minutos dependendo do poder de processamento do computador. No final, uma nova janela com a mensagem “Dev-C++ has been configured successfully”, indicando que a configuração das opções do ambiente foi feita com sucesso. Clique no botão **OK** para continuar.

Toda vez que o programa Dev-C++ é executado, a janela **Dica do Dia** aparece (Figura 10). Esta janela mostra dicas de uso do ambiente de programação. Leia com atenção as dicas até se familiarizar com o ambiente de programação. Para desabilitar a janela **Dica do Dia** na inicialização do Dev-C++, clique em “Não exibir dicas na inicialização”. Clique nos botões **Próxima** e **Anteri-**

or caso queira ver, respectivamente, novas dicas e dicas já vistas.  
 Clique no botão **Fechar** para iniciar o uso do ambiente de programação.

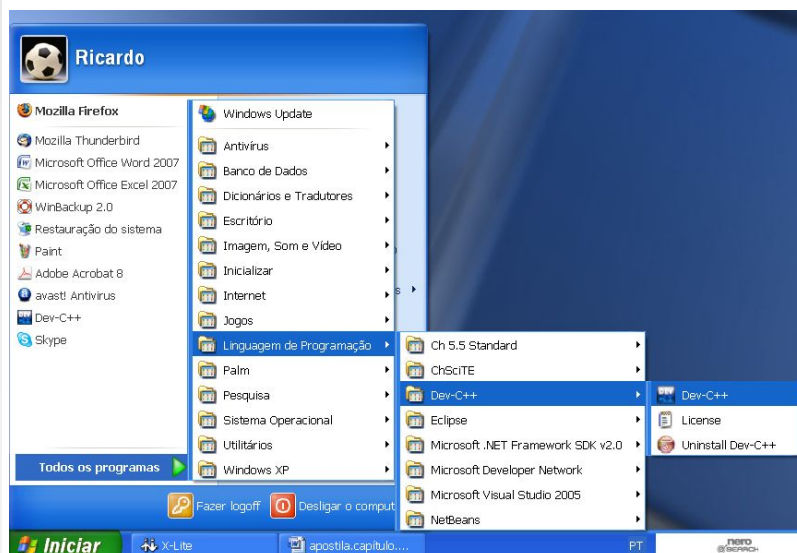


Figura 9. Localizando o programa Dev-C++ usando o Menu Iniciar do Windows XP.

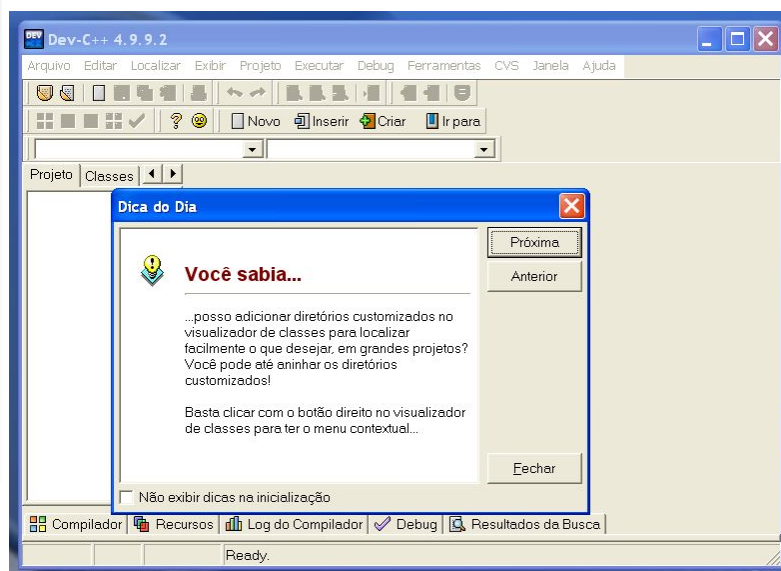
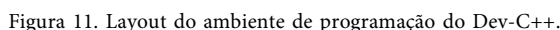


Figura 10. Janela “Dica do Dia”.



[illegible]

Para digitar um programa-fonte na linguagem C, deve-se criar um **novo projeto**. Um projeto permite que vários **arquivos fonte** façam parte de um mesmo grupo e, por conseguinte, sejam compilados conjuntamente para produzir um único programa executável. Para criar um novo projeto, selecione a opção **Novo** do menu **Arquivo** e logo em seguida selecione a opção **Projeto** (Figu-

ra 12). Na janela **Novo projeto**, selecione **Console Application**, escolha um nome para o seu projeto em **Nome** e depois clique no botão **Ok** (Figura 13). Em seguida, a janela **Create new project** permitirá a escolha do diretório no qual os arquivos do novo projeto serão armazenados. Escolha um diretório e clique no botão **Salvar** (Figura 14). O programa Dev-C++ irá abrir um modelo de programa-fonte, ou seja, irá apresentar um programa que possui as partes mínimas para ser compilado (Figura 15). Este modelo de programa-fonte, em particular, ainda não faz nada! Ele apenas possui as partes comumente necessárias para qualquer programa que você irá desenvolver (i.e., reduz-se com isto o esforço de programação). A partir de agora, você pode começar a digitar novas linhas de código referentes ao seu programa (Figura 16).

Note que o editor do Dev-C++ diferencia o texto do programa-fonte por cores, de acordo com o significado das palavras. Por exemplo, **int**, **char** e **return** são mostrados em negrito na cor preta, já que são **palavras reservadas** da linguagem C. Já cadeias de caracteres em um trecho entre aspas duplas (tal como, “Minha primeira linha de código !!! \n”) são mostradas na cor vermelha. O uso de cores facilita e torna mais agradável a programação.

Em especial, deve-se salvar periodicamente o programa-fonte sendo digitado para evitar a perda de dados em uma eventual queda de energia. Para salvar o programa-fonte, pode-se usar a tecla de atalho **Control+S** ou pode-se acessar a opção **Salvar** do menu **Arquivo**.

Por fim, para **compilar** e **executar** um programa, deve-se teclar **F9** ou clicar no terceiro botão da segunda fileira de botões abaixo do menu (Figura 17). Se o compilador verificar que o programa está correto (i.e., sem erros léxicos, sintáticos e semânticos), ele será executado e uma janela aparecerá com o resultado da sua execução (Figura 18).

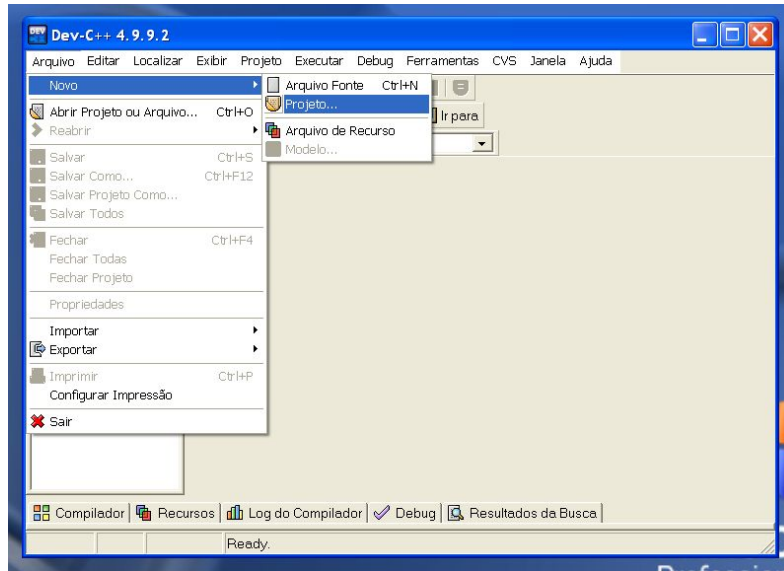


Figura 12. Criação de um novo Projeto.

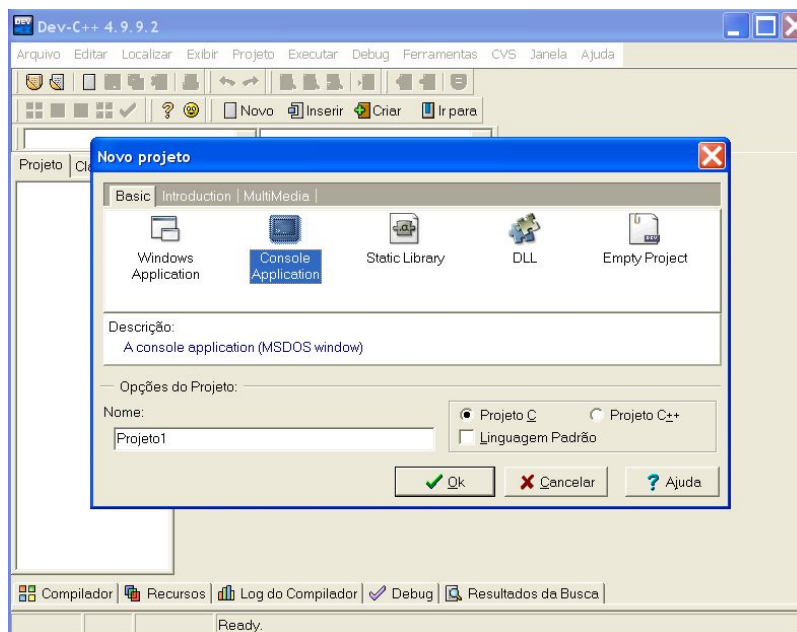


Figura 13. Novo projeto.

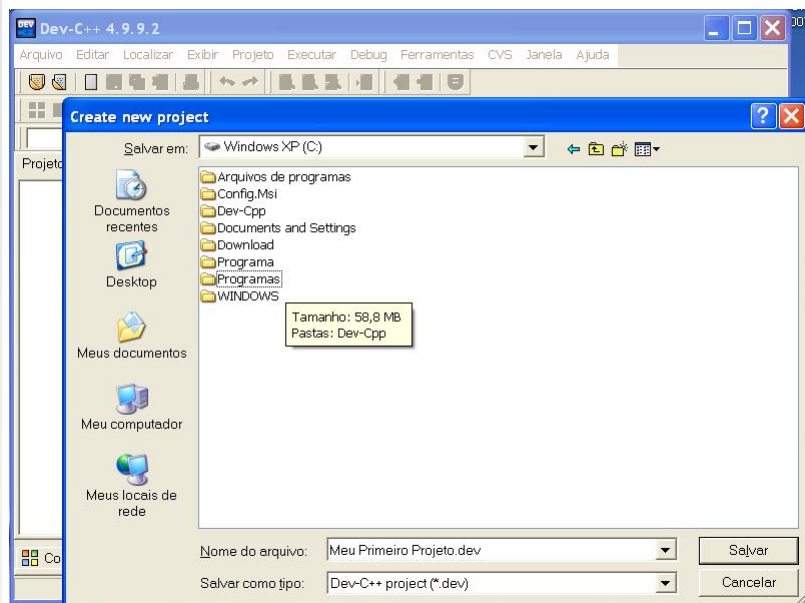


Figura 14. Janela "Create new Project".

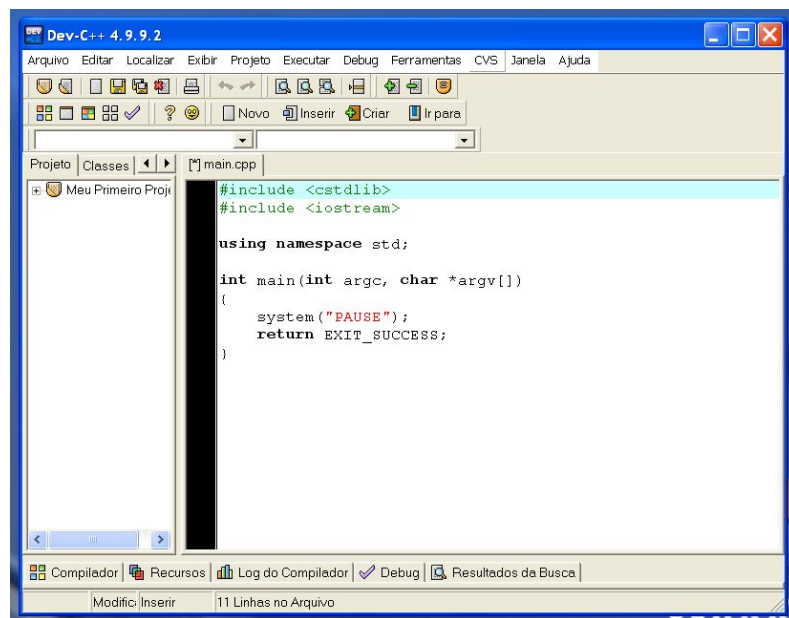


Figura 15. Modelo de programa-fonte.

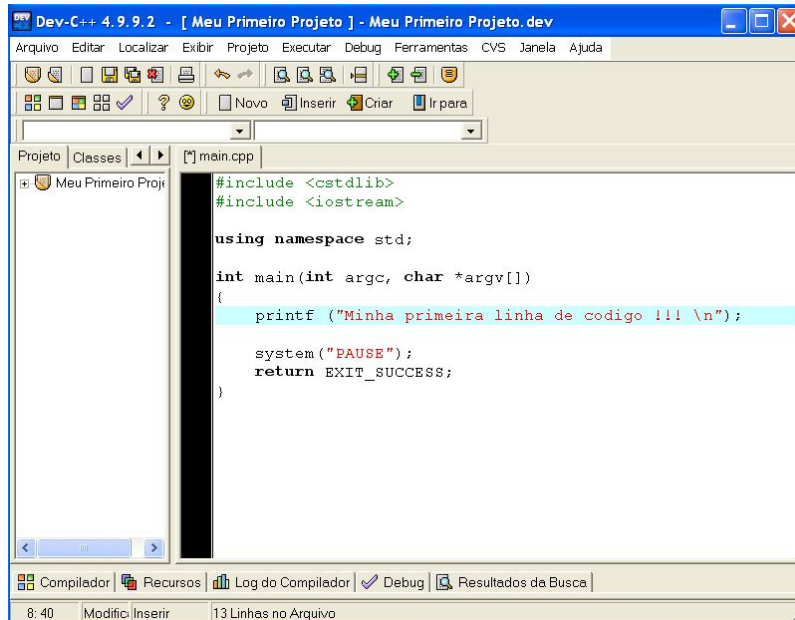


Figura 16. Digitação de novas linhas de código.

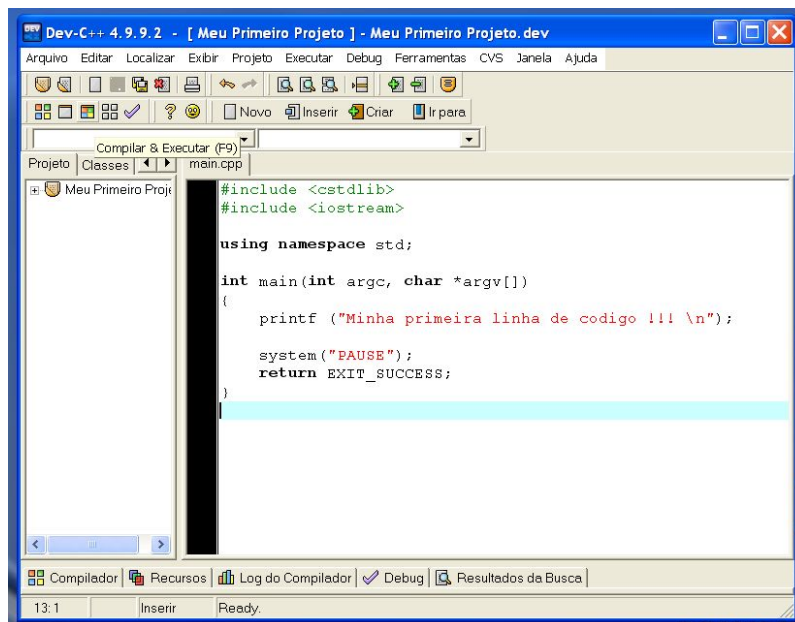


Figura 17. Botão “Compilar & Executar (F9)”.

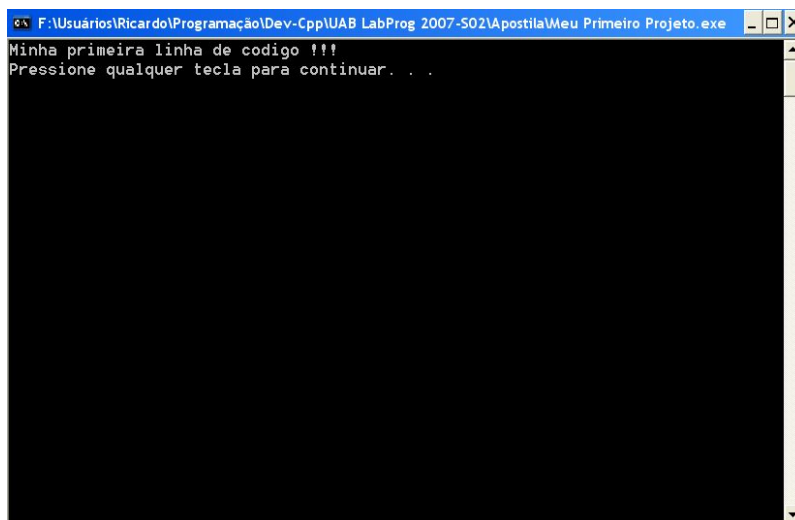


Figura 18. Resultado da execução de um programa.

## 1.4 Lista de Compiladores C

Esta seção apresenta uma lista de compiladores e ambientes de desenvolvimento para a linguagem C (Tabela 3). Alguns destes compiladores e ambientes são gratuitos, ao passo que outros são pagos. A lista não pretende ser exaustiva, ou seja, ela apresenta apenas os principais compiladores e ambientes para a plataforma Windows XP. Vale lembrar que nesta disciplina IntProg será adotado o ambiente de desenvolvimento integrado Dev-C++ 5. Porém, sugere-se que o aluno explore outros ambientes no decorrer da disciplina para estender os seus conhecimentos.

compilador / interpretador / ambiente de desenvolvimento	descrição
Ch <a href="http://www.softintegration.com/">http://www.softintegration.com/</a> ChSciTE <a href="http://chscite.sourceforge.net/">http://chscite.sourceforge.net/</a>	Ch é um interpretador C que pode ser usado diretamente em uma janela de comandos do Windows (i.e., programa cmd). A sua grande vantagem é que o aluno pode ir digitando comandos da linguagem C e já ver os resultados, sem ter que escrever um programa-fonte completo que compile. Já ChSciTE é um ambiente de desenvolvimento integrado para editar e executar programas C usando o interpretador Ch.
compilador / interpretador / ambiente de desenvolvimento	descrição
Dev-C++ 5 <a href="http://www.bloodshed.net/devcpp.html">http://www.bloodshed.net/devcpp.html</a>	Bloodshed Dev-C++ é um completo ambiente de desenvolvimento integrado para programação na linguagem C. O Dev-C++ usa a porta Mingw do compilador GCC (the GNU Compiler Collection) como seu compilador. O compilador GCC é muito usado no desenvolvimento de programas em C, especialmente nas plataformas Unix, Linux e Windows. O compilador GCC é um projeto de software livre que objetiva melhorar a qualidade dos compiladores usados no projeto GNU. Além da linguagem C, o GCC possui front-end para as linguagens Fortran, Java e ADA. O Projeto GNU (pronuncia-se "guh-noo."), iniciado em 1984, visa desenvolver um sistema operacional compatível com o Unix, mas que seja um software livre.

<p>Eclipse IDE for C/C++ Developers <a href="http://www.eclipse.org/">http://www.eclipse.org/</a></p>	<p>Eclipse IDE for C/C++ Developers executa sob a plataforma Eclipse, na atualidade amplamente usada por desenvolvedores. Este ambiente de desenvolvimento integrado propicia funcionalidades avançadas para desenvolvedores em C. Dentre estas funcionalidades, pode-se citar: um editor que destaca as palavras e usa cores em função da funcionalidade da palavra e também completa automaticamente código, um debugger (i.e., programa que permite analisar erros em um programa C) e um gerador de makefile (i.e., permite compilar de forma rápida e fácil diversos programas-fonte para gerar um único programa executável).</p>
<p>NetBeans IDE 6.0 <a href="http://www.netbeans.org/">http://www.netbeans.org/</a></p>	<p>NetBeans IDE 6.0 é um ambiente de desenvolvimento integrado de código aberto. Este ambiente provê suporte para a linguagem C e outras linguagens, tais como Java e Ruby. Atualmente, é um ambiente amplamente usado por desenvolvedores e possui funcionalidades que permite a criação de programas profissionais para computadores desktop e servidores, web e aplicações móveis. NetBeans IDE 6.0 roda nas plataformas Windows, Linux, Mac OS X e Unix Solaris.</p>
<p>Microsoft Visual Studio 2008 <a href="http://msdn2.microsoft.com/en-us/vs2008/default.aspx">http://msdn2.microsoft.com/en-us/vs2008/default.aspx</a></p>	<p>Microsoft Visual Studio 2008 é um produto pago que oferece funcionalidades avançadas de desenvolvimento C na plataforma Windows. Dentre estas funcionalidades, pode-se citar: debugger avançado, suporte a banco de dados, desenvolvimento rápido de aplicações e colaboração entre desenvolvedores.</p>



## 1.5 Conceitos Básicos de Programação

Nesta seção, você aprenderá sobre os seguintes conceitos aplicados à linguagem C:

programa, estrutura geral de um programa C, arquivos de cabeçalho, macros do pré-processador, função `main( )`, tipos de dados, variável, constante, strings, as funções `printf( )` e `scanf( )`, comando de atribuição, comentários e documentação de programas.

### 1.5.1 Estrutura Geral de um Programa

A estrutura geral de um programa na linguagem C é ilustrada na Figura 19.

```
// as duas barras significam comentário em uma linha
// todo comentário será ignorado pelo compilador
// é como se o texto do comentário não existisse
```

```
// arquivos de cabeçalho
#include <stdio.h>
#include <stdlib.h>
...
#include <nomeBiblioteca.h>
```

```
// declarações globais
int totalVendas;
float faturamento;
...
double distanciaMedia;
```

```
// protótipos de funções
int funcao_1(int);
float função_2(int, float);
...
char função_k(float);
```

```
// função principal
int main(int argc, char *argv[])
{
    // declarações
    // sequência de comandos
}

// funções adicionais

int funcao_1(int p)
{
    // declarações
    // sequência de comandos
}

float função_2(int a, float b)
{
    // declarações
    // sequência de comandos
}

...

char função_k(float w);
{
    // declarações
    // sequência de comandos
}
```

Figura 19. Estrutura geral de um programa.

Um programa-fonte na linguagem C sempre começa com a inclusão de arquivos de cabeçalho. Um arquivo de cabeçalho possui código-fonte necessário para a compilação do programa. Mais detalhadamente, um arquivo de cabeçalho armazena uma biblioteca de funções, ou seja, um conjunto de funções que, ao serem incluídas, podem ser usadas no programa. Funções comumente usadas pelos programadores, tais como funções de E/S (entrada e saída de dados) e funções matemáticas, são disponibilizadas pelos ambientes de desenvolvimento para a linguagem C. O padrão ANSI C define um conjunto básico de bibliotecas de funções, denomina-

do de biblioteca C padrão. Nós iremos explorar algumas destas bibliotecas no decorrer da disciplina. Arquivos de cabeçalho, portanto, permitem o compartilhamento de funções comuns entre os programadores.

Uma linha de código que se inicia com `#include` permite a inclusão de um arquivo de cabeçalho. O nome de um arquivo de cabeçalho sempre possui a extensão `.h` (i.e., header file). As bibliotecas de funções disponibilizadas por um ambiente de desenvolvimento integrado são escritas entre os símbolos de menor e maior (i.e., entre `<` e `>`). A Figura 19 exemplifica a inclusão das bibliotecas `stdio` e `stdlib`. Programadores também podem definir as suas próprias bibliotecas. Neste caso, as bibliotecas são escritas entre aspas (i.e., entre `"` e `"`, como em `"minhaBiblioteca.h"`).

Após a inclusão de arquivos de cabeçalho são realizadas as declarações globais do programa. Declarações de variáveis e constantes feitas nesta parte do programa são visíveis para todas as funções do programa, ou seja, as variáveis e constantes podem ser usadas e compartilhadas pelas funções. Ainda não se preocupe em entender o significado dos termos variáveis, constantes, funções, dentre outros. Todos os termos serão explicados posteriormente nesta apostila ou em leituras nos livros-texto. Porém, fixe bem a questão de visibilidade nas declarações globais. Declarações globais devem ser usadas muito cuidadosamente e evitadas sempre que possível, desde que podem induzir a erros na lógica do programa, por exemplo, devido à perda de controle sobre as alterações em uma variável global. A Figura 19 ilustra as declarações globais das variáveis `totalVendas`, `faturamento` e `distanciaMedia`.

Um programa C é formado por um conjunto de funções. Todo programa possui uma função `main()`, pela qual se inicia a execução do programa. A função `main()` pode chamar outras funções adicionais, as quais também podem chamar outras funções e assim sucessivamente. Para permitir o uso de funções adicionais, além do uso da função `main()`, as funções adicionais devem ter os seus protótipos declarados. O protótipo de uma função consiste na definição do tipo de dado de retorno da função, seguido do nome da função e depois entre parênteses devem ser colocados os tipos de dados dos

parâmetros da função (e.g., `float funcao_2(int, float);`). Note que o protótipo de uma função sempre termina com ponto-e-vírgula (;) que é o separador de comandos na linguagem C. A Figura 19 mostra a declaração dos protótipos das funções `funcao_1`, `funcao_2` e `funcao_k`.

A função principal, ou `main()`, inicia-se com a linha de código `int main(int argc, char *argv[])`. A primeira parte desta linha de código, `int`, corresponde ao tipo de dado (i.e., valor inteiro) que a função `main()` retornará ao sistema operacional. Toda função retorna um valor. Já `int argc` refere-se ao parâmetro que indica o número de argumentos passados para a função `main()`, por exemplo, na chamada do programa executável pela linha de comando. Estes argumentos são armazenados no parâmetro `argv`. A função `main()`, assim como qualquer outra função, é delimitada por chaves (i.e., { e }). Assim, todas as declarações e comandos da função `main()` devem ser colocados entre as chaves.

Após o código da função `main()`, o programa deve conter o código de cada função adicional que foi declarada nos protótipos de funções. Uma função adicional segue basicamente o mesmo formato descrito para a função `main()`. Porém, a primeira linha de código de uma função adicional difere da função `main()`. A Figura 19 mostra o formato do código das funções adicionais `funcao_1`, `funcao_2` e `funcao_k`. Neste ponto, termina a estrutura geral de um programa C.

## 1.5.2 Exemplo de Programa 1

O Programa 1 abaixo exemplifica o uso de alguns dos conceitos discutidos na estrutura geral de um programa e também apresenta novos conceitos e detalhes de programação na linguagem C. Este programa, assim como a maioria dos programas descritos nesta apostila, usa dados sobre funcionários de uma empresa.

O Programa 1 realiza a leitura dos dados de um funcionário (i.e., código, idade, sexo e salário) e depois mostra os dados lidos na saída padrão. Da mesma forma que para algoritmos, para pro-

gramas na linguagem C a regra de execução de cada comando é que: (1) os comandos são executados seqüencialmente e na ordem em que estão apresentados. Um ponto-e-vírgula atua como um separador de comandos; e (2) O próximo comando somente é executado quando o comando anterior já tiver terminado.

### Programa 1

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[])
5  {
6      int codFunc; // código do funcionário
7      int idadeFunc; // idade do funcionário
8      char sexoFunc; // sexo do funcionário, M (masculino) e F (feminino)
9      float salFunc; // salário do funcionário
10
11     // entrada de dados do funcionário (teclado)
12
13     printf("A seguir, entre com todos os dados do funcionario.\n\n");
14     printf("Digite o codigo: ");
15     scanf("%d", &codFunc);
16     printf("Digite a idade: ");
17     scanf("%d", &idadeFunc);
18     printf("Digite o sexo [F ou M]: ");
19     scanf("%c%c", &sexoFunc);
20     printf("Digite o salario (R$): ");
21     scanf("%f", &salFunc);
22     printf("\n");
23
24     // saída de dados para a tela (monitor de vídeo)
25
26     printf("Os dados do funcionario sao:\n\n");
27     printf("Codigo: %d\n", codFunc);
28     printf("Idade: %d\n", idadeFunc);
29     printf("Sexo: %c\n", sexoFunc);
30     printf("Salario (R$): %.2f\n", salFunc);
31     printf("\n");
32
33     // finalização do programa principal
34
35     system("PAUSE");
36     return 0;
37 }
```

A inclusão dos arquivos de cabeçalho `stdio.h` e `stdlib.h` é realizada nas linhas 1 e 2 do Programa 1. A biblioteca `stdio` é muito utilizada em programas C, desde que concentra a maioria das funções de E/S. No Programa 1, a inclusão da biblioteca `stdio` permite o uso das funções `printf( )` e `scanf( )`. Já a biblioteca `stdlib` possui uma miscelânea de funções úteis que realizam, dentre outras coisas, a conversão de cadeias de caracteres (i.e., strings) para números, geração de números, gerenciamento de alocação dinâmica de memória, ordenação e algumas funções aritméticas. No Programa 1 é necessário a inclusão desta biblioteca para usar a função `system( )`. Note que a linha 3 não possui nenhum comando e serve como separador entre partes do programa C. Isto facilita a visualização e a organização do código-fonte. As linhas 4 e 5 já foram explicadas anteriormente nesta apostila.

As declarações de variáveis dentro da função `main( )` são feitas nas linhas 6 a 9 do Programa 1. Cada uma destas linhas representa a declaração de uma variável. Uma variável permite que o programa armazene e recupere dados na memória primária. Portanto, uma variável corresponde a um dado armazenado na memória. De fato, um programa recebe dados de entrada, realiza processamentos nestes dados, produz novos dados e por fim gera os resultados na forma de dados de saída. Para tanto, um programa necessita usar variáveis para, por exemplo, armazenar e recuperar os dados de entrada, os dados resultantes de processamentos e os dados de saída. Note que o termo variável indica que o valor do dado armazenado na memória primária pode mudar ao longo da execução do programa, ou seja, o valor não é fixo e sequer estático. Note também que a declaração de uma variável não associa nenhum valor a ela, apenas reserva um espaço em memória primária para receber corretamente os dados.

O nome de uma variável, tais como `idadeFunc` e `sexoFunc` no Programa 1, é chamado identificador. O primeiro caractere de um identificador deve ser uma letra ou um sublinhado (i.e., `_`). Os demais caracteres do identificador podem ser letras, números ou sublinhados. Os identificadores devem ser representativos quanto ao dado que irão armazenar. Deste modo, um identificador que armazenará o sexo de um funcionário de uma empresa (i.e., F para

feminino e M para masculino) pode ser `sexoFuncionario`, `sexoFunc` ou simplesmente `sexo`. Usar os identificadores `SF`, `k` ou `alpha` não é apropriado no contexto deste exemplo, pois perde-se a semântica da variável e por conseguinte dificulta-se o entendimento do código-fonte. Note que a linguagem C diferencia letras minúsculas de letras maiúsculas. Assim, `sexoFuncionario`, `SexoFuncionario`, `SEXOFuncionario` e `sexoFUNCIONARIO` não são o mesmo identificador. Outra recomendação: evite usar identificadores com acento, tal como `sexoFuncionário`. Em geral, evite usar acentos no programa-fonte.

Toda variável é associada a um **tipo de dados**. Esta associação permite que o dado armazenado na variável seja interpretado corretamente. Um tipo de dados especifica: (1) o domínio dos dados, ou seja, o conjunto de valores que pode ser armazenado; (2) a forma de representação física dos dados na memória primária; (3) a quantidade de memória alocada para representar os dados (em bytes); e (4) as operações que podem ser aplicadas aos dados (por exemplo, a operação soma para números inteiros). A linguagem C possui 5 tipos de dados básicos: **char**, **int**, **float**, **double** e **void**.

O tipo **char** é utilizado para armazenar um único caractere, tal como a letra A, o número 8 ou um espaço em branco. O tipo **int** é usado para armazenar números inteiros, tais como -3, 0 e 12. Os tipos **float** e **double** são usados para armazenar números reais (i.e., números com uma parte inteira e outra parte fracionária), tais como -10.31, -1.873, 0.2345 e 34.72829. Estes tipos diferem apenas quanto à precisão na representação dos números reais, sendo o tipo **float** de precisão simples e o tipo **double** de precisão dupla. O tipo **void** significa ausência de valor (i.e., vazio). Este tipo é mais usado com funções para indicar que a função não retorna valor. Não há sentido em usar o tipo **void** com variáveis, exceto com ponteiros. Ponteiros são explicados posteriormente neste texto.

A Tabela 4 resume as características dos tipos de dados básicos da linguagem C. Estas características foram coletadas no ambiente Dev-C++ 5 em um computador PC com processador AMD 64 bits rodando o sistema operacional Windows XP. Infelizmente, estas características podem variar de uma configuração para outra.

Isto consiste em uma grande desvantagem da linguagem C e prejudica a portabilidade de programas.

tipo de dados	domínio dos dados	quantidade de memória (em bytes)	exemplos de operações
char	símbolos da tabela ASCII ou número inteiro equivalente entre -128 e +127	1	detectar se caractere é dígito (0, ..., 9) ou letra (A, ..., Z)
int	-2.147.483.648 a 2.147.483.647	4	soma de números inteiros
float	números reais com 6 dígitos de precisão	4	subtração de números reais
double	números reais com 10 dígitos de precisão	8	divisão de números reais
void	sem valor	não se aplica	não se aplica

Tabela 4. Características dos tipos de dados básicos (do menos para o mais abrangente).

Portanto, as linhas 6 a 9 do Programa 1 declaram duas variáveis do tipo int (i.e., `codFunc` e `idadeFunc`), uma variável do tipo char (i.e., `sexoFunc`) e uma variável do tipo float (i.e., `salFunc`). Recomenda-se que a declaração de variáveis seja feita logo no início do corpo da função, antes da sequência de comandos. No entanto, a linguagem C não impõe qualquer restrição ao local de declaração e esta pode ser em qualquer parte entre as chaves que delimitam a função (i.e., entre `{` e `}`). De qualquer forma, uma variável deve ser declarada antes do seu primeiro uso, independentemente do local de sua declaração.

Note que nas linhas 6 a 9 ocorrem os caracteres `//`. Estes caracteres indicam o início de **comentários** no programa-fonte, os quais se estendem até o final da linha. Assim, todo texto desde a primeira / até o final da linha é ignorado no processo de compilação. Comentários devem sempre ser usados para melhorar a



legibilidade do código-fonte. Porém, não exagere repetindo o significado dos comandos. Use comentários para adicionar uma explicação importante e desconhecida, não para repetir algo conhecido da linguagem. Os comentários usados na declaração das variáveis das linhas 6 a 9 ajudam a entender o significado das variáveis. Outra forma de se colocar comentários na linguagem C é usando o par `/* e */`. Desta forma, todo texto entre este par, e inclusive o par `/* e */`, é considerado comentário. Este texto pode englobar várias linhas do programa-fonte, ao contrário do uso de `//`. Por fim, variáveis de mesmo tipo de dados podem ser declaradas em sequência usando vírgula para separá-las (por exemplo, `int codFunc, idadeFunc` ; ). Novamente, vale destacar o uso do ponto-e-vírgula como separador de comandos na linguagem C. Porém, em algumas partes do programa o uso de ponto-e-vírgula é desnecessário (por exemplo, em `#include` ou no cabeçalho de uma função, tal como em `int main(...)`). Esta diferenciação será fixada com a prática de programação. Portanto, programe!

As linhas 10 e 12 são linhas em branco, sem comandos nem comentários, e servem para separar visualmente as partes do programa. Estas linhas são desprezadas no processo de compilação. A linha 11 é um comentário, o qual define o significado das linhas 13 a 22. Estas linhas, assim como as linhas 26 a 31, realizam E/S de dados. Tipicamente, a saída de dados pelo monitor de vídeo (ou tela) é realizada pela função `printf( )` que pertence à biblioteca C padrão stdio. Por outro lado, a entrada de dados pelo teclado é realizada comumente pela função `scanf( )`, também da biblioteca stdio. Portanto, as linhas 13 a 22 e 26 a 31 realizam conjuntamente a leitura de dados do teclado (i.e., dados de entrada) e a saída de dados para o monitor (i.e., dados de saída).

A função `printf( )` possui pelo menos 1 parâmetro (i.e., `printf(parametro1)` ). Este parâmetro corresponde a uma cadeia de caracteres entre aspas (por exemplo, "este é um exemplo"), chamada **string de controle**, que será enviada para a saída padrão (i.e., monitor de vídeo, também representado pela constante `stdout`). A função `printf( )` também pode imprimir valores de variáveis. Para isto, usa-se especificadores de formato no primeiro parâmetro. Um

especificador de formato começa com o símbolo % e depois é seguido por um caractere que indica o tipo de dado que será impresso na saída padrão. A string de controle "O valor de %d = " imprimirá a letra O, espaço em branco, a palavra valor, espaço em branco, a palavra de, espaço em branco, um valor inteiro (i.e., %d indica um valor do tipo de dados int), espaço em branco, caractere = e espaço em branco. Quando usamos um **especificador de formato** no primeiro parâmetro não informamos de onde será lido o dado que será impresso, apenas informamos o tipo de dados que será impresso na saída padrão. Um parâmetro adicional deve informar de onde será lido o dado, por exemplo, o dado pode ser lido de uma variável. No exemplo anterior, isto corresponde ao comando **printf("O valor de %d = ", codFunc)**. Assim, o valor inteiro que será impresso corresponde ao valor da variável codFunc.

A função printf( ) possui k parâmetros adicionais, onde k corresponde ao número de especificadores de formato que ocorrem no primeiro parâmetro. Os especificadores de formato mais comuns são: %c para caractere (e.g., tipo de dados char), %d para números inteiros (e.g., tipo de dados int), %f para números reais exibidos com parte inteira, ponto e parte fracionária (e.g., tipos de dados float e double), %e para números reais exibidos em notação científica (e.g., tipos de dados float e double, tal como 1.034e+017) e %s para cadeias de caracteres chamadas strings (e.g., "cadeia de caracteres").

O primeiro parâmetro da função scanf( ) também corresponde a uma cadeia de caracteres entre aspas, chamada string de controle. Nesta string ocorrem basicamente 2 tipos de seqüência: especificadores de formato (e.g., %c, %d, %f) e caracteres diferentes de espaço em branco. Um especificador de formato indica o tipo do dado que será lido da entrada padrão (i.e., teclado, também representado pela constante stdin). Por exemplo, %f indica a leitura de um número real com parte inteira, ponto e parte fracionária. Vários especificadores de formato podem aparecer na mesma string de controle, indicando a leitura seqüencial de vários dados. Por exemplo, a string de controle "%d%f%c" indica a leitura de 3 dados, sendo o primeiro um número inteiro, o segundo um número

real com parte inteira e fracionária, e o terceiro um caractere. Um caractere diferente de espaço em branco é usado na string de controle para descartar dados de entrada. Por exemplo, a string de controle "%d+%d" descarta o caractere + da entrada de dados, ou seja, se for digitado no teclado 10+20, apenas os valores 10 e 20 serão armazenados.

Outra forma de descartar dados de entrada é usar \* após o % do especificador de formato. Assim, a string de controle "%\*c%c" indica a leitura de um caractere que será descartado (i.e., %\*c) seguida da leitura de outro caractere que será armazenado (i.e., %c). O local do armazenamento dos dados de entrada é fornecido nos parâmetros subseqüentes da função scanf( ). A função scanf( ) possui k parâmetros adicionais, onde k corresponde ao número de especificadores de formato que ocorrem no primeiro parâmetro. Um parâmetro adicional geralmente corresponde ao endereço de uma variável, o qual é representado pelo símbolo & seguido do nome da variável (e.g., &codFunc é o endereço da variável codFunc). O relacionamento entre os conceitos de **variável**, dado armazenado na variável (i.e., valor) e o endereço da variável é ilustrado na Figura 20. Note que o endereço corresponde ao endereço do primeiro byte da variável. Assim scanf("%d", &codFunc) indica a leitura de um número inteiro que será armazenado na variável codFunc.

10	<b>valor</b>	variável <b>sexoFunc</b> do tipo char (1 byte), <b>endereço 10</b>
11	<b>valor</b>	variável <b>idadeFunc</b> do tipo int (4 bytes), <b>endereço 11</b>
12		
13		
14		
...		

Figura 20. Relacionamento entre os conceitos de variável, valor e endereço.

No Programa 1, as linhas 13 a 22 realizam as seguintes ações:

- A linha 13 imprime a mensagem "A seguir, entre com todos os dados do funcionario." e depois pula 2 linhas na saída padrão. Para indicar uma nova linha é usado o caractere especial de barra invertida seguido do caractere n (i.e., \n).
- A linha 14 imprime a mensagem "Digite o codigo: ", sem pular linha e com espaço após os dois-pontos.
- A linha 15 lê um número inteiro e armazena-o na variável codFunc. Vale destacar que na digitação, o usuário irá digitar um número inteiro seguido da tecla <enter>. Esta tecla fará a mudança automática de linha no monitor, sendo desnecessário um \n.
- A linha 16 imprime a mensagem "Digite a idade: ".
- A linha 17 lê um número inteiro e armazena-o na variável idadeFunc.
- A linha 18 imprime a mensagem "Digite o sexo [F ou M]: ".
- A linha 19 lê um primeiro caractere que será descartado (i.e., <enter> do comando anterior) e em seguida lê outro caractere (i.e., F ou M) que será armazenado na variável sexoFunc.
- A linha 20 imprime a mensagem "Digite o salario (R\$): ".
- A linha 21 lê um número real com parte inteira e fracionária e armazena-o na variável salFunc.
- A linha 22 imprime \n, ou seja, salta de linha.
- As linhas 23 a 25 são linhas em branco e comentário.
- A linha 26 imprime a mensagem "Os dados do funcionario sao:" e depois pula 2 linhas na saída padrão.
- A linha 27 imprime a mensagem "Codigo: " seguido do valor da variável codFunc.
- As linhas 28 a 30 são similares à linha 27, ou seja, essas linhas imprimem uma mensagem seguida do valor de uma variável. O uso de .2 na especificação de formato %.2f na linha 30 indica que o número real será exibido com 2 casas decimais.
- A linha 31 imprime \n, ou seja, salta de linha.
- Novamente, as linhas 32 a 34 são linhas em branco e comentário.

- A linha 35 faz uma chamada para a função `system( )` com o parâmetro `PAUSE` que permite que o usuário veja os resultados antes de terminar o programa. Caso seja omitido, o programa é executado e automaticamente a janela de resultados é fechada, sem dar tempo do usuário ver o resultado do processamento.
- Por fim, a linha 36 possui o comando `return` que indica o término da função `main( )`, retornando o valor 0 para o sistema operacional. Por convenção, o retorno de um valor negativo indica erro, ao passo que o retorno de um valor 0 ou positivo indica a execução normal do programa.

Antes de terminar esta seção, vale enfatizar que:

- o corpo da função `main( )`, no qual são permitidas declarações e comandos, é delimitado por chaves (i.e., entre `{` e `}`), sem ponto-e-vírgula após o fecha chave. Como exemplo, veja a linha 37 do Programa 1;
- deve-se usar ponto-e-vírgula para separar comandos. Por exemplo, linhas 13 a 22 do Programa 1;
- deve-se evitar usar acentos nas declarações de variáveis e também nas mensagens de texto exibidos pela função `printf( )`. O uso de acentos nas mensagens de texto implica no aparecimento de caracteres estranhos na tela. Há formas alternativas de se mostrar acentos no monitor, mas isto não é o objetivo atual do nosso aprendizado;
- a linguagem C diferencia o uso de letras minúsculas e maiúsculas;
- o programa-fonte deve ser documentado. Por exemplo, o programa deve ter comentários que expliquem o significado da execução de um bloco de comandos ou ainda explicação adicional para as declarações de variáveis que permitam compreender melhor o significado delas;
- o programa-fonte deve ser escrito usando indentação, ou seja, os comandos devem ser alinhados verticalmente e com espaçamento de forma a facilitar a leitura do código. Evite escrever diversos comandos em uma mesma linha, a não ser que estes comandos sejam muito simples. Use as mesmas recomendações apresentadas para a apresentação de algoritmos; e

- a numeração das linhas nos programas é feita apenas para que cada comando possa ser referenciado no texto e, assim, não faz parte dos programas.

### 1.5.3 Exemplo de Programa 2

Esta apostila apresentará diversos exemplos de programas na linguagem C para explicar os conceitos da linguagem. Porém, a partir de agora, os programas serão explicados de forma mais sucinta. Linhas de código similares às que já foram explicadas em programas anteriores não serão explicadas novamente.

O Programa 2 inicializa os dados de um funcionário, por meio do uso de constante, comando de atribuição e leitura de dados do teclado. Este programa depois mostra todos os dados na saída padrão.

#### Programa 2

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  /* declarações globais */
5  #define TAMANHO 30 // tamanho máx. do nome de um funcionário
6
7  int main(int argc, char *argv[])
8  {
9      const int codFunc = 1;           // código do funcionário
10     char nomeFunc[TAMANHO];         // nome do funcionário
11     int idadeFunc;                   // idade do funcionário
12     char sexoFunc;                   // sexo do funcionário, M e F
13     float salFunc;                   // salário do funcionário
14
15     // inicialização e entrada de dados do funcionário (teclado)
16
17     idadeFunc = 23;
18     sexoFunc = 'M';
19     salFunc = 500.00;
20
21     printf("A seguir, entre com o nome do funcionário.\n\n");
22     printf("Digite o nome: ");
23     scanf("%s", &nomeFunc);
```

```
24     printf("\n");
25
26     // saída de dados para a tela (monitor de vídeo)
27
28     printf("Os dados do funcionario sao:\n\n");
29     printf("Codigo: %d\n", codFunc);
30     printf("Nome: %s\n", nomeFunc);
31     printf("Idade: %d\n", idadeFunc);
32     printf("Sexo: %c\n", sexoFunc);
33     printf("Salario (R$): %.2f\n", salFunc);
34     printf("\n");
35
36     // finalização do programa principal
37
38     system("PAUSE");
39     return 0;
40 }
```

A linha 4 exemplifica a forma alternativa de se fazer comentários na linguagem C, ou seja, usar texto entre `/*` e `*/`.

A linha 5 define uma **macro** do pré-processador da linguagem C, chamada TAMANHO e cujo **valor de substituição** é 30. No processo de compilação, o pré-processador substitui no programa-fonte cada ocorrência de TAMANHO pelo seu valor 30. Neste caso, a macro foi usada no sentido de uma **constante**, ou seja, de um valor fixo, que não se altera durante a execução do programa. Isto é particularmente útil quando um valor se repete em várias partes do programa. Assim, qualquer mudança deste valor é realizada apenas na macro e automaticamente a mudança se reflete em todo programa. Caso contrário, a mudança deve ser realizada em cada parte do programa que usa o valor, o que pode ser tedioso e sujeito a erros (i.e., esquecimento de um local, que conduz a um erro de lógica). Use sempre macros para valores constantes.

Outra forma de se declarar uma constante é usar a palavra reservada **const** antes do tipo em uma declaração.

Na linha 9 é declarada uma constante chamada codFunc do tipo int, a qual recebe o valor fixo 1. O uso de `#define` não aloca nenhuma memória, nem associa o valor a um tipo, desde que o valor de substituição da macro é colocado diretamente no progra-

ma-fonte antes de sua tradução para código executável. Já o uso de `const` aloca uma área de memória para a constante durante a execução do programa, mas esta área não pode ter seu valor alterado, além de associar o valor fixo a um tipo. O uso de `const` é mais genérico e somente com seu uso pode-se escrever código usando constantes em alguns casos, por exemplo como parâmetro da função `printf( )`.

A linha 10 declara uma **string de caracteres** chamada `nomeFunc`. Esta string pode armazenar até 30 caracteres, de acordo com o valor de substituição da macro `TAMANHO`, e termina sempre com o caractere especial `\0` (i.e., caractere indicador de fim de string). Uma string de caracteres pode ser visualizada como um **arranjo de caracteres**, que começa na posição 0 e vai até a posição `tamanho-1`. A Figura 21 ilustra um possível conteúdo para a string `nomeFunc`. Strings de caracteres serão novamente abordadas nesta apostila, mas devido ao seu grande uso este conceito já foi introduzido neste segundo programa.

0	1	2	3	4	5	6	7	...	29
R	I	C	A	R	D	O	\0	...	

Figura 21. Conteúdo da string de caracteres `nomeFunc`.

As variáveis também podem receber um valor por meio de um comando de atribuição. As linhas 17 a 19 ilustram a utilização do comando de atribuição. Um **comando de atribuição** (i.e., símbolo `=`) possui a forma **variável = valor**. Veremos posteriormente que este valor pode ser o resultado de uma expressão, que pode envolver outras variáveis e constantes. Portanto, a linha 17 atribui o valor 23 a variável `idadeFunc`, a linha 18 atribui o valor M a variável `sexoFunc` e a linha 19 atribui o valor 500.00 a variável `salFunc`.

A leitura de uma string de caracteres é exemplificada na linha 23. Note que a função `scanf( )` usa o especificador de formato `%s` para atribuir um valor a string `nomeFunc`. De forma similar, a



função `printf( )` da linha 30 usa o especificador de formato `%s` para imprimir o conteúdo de `nomeFunc`.

### 1.5.4 Explicações Adicionais

Esta seção reforça a explicação de alguns dos conceitos estudados. Para tanto, serão usados exemplos de trechos de programas.

#### Arquivos de Cabeçalho

Um arquivo de cabeçalho é incluído no programa usando `#include`. Todo programa criado pelo ambiente de desenvolvimento integrado Dev-C++ 5 possui a inclusão das bibliotecas `stdio` e `stdlib`. Outras bibliotecas devem ser adicionadas ao programa-fonte caso queira-se usar funções específicas. A Figura 22 mostra a inclusão de alguns arquivos de cabeçalho.

```
#include <stdio.h> // biblioteca adicionada automaticamente pelo Dev-C++
#include <stdlib.h> // biblioteca adicionada automaticamente pelo Dev-C++
#include <math.h> // funções que implementam operações matemáticas
#include <time.h> // funções de manipulação de datas e horas
#include <string.h> // funções adicionais para manipulação de strings
#include <ctype.h> // funções para classificar e transformar caracteres
...
```

Figura 22. Arquivos de cabeçalho.

#### Macros do pré-processador

Uma macro é definida usando `#define`. Este é um recurso poderoso para permitir a substituição e o controle de um valor fixo (i.e., constante) em várias partes de um programa. Uma macro também pode ser usada para substituir funções **inline**. Por exemplo, a função inline `obterMaximo(a,b)` pode ser uma macro que é substituída pelo código `a>b?a:b`. O significado deste código não é importante no momento, apenas fixe que toda vez que o pré-processador

encontrar obterMaximo(a,b), ele substituirá por  $a > b ? a : b$ . A Figura 23 ilustra o uso de macros do pré-processador.

```
...
#define TAMANHO 30
#define TOTAL 100
#define IDADE 18
#define obterMaximo(a,b) a>b?a:b
...
```

Figura 23. Macros do pré-processador.

### Tipos de Dados, Modificadores e Constantes

Com exceção do tipo void, os demais tipos de dados básicos (i.e., char, int, float e double) podem usar modificadores. Um **modificador** é usado antes do tipo para que este tipo se torne ligeiramente diferente. Por exemplo, um modificador pode ser usado para estender a faixa de valores de números inteiros ou para eliminar valores negativos do domínio de números reais. Os modificadores disponíveis na linguagem C são: **short**, **long**, **signed** e **unsigned**. O modificador short e long respectivamente diminui e aumenta a faixa de valores do domínio de dados. O modificador signed e unsigned respectivamente inclui ou exclui números negativos. Assim, em algumas plataformas (i.e., isto não é uniforme), o uso de **long int** permite representar números inteiros em uma faixa de valores muito maior, tanto para números inteiros negativos quanto para números inteiros positivos, do que permitido com o tipo int. Como exemplo, se um número inteiro em uma plataforma usar 2 bytes e variar entre -32.768 e 32.767, um número long int poderá variar entre -2.147.483.648 e 2.147.483.647. Em geral, o tipo long int possui o dobro do tamanho do tipo int.

Já o uso de **unsigned int** permite representar uma maior faixa de números inteiros positivos. Por exemplo, se um número inteiro variar entre -32.768 e 32.767, um número unsigned int poderá variar entre 0 e 65.535.

O tipo `char` permite o armazenamento de um caractere. Na realidade, o valor armazenado é um número inteiro relativo ao código ASCII do caractere. Como exemplo, o caractere 'A' corresponde ao valor 65. Desta forma, uma variável do tipo `char` pode ser interpretada como um caractere, seu uso mais comum, mas também como um número inteiro. Desde que um `char` ocupa 1 byte, a sua faixa de valores varia entre -128 e 127. Usando modificador, o tipo `unsigned char` permite uma faixa de valores entre 0 e 255, os quais podem ser interpretados como um caractere ou como um número inteiro. A Figura 24 ilustra o uso de modificadores.

Ainda com relação aos tipos de dados da linguagem C, existe também o tipo **ponteiro**. Este tipo será melhor descrito posteriormente na apostila. Uma breve descrição consiste em: um ponteiro permite o armazenamento de um endereço de memória, por exemplo, do endereço de uma variável (i.e., sua posição inicial). De posse deste endereço é possível obter o valor armazenado na variável. Por isso, um ponteiro tem associado a ele um tipo de dados, o qual indica qual o tipo do valor armazenado no endereço apontado pelo ponteiro. A Figura 24 ilustra a declaração de um ponteiro chamado `pont`, o qual aponta para um endereço que armazena dados do tipo `int`. Note o uso de `*` para indicar que `pont` é um ponteiro. Há também a possibilidade de se usar um ponteiro do tipo `void` (e.g. `void *ponteiro`). Nesse caso, o ponteiro é genérico e poderá apontar para qualquer tipo de dado. Porém, o controle da quantidade de dados apontada deve ser feita de forma mais cuidadosa, pois não se sabe na declaração qual o seu tipo.

Enquanto em algoritmos foi apresentado o **tipo lógico** (ou **booleano**), a linguagem C não possui um tipo correspondente. O tipo lógico é representado na linguagem C como um inteiro (tipo `int`), sendo que o valor falso corresponde ao valor 0 e o valor verdadeiro corresponde a um valor diferente de 0. A linguagem C++, que estende a linguagem C, possui o tipo `bool` que é equivalente ao tipo lógico descrito em algoritmos, sendo o valor verdadeiro representado como `true` e o valor falso representado como `false`. No entanto, o seu uso não é permitido em um programa puro C.

Diversas constantes já foram exemplificadas nesta apostila. Uma constante entre aspas, tal como "este exemplo", é chamada de

constante literal. Uma constante de um único caractere sempre aparece entre aspas simples, tal como 'a', e não entre aspas duplas, tal como "a". Constantes literais podem ser armazenadas em strings, ao passo que uma constante de um único caractere pode ser armazenada em variáveis do tipo char. Os números -5, 10, -8.93 e 123.71 são constantes numéricas e podem ser armazenados em dados do tipo int, float e double.

```
...  
unsigned char ch1;  
unsigned int inteiro1;  
long int inteiro2;  
unsigned long int inteiro3;  
long double real;  
...  
int *pont;  
...
```

Figura 24. Tipos de dados e modificadores.

## Strings

Cadeias de caracteres são muito usadas em programas C. A biblioteca string.h oferece funções para a manipulação destas cadeias. Em particular, não é possível atribuir uma constante literal para uma variável string usando diretamente o comando de atribuição. Assim, este comando está incorreto: **nomeFunc = "Ricardo"**; Com strings é necessário usar a função strcpy( ) da biblioteca string. O comando correto é ilustrado na Figura 25.

```
#include <string.h>  
...  
strcpy(nomeFunc, "Ricardo");  
...
```

Figura 25. Biblioteca string e função strcpy( ).

## Comando de Atribuição e Conversão de Tipos (Cast)

Em um comando de atribuição pode ser necessária a conversão de tipos de dados, conhecida como cast. Algumas conver-

sões entre os tipos de dados char, int, float e double são realizadas automaticamente pelo compilador. Considere os seguintes comandos: **salFunc = 10** e **temp = 30.2**. Nota-se que em um comando de atribuição, tipos de dados diferentes podem estar associados. A variável salFunc é do tipo float, enquanto o valor 10 é uma constante inteira. Neste caso, o valor 10 é automaticamente convertido para um número real (i.e., 10.0) e depois atribuído para salFunc. O contrário também pode ocorrer. Se temp é do tipo int, a atribuição de 30.2 irá causar a perda da parte fracionária (i.e., temp ficará com o valor 30). A ordem de conversão de tipos é a seguinte: char, int, float e double. A Figura 26 ilustra exemplos de conversão de tipos em um comando de atribuição.

```
...  
int temp;  
float salFunc;  
  
temp = 30.2;      // temp ficará com o valor 30  
salFunc = 10;     // salFunc receberá 10.0  
...
```

Figura 26. Conversão de tipos em um comando de atribuição.

### 1.5.5 Mapeamento de Algoritmo para Programa C

Na disciplina de "Construção de Algoritmos" você aprendeu a definição formal de algoritmo. Um **algoritmo** é um conjunto finito de instruções (ou comandos) que, se executadas, permitem a manipulação de um conjunto finito de dados de entrada para produzir um conjunto finito de dados de saída, dentro de um tempo finito.

Um **programa** corresponde a uma formulação concreta de um algoritmo. Em outras palavras, enquanto o algoritmo é descrito baseado em um modelo de computador abstrato, um programa é descrito baseado em um computador real usando uma linguagem de programação. Assim, um programa usa representações e estruturas mais específicas, dependentes de um computador real. Como consequência, um programa pode ser executado, após compilado, por um computador. Já um algoritmo é usado para ilustrar a lógica

da solução de um problema, sem se preocupar com detalhes de implementação. Porém, um algoritmo não pode ser executado por um computador real.

Uma boa prática no desenvolvimento de uma solução computacional, especialmente nos estágios iniciais de aprendizado, é primeiro apresentar a solução usando um algoritmo e somente depois adaptar este algoritmo para uma linguagem de programação. Você verá por meio de exemplos que apesar da escrita de um algoritmo diferir da escrita de um programa, não é difícil perceber o relacionamento entre os conceitos presentes em algoritmos e programas C. Ao longo desta apostila são mostrados exemplos de mapeamento de algoritmos para programas. A explicação deste mapeamento somente será feita caso o relacionamento dos conceitos não seja direto. A seguir é apresentado o mapeamento de 2 algoritmos, cujos relacionamentos entre os conceitos são diretos.

**Algoritmo 2-2** (apostila de "Construção de Algoritmos")

```
1 { leitura de nome e idade com escrita de mensagem usando estes dados }
2
3     algoritmo
4         declare
5             nome: literal
6             idade: inteiro
7
8         { leitura de nome e idade do teclado }
9         leia(nome)
10        leia(idade)
11
12        { saída da mensagem na tela }
13        escreva(nome, " tem ", idade, " anos.")
14    fim-algoritmo
```

**Programa 3** (equivalente ao algoritmo 2-2)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  /* leitura de nome e idade
5     com escrita de mensagem usando estes dados */
6
7  int main(int argc, char *argv[])
8  {
9     char nome[30];
10    int idade;
11
12    // leitura de nome e idade do teclado
13
14    printf("Digite o nome: ");
15    scanf("%s", &nome);
16    printf("Digite a idade: ");
17    scanf("%d", &idade);
18    printf("\n");
19
20    // saída da mensagem na tela
21
22    printf("%s tem %d anos\n\n", nome, idade);
23
24    system("PAUSE");
25    return 0;
26 }
```

**Algoritmo 2-4** (apostila de "Construção de Algoritmos")

```
1 { cálculo do consumo médio de um veículo, dados a quilometragem total
2   percorrida e o número de litros utilizados nos dois únicos
3   abastecimentos realizados }
4
5   algoritmo
6       declare
7           quilometragem, consumo,
8           abastecimento1, abastecimento2: real
9
10      { obtenção da quilometragem e litros consumidos }
11      leia(quilometragem)
12      leia(abastecimento1, abastecimento2)
```

```
13
14     { cálculo do consumo }
15     consumo ← • quilometragem/(abastecimento1 + abastecimento2)
16
17     { saída do resultado calculado }
18     escreva(consumo, " km/l")
19     fim-algoritmo
```

**Programa 4** (equivalente ao algoritmo 2-4)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  /* cálculo do consumo médio de um veículo,
5   dados a quilometragem total percorrida e
6   o número de litros utilizados nos dois únicos
7   abastecimentos realizados
8   */
9
10 int main(int argc, char *argv[])
11 {
12     float quilometragem, consumo, abastecimento1, abastecimento2;
13
14     // obtenção da quilometragem e litros consumidos
15
16     printf("Digite a quilometragem: ");
17     scanf("%f", &quilometragem);
18     printf("Digite o abastecimento 1: ");
19     scanf("%f", &abastecimento1);
20     printf("Digite o abastecimento 2: ");
21     scanf("%f", &abastecimento2);
22     printf("\n");
23
24     // cálculo do consumo
25
26     consumo = quilometragem/(abastecimento1 + abastecimento2);
27
28     // saída do resultado calculado
29
30     printf("%f km/l", consumo);
31     printf("\n\n");
32
33     system("PAUSE");
34     return 0;
35 }
```



### 1.5.6 Observações Finais sobre a Linguagem C

A linguagem C possui um conjunto reduzido de palavras reservadas (total de 32 palavras). Palavras reservadas, também conhecidas como palavras-chave, são palavras que tem um significado especial para a linguagem C e que não podem ser usadas para outro propósito. Portanto, uma palavra reservada não pode, por exemplo, ser usada como nome de variável. As 32 palavras reservadas da linguagem ANSI C são: **auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile e while.**

O mapa conceitual da memória primária de um programa C é composto basicamente por 4 regiões: área de código do programa, área de variáveis globais, área de heap e área de pilha. A **área de código** do programa armazena o programa executável (i.e., conjunto de instruções em linguagem de máquina formado de zeros e uns). A **área de variáveis** globais é reservada para as declarações globais em um programa C. A **área de heap** é usada para alocação dinâmica de memória. Este é um recurso avançado da linguagem C que permite a construção de estruturas de dados mais sofisticadas, como listas e árvores. A **área de pilha** é usada para armazenar os valores de parâmetros nas chamadas de funções. As áreas de heap e pilha são complementares, ou seja, é alocada uma única área de memória, porém o seu uso é feito em sentidos opostos. Por exemplo, o heap pode alocar memória no sentido do maior endereço para o menor endereço, enquanto a pilha pode alocar memória no sentido do menor endereço para o maior endereço. Isto é possível porque o uso do heap e da pilha varia muito ao longo da execução de um programa e pode inclusive não estar sendo utilizado em um ponto da execução. Alguns autores também destacam a existência de uma quinta área, chamada área de variáveis locais, na qual é alocada memória para as variáveis declaradas dentro das funções. No entanto, o mais comum é que as variáveis locais sejam alocadas

na área de pilha. Porém, como dito, esta explicação representa o mapa conceitual da memória e pode ligeiramente diferir em diferentes plataformas de computador.

## Unidade 2

---

Bases Numéricas, Expressões e Comandos  
Condicionais

---



## 2.1 Bases Numéricas

Nesta seção, você aprenderá sobre os seguintes **conceitos**:

base numérica, base binária, base octal, base decimal, base hexadecimal, mudança de base, byte, palavra e bit.

As pessoas, em seu cotidiano, representam números usando o sistema decimal. Este sistema utiliza os dígitos de 0 a 9 para representar um número. Note que são usados 10 dígitos diferentes, reflexo de nossa maneira primitiva de contar com os dedos de nossas mãos. Por isto, este sistema é chamado **base decimal** ou **base 10**. Alguns exemplos de números nesta base são: 0, 1, 3, 7, 11, 375, 1023 e 90200. A posição relativa de um dígito em um número na base 10 indica a ordem de grandeza do dígito. Em outras palavras, cada dígito possui um valor relativo com relação ao número. Por exemplo, para o número 11, o seu primeiro dígito 1 à esquerda é 10 vezes o valor do segundo dígito 1. Portanto, um número  $d_n d_{n-1} \dots d_1 d_0$  na base decimal, onde cada  $d_i$  ( $n \geq i \geq 0$ ) corresponde a um dígito do número na posição relativa  $i$ , pode ser decomposto de acordo com a Equação 1. Desta forma, o número 11 pode ser decomposto em  $(1 \times 10^1) + (1 \times 10^0)$ , ou seja, em  $(1 \times 10) + (1 \times 1)$ .

$$numero = (d_n \times 10^n) + (d_{n-1} \times 10^{n-1}) + \dots + (d_1 \times 10^1) + (d_0 \times 10^0) \quad (1)$$

Um mesmo valor pode ser representado por diferentes números em bases numéricas distintas. Por exemplo, 11 na base decimal é equivalente a 1011 na base binária. A notação  $(n)_b$  indica que um número  $n$  está representado na base  $b$ . Assim, 11 na base decimal é representado por  $(11)_{10}$  e 1011 na base binária é representado por  $(1011)_2$ . Em computação, algumas bases são mais usuais. Os computadores representam informações (tais como dados, instruções e endereços de memória) usando seqüências de zeros e uns. Portanto, a base binária é naturalmente usada com computadores. Uma seqüência de zeros e uns é normalmente dividida e agrupada em bytes. Um **byte** corresponde a **8 bits**, sendo que cada

**bit** pode armazenar um valor zero ou um valor um, mas nunca ambos os valores ao mesmo tempo. Por exemplo, a sequência de zeros e uns 1111000011000011 pode ser agrupada em 2 bytes (byte 1 = 11110000 e byte 2 = 11000011). Em alguns tipos de computador, bits podem ser agrupados para formar unidades maiores do que 1 byte (8 bits), chamadas de **palavras**. Por exemplo, a sequência 1111000011000011 pode formar uma palavra de 16 bits em um computador. O agrupamento de bits em bytes e palavras torna o uso das bases octal e hexadecimal muito usuais em computação. Portanto, além de números na base decimal, nós teremos que aprender a usar números nas bases binária, octal e hexadecimal. Como curiosidade, o fato dos computadores usarem números na base binária, faz com que as medidas de quantidade de memória sejam expressas em potência de 2. Assim, 1 kilobyte (KB) corresponde a 1024 bytes (ou  $2^{10}$ ) e não a 1000 bytes.

Uma base numérica  $b$  permite a representação de números utilizando os dígitos 0 a  $b-1$ . Por exemplo, a base binária (ou base 2) representa números usando os dígitos 0 e 1. Já a base octal (ou base 8) representa números usando os dígitos 0 a 7. Quando a base numérica excede o uso de 10 dígitos (i.e., 0 a 9), letras maiúsculas são utilizadas para representar os dígitos subsequentes (tal como a letra A). Por exemplo, a base hexadecimal (ou base 16) representa números usando os dígitos 0 a 9 e as letras A a F. Neste caso, a letra A indica o valor  $(10)_{10}$ , a letra B indica o valor  $(11)_{10}$ , e assim sucessivamente até a letra F que indica o valor  $(15)_{10}$ .

Para qualquer base numérica  $b$ , tem-se a seguinte propriedade: um certo número  $d_n d_{n-1} \dots d_1 d_0$ , onde cada  $d_i$  ( $n \geq i \geq 0$ ) corresponde a um dígito do número na posição relativa  $i$ , pode ser decomposto usando a Equação 2. Por exemplo, o número  $(1011)_2$  pode ser decomposto em  $(1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$ , ou seja, em  $8 + 0 + 2 + 1$  que corresponde a  $(11)_{10}$ . Similarmente, o número  $(13)_8$  pode ser decomposto em  $(1 \times 8^1) + (3 \times 8^0)$ , ou seja, em  $8 + 3$  que corresponde a  $(11)_{10}$ . Já o número  $(B)_{16}$  pode ser decomposto em  $(B \times 16^0)$ , ou seja, em  $(11 \times 16^0)$  que corresponde a  $(11)_{10}$ . A Tabela 2 mostra os números 0 a 16 e 99 na base 10 e seus números correspondentes nas bases 2, 8 e 16.

$$numero = (d_n \times b^n) + (d_{n-1} \times b^{n-1}) + \dots + (d_1 \times b^1) + (d_0 \times b^0) \quad (2)$$

Em particular, os números binários que terminam em **0** são **pares** e os números que terminam em **1** são **ímpares**. Observe o exemplo para um número de 4 bits (Tabela 1).

$(0101)_2 =$	$0 \times 2^3 +$	$1 \times 2^2$	$0 \times 2^1 +$	$1 \times 2^0$
$=$	$0 +$	$4 +$	$0 +$	$1$
$=$	$4 + 1 = (5)_{10}$			

Tabela 1. Transformação do número binário 0101 para base 10.

### 2.1.1 Mudança de Base

As bases numéricas binária (base 2), octal (base 8) e hexadecimal (base 16) são potências de 2. Desta forma, pode-se usar o **método de substituição direta** para transformar um número  $n_1$  em uma destas bases em um número equivalente  $n_2$  em uma das outras bases. Para fazer a mudança de base de um número, é necessário entender a forma como cada dígito nas bases 8 e 16 é agrupado em bits. Na base octal, cada dígito corresponde a 3 bits. Na base hexadecimal, cada dígito corresponde a 4 bits.

Para transformar um número na base octal (ou hexadecimal) em um número na base binária, substitui-se cada dígito do número pelo seu equivalente em bits. Por exemplo,  $(143)_8$  pode ser representado por 001 100 011, onde 001 é a codificação de 3 bits para o número 1, 100 é a codificação de 3 bits para o número 4 e 011 é a codificação de 3 bits para o número 3. Portanto,  $(143)_8$  corresponde a  $(1100011)_2$ , desprezando-se os bits zeros à esquerda (i.e., bits não significativos). Em outro exemplo,  $(63)_{16}$  pode ser representado por 0110 0011, onde 0110 é a codificação de 4 bits para o número 6 e 0011 é a codificação de 4 bits para o número 3. Portanto,  $(63)_{16}$  corresponde a  $(1100011)_2$ , desprezando-se os bits zeros à esquerda.

base 10	base 2	base 8	base 16
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10
99	1100011	143	63

Tabela 2. Valores em diferentes bases numéricas.

Para transformar um número na base binária em um número na base octal (ou hexadecimal), faz-se o inverso. Primeiramente, o número binário é dividido em grupos de bits (i.e., 3 bits para mudança para a base octal e 4 bits para mudança para a base hexadecimal). Em seguida, converte-se cada grupo de bits em seu dígito equivalente na base destino (i.e., dígito octal ou dígito hexadecimal). Por exemplo,  $(1100011)_2$  é dividido em grupos de 3 bits para ser transformado para a base 8, ou seja, em 001 100 011. Se necessário, dígitos 0 são acrescentados à esquerda do primeiro grupo. Após, converte-se 001 para 1, 100 para 4 e 011 para 3 formando o número  $(143)_8$ . Em outro exemplo,  $(1100011)_2$  é dividido em grupos de 4 bits para ser transformado para a base 16, ou seja, em 0110 0011. Após, converte-se 0110 para 6 e 0011 para 3, formando o número  $(63)_{16}$ .



Para transformar um número na base binária, octal ou hexadecimal em um número na base decimal, usa-se a Equação 2. Exemplos deste tipo de transformação já foram mostrados anteriormente no texto. O inverso, ou seja, transformar um número na base decimal em um número na base binária, octal ou hexadecimal, é realizado usando o **método das divisões**. Neste método, o número na base decimal é dividido pelo valor da base (i.e., 2, 8 ou 16, dependendo da mudança de base), gerando um quociente  $q_1$  e um resto  $r_1$ . Enquanto o quociente gerado não for menor do que a base, repete-se a divisão. Assim, o quociente  $q_1$  é dividido pelo valor da base, gerando um quociente  $q_2$  e um resto  $r_2$ . Por sua vez, o quociente  $q_2$  é dividido pelo valor da base, gerando um quociente  $q_3$  e um resto  $r_3$ . Isto é realizado sucessivamente. No final, o número formado pelo último quociente e os restos na ordem inversa (...  $r_3$   $r_2$   $r_1$ ) correspondem ao número na base destino. Por exemplo, para transformar  $(99)_{10}$  em um número na base binária, faz-se as seguintes divisões:  $99 \div 2$  (quociente 49, resto **1**),  $49 \div 2$  (quociente 24, resto **1**),  $24 \div 2$  (quociente 12, resto **0**),  $12 \div 2$  (quociente 6, resto **0**),  $6 \div 2$  (quociente 3, resto **0**) e  $3 \div 2$  (quociente **1**, resto **1**). Os números em negrito nesta divisão formaram o número na base binária, ou seja,  $(1100011)_2$ . Outro exemplo, para transformar  $(99)_{10}$  em um número na base octal, faz-se as seguintes divisões:  $99 \div 8$  (quociente 12, resto **3**) e  $12 \div 8$  (quociente **1**, resto **4**). Portanto,  $(99)_{10}$  corresponde a  $(143)_8$ . Já para transformar  $(99)_{10}$  em um número na base hexadecimal, faz-se as seguintes divisões:  $99 \div 16$  (quociente **6**, resto **3**). Portanto,  $(99)_{10}$  corresponde a  $(63)_{16}$ . As Figuras Figura 1 e Figura 2 ilustram alguns exemplos de conversão de base.

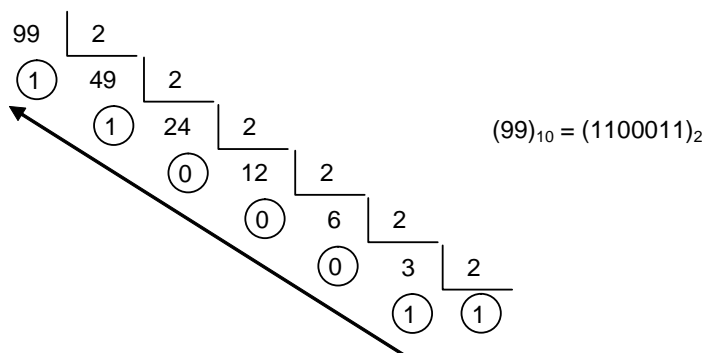


Figura 1. Conversão de  $(99)_{10}$  para a base binária.

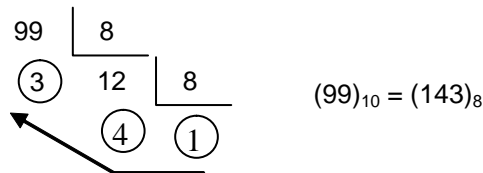


Figura 2. Conversão de  $(99)_{10}$  para a base octal.

## 2.1.2 Representação na Linguagem C

A linguagem C oferece suporte a números nas bases octal, decimal e hexadecimal. Um número na base octal deve ser precedido pelo valor 0. Por exemplo, 0143 representa  $(143)_8$ . Um número na base hexadecimal deve ser precedido pelos caracteres 0x. Por exemplo, 0x63 representa  $(63)_{16}$ . Por outro lado, um número na base decimal não precisa ser precedido por nada. Por exemplo, 99 representa  $(99)_{10}$ . As funções `printf( )` e `scanf( )` possuem especificadores de formato próprios para lidar com números nas bases octal e hexadecimal. O especificador `%o` é usado com números na base octal e os especificadores de formato `%x` (letras minúsculas) e `%X` (letras maiúsculas) são usados com números na base hexadecimal. O Programa 1 ilustra o uso destes conceitos.

### Programa 1

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[])
5  {
6      char octal, decimal, hexadecimal;
7
8      // mesmo valor em diferentes bases numéricas
9
10     octal = 0143;
11     decimal = 99;
12     hexadecimal = 0x63;
13
```

```
14    // impressão dos valores
15
16    printf("O valor em octal eh: %o\n", octal);
17    printf("O valor em decimal eh: %d\n", decimal);
18    printf("O valor em hexadecimal eh: %x\n", hexadecimal);
19    printf("\n");
20
21    system("PAUSE");
22    return 0;
23 }
```

## 2.2 Expressões

Nesta seção, você aprenderá sobre os seguintes **conceitos**:

expressão aritmética, operandos, operadores aritméticos, precedência de operadores, regra da promoção de tipos, divisão de números inteiros, operadores combinados com atribuição, operador vírgula, linguagem de montagem, operadores bit a bit, cast implícito e explícito, expressão relacional, operadores relacionais, comparação de strings, expressão lógica, condição lógica e operadores lógicos.

### 2.2.1 Expressões e Operadores Aritméticos

Uma **expressão aritmética** utiliza um conjunto de operadores aritméticos para manipular números (i.e., operandos da expressão) visando produzir um resultado, ou seja, um novo número. Portanto, o conceito de expressão aritmética está diretamente relacionado ao conceito de expressão matemática. A Equação 3 apresenta a forma geral de uma expressão aritmética.

$$\text{operando\_1} \text{ operador } \text{operando\_2} \text{ operador } \dots \text{ operando\_N} \quad (3)$$

Na linguagem C, os **operandos** de uma expressão aritmética podem ser dos tipos de dados int, float e double (com modificadores, tais como short ou unsigned, caso necessário). O tipo char também é permitido. Um operando do tipo char em uma

expressão aritmética é analisado pelo seu conteúdo (i.e., código ASCII, que é um número), e não pela sua representação em caractere, por exemplo letra 'A'. Tipicamente, um operando pode ser representado por uma constante numérica (e.g., -17, -5.78, 18 ou 99.1255243), por uma constante (e.g., const int total = 17 ou const char caractere = 'A') ou por uma variável (e.g., float altura ou double raio).

Os **operadores aritméticos**, listados na Tabela 3, representam as possíveis operações que podem ser aplicadas aos operandos. Alguns operadores são unários, ou seja, são aplicados a um único operando. Por exemplo, o operador - unário inverte o sinal de um operando e deve ser interpretado como  $-1 \times$  operando (e.g., -17 ou -altura). Já outros operadores aritméticos são binários, ou seja, são aplicados a 2 operandos (como exemplo, altura + 3 ou  $18 * 3$ ). Note que o mesmo símbolo - é usado para representar diferentes operadores aritméticos. Se o símbolo - preceder um único operando, ele é interpretado como unário (e.g., -17). Caso contrário, o símbolo - é interpretado como binário (e.g., altura - 15).

operador	significado	unário / binário	precedência
++	incremento	unário	1
--	decremento	unário	1
-	inversão de sinal	unário	2
*	multiplicação	binário	3
/	divisão	binário	3
%	módulo (ou resto)	binário	3
+	adição	binário	4
-	subtração	binário	4

Tabela 3. Operadores aritméticos.

Na avaliação de uma expressão aritmética, os operadores possuem diferentes níveis de precedência. O **nível de precedência** indica a ordem na qual os operadores serão avaliados em uma expressão. Na Tabela 3, os operadores aritméticos com precedência 1 serão avaliados em primeiro lugar, seguidos dos operadores com precedência 2 e assim sucessivamente. Desta forma, primeiramente

são avaliados os operadores de incremento e decremento, enquanto os operadores de adição e subtração são avaliados por último. Operadores que possuem o mesmo nível de precedência são avaliados de acordo com sua posição na expressão, sendo uma expressão aritmética avaliada nestes casos sempre da esquerda para a direita. O uso de parênteses em uma expressão aritmética altera a ordem de precedência dos operadores. Mais especificamente, os operadores dentro de parênteses são primeiramente avaliados. Por exemplo, na expressão  $3 + 5 * 2$ , primeiramente é avaliado o operador  $*$  (i.e., precedência 3) e depois é avaliado o operador  $+$  (i.e., precedência 4) produzindo o resultado 13. Já na expressão  $(3 + 5) * 2$ , primeiramente é avaliado o operador  $+$  dentro dos parênteses e depois é avaliado o operador  $*$  produzindo o resultado 16. Note que diferentes ordens de execução dos operadores (i.e., com e sem parênteses) podem produzir resultados distintos para uma expressão aritmética. Parênteses podem ser usados de forma aninhada em uma expressão aritmética. Ou seja, um parêntese pode estar contido dentro de outro parêntese mais externo. Os parênteses mais internos são avaliados em primeiro lugar. Como exemplo, na expressão aritmética  $((3 + 5) * (2 + 8))$ , primeiramente são avaliados os parênteses  $(3 + 5)$  e  $(2 + 8)$ , para depois ser avaliado o parêntese externo com os resultados  $(8 * 10)$ .

O **tipo de dado do resultado** produzido por uma expressão aritmética é determinado em função dos tipos de dados de seus operandos. A forma mais simples de uma expressão aritmética é quando a expressão possui todos operandos de um mesmo tipo. Por exemplo, uma expressão pode ter todos operandos do tipo float. Neste caso, o tipo do resultado será o tipo de dado dos operandos. Entretanto, uma expressão aritmética em geral possui operandos de tipos de dados diferentes, porém compatíveis entre si. Como exemplo, a expressão  $x + y * z$  pode possuir operandos de tipos diferentes, ou seja,  $x$  do tipo int,  $y$  do tipo float e  $z$  do tipo double. Neste caso, a **regra da promoção de tipos** é aplicada para transformar (cast), implicitamente, os tipos de dados dos operandos para um único tipo de dado. O tipo de dado do resultado será o tipo mais abrangente dos operandos. Para a expressão  $x + y * z$ ,

o tipo mais abrangente é double (Tabela 4). Assim, primeiramente será avaliado  $y * z$ . O tipo float de  $y$  será transformado para double antes de se efetuar a multiplicação. Iremos chamar o resultado parcial de  $rp$  do tipo double. Após, o tipo int de  $x$  será transformado para double antes de se efetuar a adição (i.e.,  $x + rp$ ). A seqüência a seguir ilustra a abrangência dos tipo de dados, do tipo menos abrangente para o tipo mais abrangente: char, int, unsigned int, long, unsigned long, float, double, long double.

O Programa 2 abaixo exemplifica o uso de expressões aritméticas. Este programa, assim como a maioria dos programas descritos nesta apostila, usa dados sobre funcionários de uma empresa. O Programa 2 realiza a leitura dos dados de 3 funcionários (i.e., dados sobre o código, a idade e o salário de cada funcionário), calcula a idade média e o salário médio dos funcionários, e no final mostra todos os dados lidos e calculados na saída padrão.

tipo de dados	domínio dos dados	quantidade de memória (em bytes)	exemplos de operações
char	símbolos da tabela ASCII ou número inteiro equivalente entre -128 e +127	1	detectar se caractere é dígito (0, ..., 9) ou letra (A, ..., Z)
int	-2.147.483.648 a 2.147.483.647	4	soma de números inteiros
float	números reais com 6 dígitos de precisão	4	subtração de números reais
double	números reais com 10 dígitos de precisão	8	divisão de números reais
void	sem valor	não se aplica	não se aplica

Tabela 4. Características dos tipos de dados básicos (do menos para o mais abrangente).

**Programa 2**

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[])
5  {
6      int codFunc1; // código do funcionário 1
7      int idadeFunc1; // idade do funcionário 1
8      float salFunc1; // salário do funcionário 1
9
10     int codFunc2; // código do funcionário 2
11     int idadeFunc2; // idade do funcionário 2
12     float salFunc2; // salário do funcionário 2
13
14     int codFunc3; // código do funcionário 3
15     int idadeFunc3; // idade do funcionário 3
16     float salFunc3; // salário do funcionário 3
17
18     int totalFunc = 0; // total de funcionários
19     int somaIdade = 0; // soma das idades dos funcionários
20     float somaSalario = 0; // soma dos salários dos funcionários
21     float idadeMedia; // idade média dos funcionários
22     float salarioMedio; // salário médio dos funcionários
23
24     /* entrada de dados do funcionário 1 e
25        cálculos intermediários */
26
27     printf("A seguir, entre com todos os dados do funcionario 1.\n\n");
28     printf("Digite o codigo: ");
29     scanf("%d", &codFunc1);
30     printf("Digite a idade: ");
31     scanf("%d", &idadeFunc1);
32     printf("Digite o salario (R$): ");
33     scanf("%f", &salFunc1);
34     printf("\n");
35
36     totalFunc = totalFunc + 1;
37     somaIdade = somaIdade + idadeFunc1;
38     somaSalario = somaSalario + salFunc1;
39
40     /* entrada de dados do funcionário 2 e
41        cálculos intermediários */
42
```

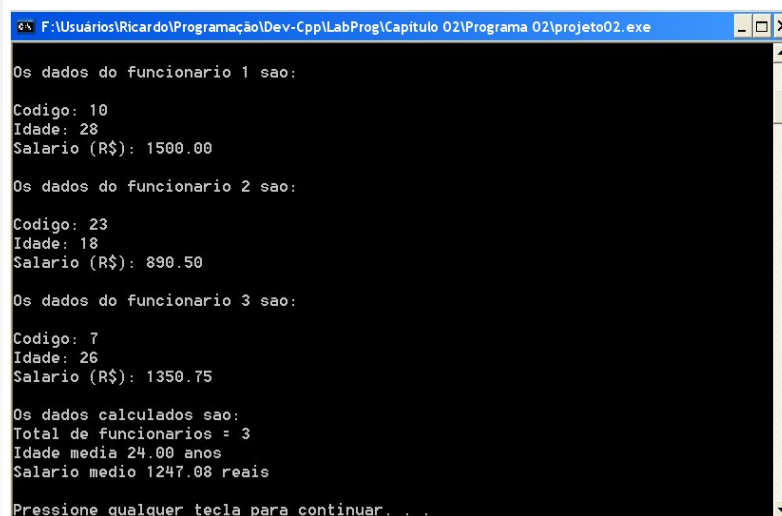
```
43     printf("A seguir, entre com todos os dados do funcionario 2.\n\n");
44     printf("Digite o codigo: ");
45     scanf("%d", &codFunc2);
46     printf("Digite a idade: ");
47     scanf("%d", &idadeFunc2);
48     printf("Digite o salario (R$): ");
49     scanf("%f", &salFunc2);
50     printf("\n");
51
52     totalFunc = totalFunc + 1;
53     somaIdade = somaIdade + idadeFunc2;
54     somaSalario = somaSalario + salFunc2;
55
56     /* entrada de dados do funcionário 3 e
57        cálculos intermediários */
58
59     printf("A seguir, entre com todos os dados do funcionario 3.\n\n");
60     printf("Digite o codigo: ");
61     scanf("%d", &codFunc3);
62     printf("Digite a idade: ");
63     scanf("%d", &idadeFunc3);
64     printf("Digite o salario (R$): ");
65     scanf("%f", &salFunc3);
66     printf("\n");
67
68     totalFunc = totalFunc + 1;
69     somaIdade = somaIdade + idadeFunc3;
70     somaSalario = somaSalario + salFunc3;
71
72     // cálculo da idade média e do salário médio dos funcionários
73
74     idadeMedia = (float) somaIdade / totalFunc;
75     salarioMedio = somaSalario / totalFunc;
76
77     // saída de dados do funcionário 1 para a tela
78
79     printf("Os dados do funcionario 1 sao:\n\n");
80     printf("Codigo: %d\n", codFunc1);
81     printf("Idade: %d\n", idadeFunc1);
82     printf("Salario (R$): %.2f\n", salFunc1);
83     printf("\n");
84
85     // saída de dados do funcionário 2 para a tela
86
```



```
87     printf("Os dados do funcionario 2 sao:\n\n");
88     printf("Codigo: %d\n", codFunc2);
89     printf("Idade: %d\n", idadeFunc2);
90     printf("Salario (R$): %.2f\n", salFunc2);
91     printf("\n");
92
93     // saída de dados do funcionário 3 para a tela
94
95     printf("Os dados do funcionario 3 sao:\n\n");
96     printf("Codigo: %d\n", codFunc3);
97     printf("Idade: %d\n", idadeFunc3);
98     printf("Salario (R$): %.2f\n", salFunc3);
99     printf("\n");
100
101     // saída dos dados calculados
102
103     printf("Os dados calculados sao:\n");
104     printf("Total de funcionarios = %d\n", totalFunc);
105     printf("Idade media %.2f anos\n", idadeMedia);
106     printf("Salario medio %.2f reais\n", salarioMedio);
107     printf("\n");
108
109     // finalização do programa principal
110
111     system("PAUSE");
112     return 0;
113 }
```

No Programa 2, as linhas 18 a 20 possuem **declarações de variáveis com inicialização**. Com isto, uma variável é declarada e já recebe um valor inicial (i.e., para as 3 variáveis o valor 0). As linhas 36 a 38, 52 a 54, 68 a 70, 74 e 75 ilustram a utilização de expressões aritméticas. Em todos os casos, a expressão aritmética está associada a um comando de atribuição (e.g., `totalFunc = totalFunc + 1`). Quando isso ocorre, o tipo de dado da variável do lado esquerdo do comando de atribuição (i.e., variável que receberá o resultado da expressão) deve ter um tipo compatível com o tipo do resultado da expressão. Em alguns casos pode ser necessária a conversão de tipos de dados (revisar o item “Comando de Atribuição e Conversão de Tipos (Cast)” na seção 1.5.4 do capítulo 1 da apostila).

Especial atenção deve ser dispensada para a **divisão de números inteiros**. Lembre-se que a divisão de números inteiros sempre gera um resultado inteiro (i.e., do tipo `int`). Por exemplo,  $5 / 2$  produz como resultado o valor 2 e não o valor 2.5. Por sua vez,  $5.0 / 2$  produz como resultado 2.5 (note que 5.0 é um número de ponto flutuante). Este fato pode afetar a lógica do programa e produzir erros. Por isto, na linha 74 foi necessário fazer uma conversão explícita de tipos de dados (i.e., **cast explícito**). A variável `somaIdade` foi explicitamente convertida do tipo `int` para o tipo `float` (i.e., `(float)somaIdade`). O operador de cast (`float`) é um operador unário e foi aplicado à variável a sua direita. Desta forma, força-se que `somaIdade / totalFunc` produza um resultado do tipo `float`, pela aplicação da regra da promoção de tipos. A Figura 3 ilustra um exemplo de execução do Programa 2.



```
F:\Usuários\Ricardo\Programação\Dev-Cpp\LabProg\Capitulo 02\Programa 02\projeto02.exe

Os dados do funcionario 1 sao:
Codigo: 10
Idade: 28
Salario (R$): 1500.00

Os dados do funcionario 2 sao:
Codigo: 23
Idade: 18
Salario (R$): 890.50

Os dados do funcionario 3 sao:
Codigo: 7
Idade: 26
Salario (R$): 1350.75

Os dados calculados sao:
Total de funcionarios = 3
Idade media 24.00 anos
Salario medio 1247.08 reais

Pressione qualquer tecla para continuar...
```

Figura 3. Execução do Programa 2.

## 2.2.2 Expressões e Operadores Relacionais

Uma **expressão relacional** permite a comparação de valores (i.e., operandos) para estabelecer uma relação de ordem total, ou seja, para estabelecer qual valor precede ou sucede o outro. A Equi-

ção 4 ilustra a forma geral de uma expressão relacional, sendo que `operando_1` e `operando_2` podem ser variáveis, constantes, constantes numéricas (e.g., -1, 10.5 e 500), constantes literais (e.g., 'A', 'Z', "Brasil" e "Itália") ou ainda uma expressão aritmética.

`operando_1 operador operando_2` (4)

Já o operador, chamado de **operador relacional**, pode ser qualquer operador listado na Tabela 5. Uma expressão relacional sempre produz como resultado um valor lógico (i.e., valor verdadeiro ou falso). Porém, a linguagem C não possui constantes para representar verdadeiro e falso. Ao invés disso, a linguagem C interpreta um valor zero como falso (i.e., falso é 0) e qualquer valor diferente de zero como verdadeiro. Por exemplo, enquanto somente 0 é considerado o valor falso, diversos valores podem ser considerados como o valor verdadeiro, tais como -1, 1, 10 e 500. Em geral, números inteiros (i.e., tipo `int`) são usados para representar verdadeiro e falso, por causa de sua representação compacta. Mas números reais (i.e., `float` e `double`) também podem ser usados, mas isto acontece muito raramente. Por convenção, o resultado de uma expressão relacional é 0 para falso e 1 para verdadeiro (apesar de diferentes valores representarem verdadeiro).

operador	significado	unário / binário	precedência
<	menor que	binário	1
<=	menor ou igual a	binário	1
>	maior que	binário	1
>=	maior ou igual a	binário	1
==	igual a	binário	2
!=	diferente de	binário	2

Tabela 5. Operadores relacionais.

Algumas observações com relação ao uso de expressões relacionais. Não confundir o operador relacional de igualdade (i.e., símbolo composto `==`) com o operador de atribuição (i.e., símbolo `=`). Em algoritmos, o operador relacional de desigualdade pode ser representado pelos símbolos `<>` e `≠`. Na linguagem C, de for-

ma diferente, o operador relacional de desigualdade é representado pelo símbolo composto `!=`. Enquanto em algoritmos os operadores relacionais “menor ou igual a” e “maior ou igual a” são representados por símbolos simples, ou seja, respectivamente por  $\leq$  e  $\geq$ , na linguagem C estes operadores são representados por símbolos compostos, ou seja, respectivamente por `<=` e `>=`. Note também que todos os operadores relacionais são binários e, portanto, envolvem 2 operandos.

O Programa 3 a seguir exemplifica o uso de expressões relacionais. Este programa realiza a leitura dos dados de 2 funcionários (i.e., dados sobre o código, o nome, a idade, o sexo e o salário de cada funcionário) e compara os dados dos funcionários para produzir valores lógicos na saída padrão.

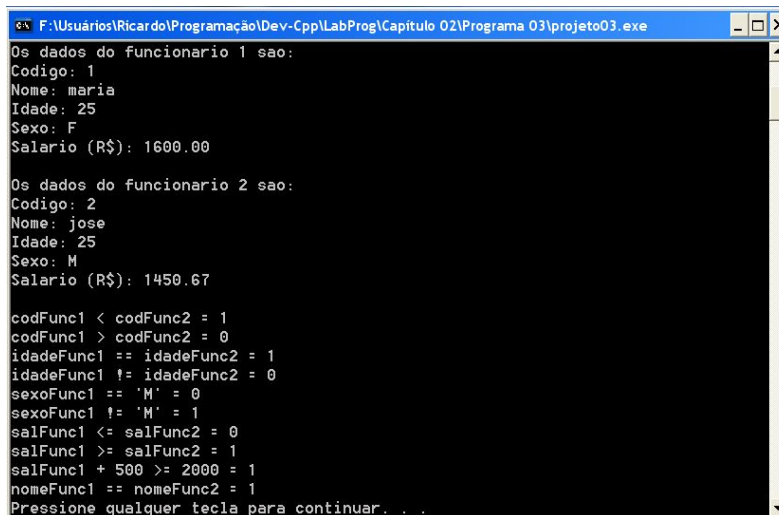
### Programa 3

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  #define TAMANHO 30 // tamanho máx. do nome de um funcionário
6
7  int main(int argc, char *argv[])
8  {
9      int codFunc1;      // código do funcionário 1
10     char nomeFunc1[TAMANHO]; // nome do funcionário 1
11     int idadeFunc1;     // idade do funcionário 1
12     char sexoFunc1;     // sexo do funcionário 1, F e M
13     float salFunc1;     // salário do funcionário 1
14
15     int codFunc2;      // código do funcionário 2
16     char nomeFunc2[TAMANHO]; // nome do funcionário 2
17     int idadeFunc2;     // idade do funcionário 2
18     char sexoFunc2;     // sexo do funcionário 2, F e M
19     float salFunc2;     // salário do funcionário 2
20
21     int resultado;      // resultado da expressão relacional
22
23     // entrada de dados do funcionário 1
24
```

```
25     printf("A seguir, entre com todos os dados do funcionario 1.\n");
26     printf("Digite o codigo: ");
27     scanf("%d", &codFunc1);
28     printf("Digite o nome: ");
29     scanf("%s", &nomeFunc1);
30     printf("Digite a idade: ");
31     scanf("%d", &idadeFunc1);
32     printf("Digite o sexo [F ou M]: ");
33     scanf("%*c%c", &sexoFunc1);
34     printf("Digite o salario (R$): ");
35     scanf("%f", &salFunc1);
36     printf("\n\n");
37
38     // entrada de dados do funcionário 2
39
40     printf("A seguir, entre com todos os dados do funcionario 2.\n");
41     printf("Digite o codigo: ");
42     scanf("%d", &codFunc2);
43     printf("Digite o nome: ");
44     scanf("%s", &nomeFunc2);
45     printf("Digite a idade: ");
46     scanf("%d", &idadeFunc2);
47     printf("Digite o sexo [F ou M]: ");
48     scanf("%*c%c", &sexoFunc2);
49     printf("Digite o salario (R$): ");
50     scanf("%f", &salFunc2);
51     printf("\n\n");
52
53     // saída de dados do funcionário 1 para a tela
54
55     printf("Os dados do funcionario 1 sao:\n");
56     printf("Codigo: %d\n", codFunc1);
57     printf("Nome: %s\n", nomeFunc1);
58     printf("Idade: %d\n", idadeFunc1);
59     printf("Sexo: %c\n", sexoFunc1);
60     printf("Salario (R$): %.2f\n", salFunc1);
61     printf("\n");
62
63     // saída de dados do funcionário 2 para a tela
64
65     printf("Os dados do funcionario 2 sao:\n");
66     printf("Codigo: %d\n", codFunc2);
67     printf("Nome: %s\n", nomeFunc2);
68     printf("Idade: %d\n", idadeFunc2);
```

```
69     printf("Sexo: %c\n", sexoFunc2);
70     printf("Salario (R$): %.2f\n", salFunc2);
71     printf("\n");
72
73     /* comparação dos dados dos funcionários
74        para produzir valores lógicos na saída padrão */
75
76     resultado = codFunc1 < codFunc2;
77     printf("codFunc1 < codFunc2 = %d\n", resultado);
78
79     resultado = codFunc1 > codFunc2;
80     printf("codFunc1 > codFunc2 = %d\n", resultado);
81
82     resultado = idadeFunc1 == idadeFunc2;
83     printf("idadeFunc1 == idadeFunc2 = %d\n", resultado);
84
85     resultado = idadeFunc1 != idadeFunc2;
86     printf("idadeFunc1 != idadeFunc2 = %d\n", resultado);
87
88     resultado = sexoFunc1 == 'M';
89     printf("sexoFunc1 == 'M' = %d\n", resultado);
90
91     resultado = sexoFunc1 != 'M';
92     printf("sexoFunc1 != 'M' = %d\n", resultado);
93
94     resultado = salFunc1 <= salFunc2;
95     printf("salFunc1 <= salFunc2 = %d\n", resultado);
96
97     resultado = salFunc1 >= salFunc2;
98     printf("salFunc1 >= salFunc2 = %d\n", resultado);
99
100    resultado = salFunc1 + 500 >= 2000;
101    printf("salFunc1 + 500 >= 2000 = %d\n", resultado);
102
103    resultado = strcmp(nomeFunc1, nomeFunc2);
104    printf("nomeFunc1 == nomeFunc2 = %d\n", resultado);
105
106    // finalização do programa principal
107
108    system("PAUSE");
109    return 0;
110 }
```

O resultado de uma expressão relacional é 0 (falso) ou 1 (verdadeiro). No Programa 3, a variável inteira “resultado” declarada na linha 21 irá armazenar o resultado de cada expressão. Note que o resultado de uma expressão relacional é armazenado somente até ser impresso na tela. Assim, apesar deste programa possuir diversas expressões, uma mesma variável temporária pode ser usada para receber o valor de cada uma das expressões. Não é necessário uma variável diferente para cada expressão relacional distinta, de acordo com a lógica deste programa. O uso de expressões relacionais localiza-se nas linhas 76 a 100. Todas as expressões são utilizadas em comandos que estão na forma “resultado = operando\_1 operador\_relacional operando\_2”. Em algumas expressões, os operandos são variáveis, como exemplo `codFunc1 < codFunc2`. Em outras expressões, um dos operandos é uma constante literal, como exemplo `sexoFunc1 == 'M'`. Há também o uso de uma expressão aritmética como um dos operandos, por exemplo `salFunc1 + 500 >= 2000` (i.e., `salFunc1 + 500` é uma expressão aritmética). A Figura 4 ilustra um exemplo de execução do Programa 3.



```
F:\Usuários\Ricardo\Programação\Dev-Cpp\LabProg\Capitulo 02\Programa 03\projeto03.exe
Os dados do funcionario 1 sao:
Codigo: 1
Nome: maria
Idade: 25
Sexo: F
Salario (R$): 1600.00

Os dados do funcionario 2 sao:
Codigo: 2
Nome: jose
Idade: 25
Sexo: M
Salario (R$): 1450.67

codFunc1 < codFunc2 = 1
codFunc1 > codFunc2 = 0
idadeFunc1 == idadeFunc2 = 1
idadeFunc1 != idadeFunc2 = 0
sexoFunc1 == 'M' = 0
sexoFunc1 != 'M' = 1
salFunc1 <= salFunc2 = 0
salFunc1 >= salFunc2 = 1
salFunc1 + 500 >= 2000 = 1
nomeFunc1 == nomeFunc2 = 1
Pressione qualquer tecla para continuar. . .
```

Figura 4. Execução do Programa 3.

Deve-se ter atenção especial na **comparação de strings** (i.e., cadeias de caracteres), tal como o nome de cada funcionário (i.e., variáveis `nomeFunc1` e `nomeFunc2`). Para fazer a comparação entre duas strings, é necessário utilizar a função `strcmp(string1, string2)` da biblioteca `string`. A linha 3 inclui o arquivo de cabeçalho desta biblioteca. Esta função retorna 0 se as strings são iguais, retorna -1 se a `string1` é menor que a `string2` e retorna 1 se a `string1` é maior que a `string2`. Os termos “menor” e “maior” são definidos de acordo com a ordem lexicográfica da tabela ASCII. Por exemplo, “Maria” é maior que “José”, desde que o caractere M tem o valor 77 e o caractere J tem o valor 74 na tabela ASCII. Já “Mariana” é menor que “Mariane”. Estes nomes possuem uma parte inicial em comum (i.e., “Marian”). Os caracteres que diferenciam estes nomes são o caractere a (valor 97) e o caractere e (valor 101). Portanto, 97 é menor que 101. Note que letras maiúsculas e minúsculas possuem valores diferentes na tabela ASCII. Por exemplo, o valor do caractere M é 77 e o valor do caractere m é 109. Isto pode alterar o significado da comparação de strings. Por fim, vale lembrar que, enquanto 0 significa falso em expressões relacionais, 0 significa que duas strings são iguais para a função `strcmp( )`. A linha 103 do Programa 3 ilustra o uso da função `strcmp( )`.

### 2.2.3 Expressões e Operadores Lógicos

Uma **expressão lógica** permite que diversas condições sejam analisadas conjuntamente para produzir como resultado um valor lógico (i.e., verdadeiro ou falso). Cada condição em uma expressão lógica, chamada **condição lógica**, geralmente é formada por uma expressão relacional, mas também pode ser constituída por uma variável que armazene um valor lógico (i.e., 0 para falso, outro número para verdadeiro) ou por uma função que retorne um valor lógico. A Equação 5 ilustra a forma geral de uma expressão lógica. Uma expressão lógica também pode ser formada apenas por um operador unário e uma condição lógica (Equação 6) e por outras formas que combinam o uso de operadores lógicos unários e binários (Equação 7). Nesta última equação, o primeiro e o terceiro



operadores representam operadores unários aplicados sobre a condição lógica à direita e o segundo operador representa um operador binário.

*CondLógica operador CondLógica ... operador CondLógica* (5)

*operador CondLógica* (6)

*operador CondLógica operador operador CondLógica* (7)

A linguagem C possui 3 **operadores lógicos**, sendo 1 operador unário (i.e., NOT lógico) e 2 operadores binários (i.e., AND lógico e OR lógico). A Tabela 6 ilustra as propriedades dos operadores lógicos. As tabelas-verdade dos operadores NOT lógico, AND lógico e OR lógico são mostradas, respectivamente, nas Tabelas Tabela 7, Tabela 8 e Tabela 9. Uma tabela-verdade determina o resultado de uma expressão lógica em função dos valores das condições lógicas.

O operador **NOT lógico** inverte o valor de uma condição lógica. Assim, se condição for verdadeira, o resultado será falso. Por outro lado, se condição for falsa, o resultado será verdadeiro. Este operador é muito utilizado com variáveis flag, as quais indicam se uma condição já ocorreu no programa. Por exemplo, para representar a condição lógica que um arquivo não acabou pode-se usar “!acabouArquivo”, onde acabouArquivo é uma variável flag que armazena 1 (verdadeiro) se o arquivo já chegou ao fim. O operador **AND lógico** somente produz o resultado 1 (verdadeiro) se ambas as condições lógicas forem verdadeiras. Caso contrário, o operador AND lógico produz o resultado 0 (falso). O operador **OR lógico** produz o resultado 1 (verdadeiro) se pelo menos uma das condições lógicas for verdadeira. Em outras palavras, o operador OR lógico produz o resultado 0 (falso) apenas se ambas as condições lógicas forem falsas.

operador	significado	unário / binário	precedência
!	NOT lógico (não)	unário	1
&&	AND lógico (e)	binário	2
	OR lógico (ou)	binário	3

Tabela 6. Operadores lógicos.

condição lógica	resultado
0 (falso)	1 (verdadeiro)
$\neq$ . 0 (verdadeiro)	0 (falso)

Tabela 7. Tabela-verdade NOT lógico.

condição lógica 1	condição lógica 2	resultado
0 (falso)	0 (falso)	0 (falso)
0 (falso)	$\neq$ . 0 (verdadeiro)	0 (falso)
$\neq$ . 0 (verdadeiro)	0 (falso)	0 (falso)
$\neq$ . 0 (verdadeiro)	$\neq$ . 0 (verdadeiro)	1 (verdadeiro)

Tabela 8. Tabela-verdade AND lógico.

condição lógica 1	condição lógica 2	resultado
0 (falso)	0 (falso)	0 (falso)
0 (falso)	$\neq$ . 0 (verdadeiro)	1 (verdadeiro)
$\neq$ . 0 (verdadeiro)	0 (falso)	1 (verdadeiro)
$\neq$ . 0 (verdadeiro)	$\neq$ . 0 (verdadeiro)	1 (verdadeiro)

Tabela 9. Tabela-verdade OR lógico.

O Programa 4 a seguir exemplifica o uso de expressões lógicas. Este programa realiza a leitura dos dados de um funcionário (i.e., dados sobre o código, o nome, a idade, o sexo e o salário do funcionário) e determina se o funcionário satisfaz um conjunto de expressões lógicas, tal como (código < 30) - ((idade < 20) / (idade > 40)) - (sexo = 'M') - (salário > R\$ 800,00). Em matemática, o símbolo - representa AND lógico e o símbolo / representa OR lógico.

**Programa 4**

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  #define TAMANHO 30 // tam. máx. do nome de um funcionário
6
7  int main(int argc, char *argv[])
8  {
9      int codFunc;      // código do funcionário
10     char nomeFunc[TAMANHO]; // nome do funcionário
11     int idadeFunc;     // idade do funcionário
12     char sexoFunc;     // sexo do funcionário, F e M
13     float salFunc;     // salário do funcionário
14
15     int resultado;     // resultado da expressão lógica
16
17     // entrada de dados do funcionário
18
19     printf("A seguir, entre com todos os dados do funcionario.\n");
20     printf("Digite o codigo: ");
21     scanf("%d", &codFunc);
22     printf("Digite o nome: ");
23     scanf("%s", &nomeFunc);
24     printf("Digite a idade: ");
25     scanf("%d", &idadeFunc);
26     printf("Digite o sexo [F ou M]: ");
27     scanf("%*c%c", &sexoFunc);
28     printf("Digite o salario (R$): ");
29     scanf("%f", &salFunc);
30     printf("\n\n");
31
32     // saída de dados do funcionário para a tela
33
34     printf("Os dados do funcionario sao:\n");
35     printf("Codigo: %d\n", codFunc);
36     printf("Nome: %s\n", nomeFunc);
37     printf("Idade: %d\n", idadeFunc);
38     printf("Sexo: %c\n", sexoFunc);
39     printf("Salario (R$): %.2f\n", salFunc);
40     printf("\n");
```

```
41
42 // expressões lógicas
43
44 /* expressão 1
45     código do funcionário entre 10 e 40, inclusive, ou seja,
46     no intervalo [10,40] */
47
48 resultado = codFunc >= 10 && codFunc <= 40;
49 printf("expressao logica 1 = %d\n", resultado);
50
51 /* expressão 2
52     idade do funcionário inferior a 20 anos ou superior a 40 anos,
53     ou seja, no intervalo [0,19] ou [41,infinito] */
54
55 resultado = idadeFunc < 20 || idadeFunc > 40;
56 printf("expressao logica 2 = %d\n", resultado);
57
58 /* expressão 3
59     sexo do funcionário masculino e
60     o seu salário deve ser superior a R$ 1000,00 */
61
62 resultado = sexoFunc == 'M' && salFunc > 1000.00;
63 printf("expressao logica 3 = %d\n", resultado);
64
65 /* expressão 4
66     código do funcionário inferior a 30 E
67     idade do funcionário inferior a 20 anos ou superior a 40 anos, E
68     sexo do funcionário masculino E
69     salário do funcionário superior a R$ 800,00 */
70
71 resultado = (codFunc < 30) && ((idadeFunc < 20) || (idadeFunc > 40)) &&
72     (sexoFunc == 'M') && (salFunc > 800.00);
73 printf("expressao logica 4 = %d\n", resultado);
74
75 /* expressão 5
76     ... uso de strings em uma condição lógica
77     sexo do funcionário masculino E
78     nome diferente de Jose */
79
80 resultado = sexoFunc == 'M' && (strcmp(nomeFunc,"Jose")!=0);
81 printf("expressao logica 5 = %d\n", resultado);
82
83 // finalização do programa principal
84
```

```
85    system("PAUSE");  
86    return 0;  
87 }
```

A linha 15 do Programa 4 declara a variável resultado, a qual armazenará o resultado de cada expressão lógica (i.e., o valor 0 para falso e o valor 1 para verdadeiro). As expressões lógicas e os seus comentários começam na linha 42 e terminam na linha 81. Para cada expressão lógica é descrito o seu enunciado em texto corrente e depois a expressão é representada na linguagem C por um comando de atribuição na forma “resultado = ...”. Note que os operadores relacionais possuem precedência sobre os operadores lógicos, ou seja, os operadores relacionais são executados antes dos operadores lógicos. Portanto, não é necessário a utilização de parênteses em expressões lógicas do tipo “codFunc >= 10 && codFunc <= 40”. No entanto, na expressão lógica 4 é necessário o uso de parênteses para representar uma disjunção de idades em uma expressão conectada por operadores AND lógico. Os demais parênteses da expressão 4 são usados apenas para facilitar a leitura da expressão. Por fim, para usar strings em uma condição lógica é necessário a chamada da função strcmp( ) para comparar o valor de uma variável com o valor de uma constante literal. Se a expressão relacional “strcmp(nomeFunc,”Jose”)!=0” for verdadeira (i.e., para nomes diferentes a função retorna -1 ou 1, que é diferente de zero), ela retornará o valor 1. O significado da função strcmp( ) já foi explicado anteriormente no texto. A Figura 5 ilustra um exemplo de execução do Programa 4.

```

F:\Usuários\Ricardo\Programação\Dev-Cpp\LabProg\Capitulo 02\Programa 04\projeto04.exe
A seguir, entre com todos os dados do funcionario.
Digite o codigo: 11
Digite o nome: Pedro
Digite a idade: 35
Digite o sexo [F ou M]: M
Digite o salario (R$): 1350.00

Os dados do funcionario sao:
Codigo: 11
Nome: Pedro
Idade: 35
Sexo: M
Salario (R$): 1350.00

expressao logica 1 = 1
expressao logica 2 = 0
expressao logica 3 = 1
expressao logica 4 = 0
expressao logica 5 = 1
Pressione qualquer tecla para continuar. . .
  
```

Figura 5. Execução do Programa 4.

## 2.2.4 Explicações Adicionais

Esta seção reforça a explicação de alguns dos conceitos estudados. Para tanto, serão usados exemplos de trechos de programas.

### Operadores Aritméticos

A linguagem C oferece dois operadores simples, compactos e eficientes para incrementar e decrementar o valor de uma variável. O **operador de incremento**, representado pelo símbolo ++, é usado para realizar operações da forma **variável = variável + 1**. Por exemplo, o comando `totalFunc = totalFunc + 1`, presente nas linhas 36, 52 e 68 do Programa 2, pode ser reescrito usando o operador de incremento como `totalFunc++`. De forma similar, o **operador de decremento**, representado pelo símbolo —, é usado para realizar operações da forma **variável = variável - 1**. Um programa C profissional sempre usa a notação dos operadores de incremento e decremento, ao invés de seus comandos equivalentes na forma variá-

vel = variável + 1 e variável = variável - 1, respectivamente. Os operadores de incremento e decremento são otimizados pelo compilador e por isso executam mais rapidamente. Além disso, o uso destes operadores deixa o programa com uma escrita mais sucinta.

Os operadores de incremento e decremento também podem ser utilizados em expressões aritméticas mais complexas. Nestas expressões, estes operadores podem assumir a forma de prefixo (i.e., ++variável e —variável) ou de sufixo (variável++ e variável—). O resultado final armazenado na variável é o mesmo tanto para a forma de prefixo quanto para a forma de sufixo. Porém, o resultado da expressão aritmética não será o mesmo. A **forma de prefixo** faz com que o valor da variável seja primeiramente incrementado (ou decrementado) para depois este valor ser usado na expressão. Já a **forma de sufixo** usa o valor atual da variável na expressão e somente depois o valor da variável é incrementado (ou decrementado). O trecho de programa a seguir ilustra o uso dos operadores de incremento e decremento nas suas formas de prefixo e de sufixo. Note também o uso do **operador vírgula** (símbolo ,). Este operador tem a semântica de “faça isto, depois faça isto, depois faça isto, ...”. Assim, no comando “x = 1, y = 3;”, x recebe o valor 1 e depois y recebe o valor 3.

```
...
int expressao;
int x, y, z, w;

x = 1, y = 3;
expressao = x + ++y;
printf("expressao = %d\n", expressao); // valor 5
printf("y = %d\n\n", y); // valor 4

x = 1, y = 3;
expressao = x + y++;
printf("expressao = %d\n", expressao); // valor 4
printf("y = %d\n\n", y); // valor 4

z = 5, w = 7;
expressao = z + —w;
printf("expressao = %d\n", expressao); // valor 11
printf("w = %d\n\n", w); // valor 6
```

```
z = 5, w = 7;
expressao = z + w—;
printf("expressao = %d\n", expressao); // valor 12
printf("w = %d\n\n", w); // valor 6
...
```

O trecho de programa a seguir mostra o uso de todos os operadores aritméticos. Note que % retorna o resto da divisão de números inteiros. Note também o uso do operador unário - para inverter o sinal da variável d.

```
...
double expressao;
float a, c;
int b, d;

a = 1.5, b = 2, c = 3.5, d = -4;
expressao = a * b / (c + d);
printf("expressao = %f\n", expressao); // valor -6.000000

expressao = a - b / (c / -d);
printf("expressao = %f\n", expressao); // valor -0,785714

expressao = (b * 4) % (-d + 1);
printf("expressao = %f\n", expressao); // valor 3.000000

expressao = b++ + —d;
printf("expressao = %f\n", expressao); // valor -3.000000
...
```

### Operadores Combinados com Atribuição

A linguagem C oferece um conjunto de operadores (i.e., +=, -=, \*=, /= e %=) que facilita a escrita de comandos da forma **variável = variável operador expressão**. Por exemplo, o comando `salFunc += 10` corresponde a `salFunc = salFunc + 10`. Em outro exemplo, o comando `aumentoArea -= 50` corresponde a `aumentoArea = aumentoArea - 50`. O uso de **operadores combinados com atribuição** produz código reduzido e sempre que possível é usado em pro-



gramas C. O trecho de programa a seguir mostra o uso de todos os operadores combinados com atribuição.

```
...
float numReal = 20.0;
int numInteiro = 20;

numReal += 5; // numReal = numReal + 5
numReal -= 15; // numReal = numReal - 15
numReal *= 4; // numReal = numReal * 4
numReal /= 2; // numReal = numReal / 2
printf("numReal = %f\n", numReal); // valor 20.000000

numInteiro %= 3; // numInteiro = numInteiro % 3
printf("numInteiro = %d\n", numInteiro); // valor 2
...
```

### Operadores Bit a Bit

Um dos objetivos principais da linguagem C é substituir o uso da **linguagem de montagem** (ou **linguagem assembler**) na escrita de programas de baixo nível (por exemplo, na interface com um dispositivo de E/S), sem que isto ocasione perda de desempenho, ou seja, o programa executável gerado pela linguagem C deve ser tão rápido quanto o programa executável gerado pela linguagem de montagem. Um programa em linguagem de montagem usa mnemônicos (i.e., símbolos especiais) para facilitar a escrita de programas em linguagem de máquina. Assim, uma sequência 00010011 que representa uma instrução de soma pode ser representada pelo mnemônico ADD. Apesar dessa facilidade, programar em linguagem de montagem ainda é muito mais difícil e improdutivo do que trabalhar em uma linguagem de mais alto nível, como a linguagem C.

A linguagem C possui um conjunto de operadores para a manipulação direta de bits (Tabela 10). Os operadores **AND bit a bit** (símbolo &), **OR bit a bit** (símbolo |) e **XOR bit a bit** (símbolo ^) permitem a comparação bit a bit de 2 números na base binária (i.e., base 2) para produzir um novo número também na base bi-

nária. Para isto, estes operadores usam as tabelas-verdade Tabela 11, Tabela 12 e Tabela 13, as quais indicam o resultado de uma comparação bit a bit de acordo com o valor dos bits dos números 1 e 2. Em uma tabela verdade, as duas primeiras colunas (i.e., número 1 e número 2) representam os dados de entrada, enquanto a terceira coluna (i.e., resultado) indica o resultado da operação. Por exemplo, na segunda linha da Tabela 11, os dados de entrada são 0 e 1. Já o resultado é 0. Portanto, 0 AND 1 produz o resultado 0.

Note que o operador AND bit a bit somente produz o resultado 1 se os bits de ambos os números 1 e 2 forem 1. Caso contrário, o operador AND bit a bit produz o resultado 0. Por outro lado, o operador OR bit a bit produz o resultado 1 se pelo menos um dos bits dos números 1 e 2 for 1. O operador OR bit a bit produz o resultado 0 apenas se os bits de ambos os números 1 e 2 forem 0. O operador XOR bit a bit (também chamado OU exclusivo), por sua vez, produz o resultado 1 se apenas um dos números 1 e 2 tiver o bit 1. Caso contrário, o operador XOR bit a bit produz o resultado 0. O trecho de programa a seguir ilustra o uso dos operadores &, | e ^. Observe o uso do tipo char para representar um número na base binária. O tipo int também poderia ter sido usado. Porém, com o tipo int a variável deve receber o seu valor em hexadecimal e deve ocupar 2 bytes (e.g., numero1 = 0x00F2 e 2 bytes considerando o Dev-C++ 5 na plataforma Windows XP).

```
...
/* apesar do tipo char,
   estas variáveis serão interpretadas como
   um número inteiro entre 0 e 255 */

unsigned char numero1, numero2;
unsigned char resultado;

numero1 = 242; // 11110010 na base 2
numero2 = 10; // 00001010 na base 2

// operador AND bit a bit
resultado = numero1 & numero2; // 00000010 na base 2
printf("resultado AND bit a bit = %d\n", resultado); // ou 2 na base 10
```

```
// operador OR bit a bit
resultado = numero1 | numero2;           // 11111010 na base 2
printf("resultado OR bit a bit = %d\n", resultado); // ou 250 na base 10

// operador XOR bit a bit
resultado = numero1 ^ numero2;           // 11111000 na base 2
printf("resultado XOR bit a bit = %d\n", resultado); // ou 248 na base 10
...
```

operador	significado	unário / binário	precedência
~	complemento de 1	unário	1
<<	deslocamento de bits à esquerda	binário	2
>>	deslocamento de bits à direita	binário	2
&	AND bit a bit	binário	3
^	XOR bit a bit	binário	4
	OR bit a bit	binário	5

Tabela 10. Operadores bit a bit

número 1	número 2	resultado
0	0	0
0	1	0
1	0	0
1	1	1

Tabela 11. Tabela-verdade AND bit a bit.

número 1	número 2	resultado
0	0	0
0	1	1
1	0	1
1	1	1

Tabela 12. Tabela-verdade OR bit a bit.

número 1	número 2	resultado
0	0	0
0	1	1
1	0	1
1	1	0

Tabela 13. Tabela-verdade XOR bit a bit.

O operador **complemento de 1** (símbolo `~`) inverte cada bit em um número na base 2, ou seja, os bits zeros tornam-se bits uns e os bits uns tornam-se bits zeros. O trecho de programa a seguir ilustra o uso do operador `~`.

```
...
unsigned char numero;
unsigned char resultado;

numero = 0;          // 00000000 na base 2 ou 0 na base 10
resultado = ~numero; // 11111111 na base 2 ou 255 na base 10
printf("resultado = %d\n", resultado);

numero = 255;        // 11111111 na base 2 ou 255 na base 10
resultado = ~numero; // 00000000 na base 2 ou 0 na base 10
printf("resultado = %d\n", resultado);

numero = 170;        // 10101010 na base 2 ou 170 na base 10
resultado = ~numero; // 01010101 na base 2 ou 85 na base 10
printf("resultado = %d\n", resultado);
...
```

Por fim, os operadores de **deslocamento de bits à esquerda** (símbolo `<<`) e de **deslocamento de bits à direita** (símbolo `>>`) deslocam os bits de um número na base 2 em `p` posições, respectivamente, à esquerda e à direita. Bits zeros são inseridos para preencher as `p` posições deslocadas. Por exemplo, o comando “variável `<< 3`” desloca os bits de variável em 3 posições à esquerda. Assim, se o valor de variável for 11111111, após a aplicação do operador `<<` o resultado será 11111000. O trecho de programa a seguir ilustra outros exemplos de uso dos operadores `>>` e `<<`.

```
...
unsigned char numero;
unsigned char resultado;

numero = 170;          // 10101010 na base 2 ou 170 na base 10
resultado = numero << 1; // 01010100 na base 2 ou 84 na base 10
printf("resultado = %d\n", resultado);
```

```
numero = 170;      // 10101010 na base 2 ou 170 na base 10
resultado = numero << 4; // 10100000 na base 2 ou 160 na base 10
printf("resultado = %d\n", resultado);

numero = 170;      // 10101010 na base 2 ou 170 na base 10
resultado = numero >> 3; // 00010101 na base 2 ou 21 na base 10
printf("resultado = %d\n", resultado);

numero = 170;      // 10101010 na base 2 ou 170 na base 10
resultado = numero >> 6; // 00000010 na base 2 ou 2 na base 10
printf("resultado = %d\n", resultado);
...
```

### Precedência de Operadores

A precedência descrita para os operadores aritméticos (Tabela 3), operadores relacionais (Tabela 5) e operadores lógicos (Tabela 6) tem significado local, ou seja, é válida apenas entre operadores do mesmo tipo. Portanto, o fato do operador lógico ! ter precedência 1 e o operador relacional < também ter precedência 1 não significa que ambos os operadores possuem a mesma precedência global entre operadores (ou ordem global de precedências de operadores). De fato, o operador lógico ! possui precedência sobre o operador relacional <. As regras de precedência não são triviais na linguagem C, principalmente devido ao uso de um mesmo símbolo para operadores distintos. O livro “C Completo e Total”, 3ª Edição, de Herbert Schildt, descreve na página 56 (Tabela 2.8) a precedência global dos operadores na linguagem C.

### Expressões Literais

A linguagem C não possui nenhum operador específico para tratar de expressões literais. Na apostila da disciplina de “Construção de Algoritmos” foi apresentado o operador de concatenação (i.e., símbolo +). Em C, a concatenação de strings e caracteres é realizada por meio da função strcat( ) da biblioteca string. A sua sintaxe é strcat(string1, string2). Esta função concatena uma có-

pia de string2 em string1 e adiciona ao seu final \0. O resultado é equivalente a “string1 + string2” em algoritmo.

## 2.2.5 Mapeamento de Algoritmo para Programa C

A seguir são apresentados os mapeamentos de 3 algoritmos para programas C.

### Algoritmo 3-1 (apostila de “Construção de Algoritmos”)

```
1 { apresentar, para uma conta de lanchonete, o valor total da conta,  
2   o valor pago e o troco devolvido, dados o total consumido de bebidas  
3   e alimentos, além de se saber que a porcentagem do garçom é 10% }  
4  
5   algoritmo  
6       declare totalBebidas, totalAlimentos,  
7           valorPago, troco,  
8           porcentagemGarçom: real  
9  
10      { obtenção dos dados necessários }  
11      leia(totalBebidas, totalAlimentos, valorPago)  
12  
13      { cálculos necessários }  
14      porcentagemGarçom  $\leftarrow$  (totalBebidas + totalAlimentos) *  
15                               10/100,0 { 10% }  
16      troco  $\leftarrow$  valorPago - (totalBebidas + totalAlimentos +  
17                               porcentagemGarçom)  
18  
19      { apresentação dos resultados }  
20      escreva("Valor da conta:",  
21              totalBebidas + totalAlimentos + porcentagemGarçom)  
22      escreva("Valor pago:", valorPago)  
23      escreva("Troco:", troco)  
24      fim-algoritmo
```

**Programa 5 (equivalente ao algoritmo 3-1)**

```
1  /* apresentar, para uma conta de lanchonete, o valor total da conta,  
2  o valor pago e o troco devolvido, dados o total consumido de bebidas  
3  e alimentos, além de se saber que a porcentagem do garçom é 10% */  
4  
5  #include <stdio.h>  
6  #include <stdlib.h>  
7  
8  int main(int argc, char *argv[])  
9  {  
10     float totalBebidas, totalAlimentos, valorPago,  
11         troco, porcentagemGarcom;  
12  
13     // obtenção dos dados necessários  
14     printf("Digite o total de bebidas: ");  
15     scanf("%f", &totalBebidas);  
16     printf("Digite o total de alimentos: ");  
17     scanf("%f", &totalAlimentos);  
18     printf("Digite o valor pago: ");  
19     scanf("%f", &valorPago);  
20     printf("\n");  
21  
22     // cálculos necessários  
23     porcentagemGarcom = (totalBebidas + totalAlimentos) * 10 / 100.00;  
24     troco = valorPago - (totalBebidas + totalAlimentos +  
25         porcentagemGarcom);  
26  
27     // apresentação dos resultados  
28     printf("Valor da conta: %.2f\n", totalBebidas + totalAlimentos +  
29         porcentagemGarcom);  
30     printf("Valor pago: %.2f\n", valorPago);  
31     printf("Troco: %.2f\n\n", troco);  
32  
33     system("PAUSE");  
34     return 0;  
35 }
```

### Algoritmo 3-2 (apostila de “Construção de Algoritmos”)

```
1  { dados, separadamente, prenome e sobrenome, escrevê-los em dois
2  formatos: “Fulano Tal” e “Tal, F” }
3
4  algoritmo
5      declare prenome, sobrenome,
6          formato1, formato2: literal
7
8      { obtenção dos nomes }
9      leia(prenome, sobrenome)
10
11     { composição dos nomes }
12     formato1 ← prenome + “ ” + sobrenome
13     formato2 ← sobrenome + “,” + subLiteral(prenome, 1, 1) + “.”
14     { obs.: subLiteral retorna uma parte do literal }
15
16     { resultados }
17     escreva(formato1)
18     escreva(formato2)
19 fim-algoritmo
```

### Programa 6 (equivalente ao algoritmo 3-2)

```
1  /* dados, separadamente, prenome e sobrenome, escrevê-los em dois
2  formatos: “Fulano Tal” e “Tal, F.” */
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string.h>
6
7  int main(int argc, char *argv[])
8  {
9      // tamanho de formato1 = prenome + sobrenome + espaço + \0
10     // tamanho de formato2 = prenome + sobrenome + “,” + “.” + \0
11     char prenome[30], sobrenome[50], formato1[82], formato2[84];
12     char temp[2]; // string para receber primeira letra de prenome
13
14     // obtenção dos nomes
15     printf(“Digite o prenome: “);
16     scanf(“%s”, &prenome);
17     printf(“Digite o sobrenome: “);
18     scanf(“%s”, &sobrenome);
```



```
19     printf("\n");
20
21     // composição dos nomes
22
23     strcpy(formato1, prenome);
24     strcat(formato1, " ");
25     strcat(formato1, sobrenome);
26
27     strcpy(formato2, sobrenome);
28     strcat(formato2, " ");
29     strncpy(temp, prenome, 1);
30     strcat(formato2, temp);
31     strcat(formato2, " ");
32
33     // resultados
34     printf("formato 1 = %s\n", formato1);
35     printf("formato 2 = %s\n\n", formato2);
36
37     system("PAUSE");
38     return 0;
39 }
```

Com relação ao Programa 6, algumas explicações adicionais são necessárias. Em um algoritmo, um comando pode concatenar 2 ou mais cadeias de caracteres de uma só vez. Por exemplo, o comando `prenome + " " + sobrenome` concatena 3 cadeias de caracteres usando 2 operadores de concatenação (i.e., símbolo +). Na linguagem C, isto não é possível. Assim, o processo de concatenação de 2 ou mais cadeias de caracteres é dividido em vários passos. Primeiramente, a primeira cadeia deve ser copiada na variável que irá formar a concatenação (linhas 23 e 27). Este passo é realizado utilizando a função `strcpy(string1, string2)` da biblioteca `string`. Esta função copia `string2` em `string1`. Os próximos passos consistem em concatenações aos pares usando a função `strcat( )` da biblioteca `string` (linhas 24 a 25 e linhas 28 a 31). Além disso, para realizar a implementação de `subLiteral(prenome, 1, 1)` foi usada a função `strncpy(string1, string2, k)` que copia os `k` primeiros caracteres de `string2` em `string1`. No caso, copiou-se o primeiro caractere de `prenome`.

### Algoritmo 3-3 (apostila de “Construção de Algoritmos”)

```

1      { determinar, dados os comprimentos dos lados de um triângulo, se
0
2      triângulo pode ser formado (se existe) e qual o tipo dele (equilátero,
3      isósceles ou escaleno) }
4
5      algoritmo
6          declare
7              lado1, lado2, lado3: real
8              existeTriângulo,
9              éEquilátero, éIsósceles, éEscaleno: lógico
10
11          { leitura dos comprimentos dos lados }
12          leia(lado1, lado2, lado3)
13
14          { verificações necessárias }
15          existeTriângulo ← lado1 < lado2 + lado3 e
16                          lado2 < lado1 + lado3 e
17                          lado3 < lado1 + lado2
18          éEquilátero ← existeTriângulo e lado1=lado2 e lado2=lado3
19          éIsósceles ← existeTriângulo e não éEquilátero e
20                      (lado1 = lado2 ou lado2 = lado3 ou lado1 = lado3)
21          éEscaleno ← existeTriângulo e não éEquilátero e não éIsósceles
22
23          { resultados }
24          escreva(“Triângulo existe?”, existeTriângulo)
25          escreva(“É equilátero?”, éEquilátero)
26          escreva(“É isósceles?”, éIsósceles)
27          escreva(“É escaleno?”, éEscaleno)
28      fim-algoritmo

```

### Programa 7 (equivalente ao algoritmo 3-3)

```

1      /* determinar, dados os comprimentos dos lados de um triângulo, se o
2      triângulo pode ser formado (se existe) e qual o tipo dele (equilátero,
3      isósceles ou escaleno) */
4
5      #include <stdio.h>
6      #include <stdlib.h>
7
8      int main(int argc, char *argv[])

```

```
9      {
10      float lado1, lado2, lado3;
11      int existeTriangulo, eEquilatero, eIsosceles, eEscaleno;
12
13      // leitura dos comprimentos dos lados
14      printf("Digite o lado 1:");
15      scanf("%f", &lado1);
16      printf("Digite o lado 2:");
17      scanf("%f", &lado2);
18      printf("Digite o lado 3:");
19      scanf("%f", &lado3);
20
21      // verificações necessárias
22      existeTriangulo = lado1 < lado2 + lado3 &&
23                      lado2 < lado1 + lado3 &&
24                      lado3 < lado1 + lado2;
25      eEquilatero = existeTriangulo && lado1 == lado2 && lado2 ==
lado3;
26      eIsosceles = existeTriangulo && !eEquilatero &&
27                  (lado1 == lado2 || lado2 == lado3 || lado1 == lado3);
28      eEscaleno = existeTriangulo && !eEquilatero && !eIsosceles;
29
30      // resultados
31      printf("Triangulo existe? %d\n", existeTriangulo);
32      printf("Eh equilatero? %d\n", eEquilatero);
33      printf("Eh isosceles?? %d\n", eIsosceles);
34      printf("Eh escaleno? %d\n\n", eEscaleno);
35
36      system("PAUSE");
37      return 0;
38      }
```

## 2.3 Comandos Condicionais

Até o presente momento, todos os programas apresentados na apostila seguiram integralmente a **regra de execução dos comandos**. Esta regra estabelece que: (1) os comandos são executados seqüencialmente e na ordem em que estão apresentados; e (2) O próximo comando somente é executado quando o comando anterior já tiver terminado. Assim, o fluxo de execução dos programas anteriormente descritos foi estritamente seqüencial.

No entanto, a resolução de problemas usando programas frequentemente necessita alterar o fluxo de execução. Por exemplo, em um trecho do programa pode ser necessário executar um conjunto de comandos somente se uma certa condição lógica for satisfeita (i.e., for verdadeira). Para alterar o fluxo de execução de um programa utiliza-se **comandos condicionais**, também conhecidos por **comandos de seleção** ou **comandos de processamento condicional**. Nesta apostila serão estudados os comandos **if-else** e **switch**.

No restante desta seção, você aprenderá sobre os seguintes **conceitos**:

comando if-else, blocos de comandos, legibilidade e indentação, aninhamento de comandos if-else, operador condicional ?, comando switch e comando break.

### 2.3.1 Comando Condicional if-else

O **formato completo** de um **comando if-else** é ilustrado na Figura 6. O comando if-else da linguagem C funciona da mesma forma que o comando se-então-senão-fim-se de algoritmos. Se a expressão lógica for verdadeira (i.e., se tiver um valor diferente de zero), o conjunto de comandos 1 será executado e depois o fluxo de execução do programa será desviado para o próximo comando após o comando if-else. Se a expressão lógica for falsa (i.e., se tiver um valor igual a zero), o conjunto de comandos 2 será executado e o fluxo de execução do programa continua no próximo comando após o comando if-else. Note que as palavras reservadas “if” e “else” da linguagem C correspondem, respectivamente, às palavras reservadas “se” e “senão” de algoritmos. No entanto, o comando if-else é mais enxuto e omite a palavra reservada “então” de algoritmos. O comando if-else também pode omitir a cláusula else (**formato simples** de um comando if-else), conforme ilustrado na Figura 7. Neste caso, se a expressão lógica for falsa, nenhum comando é executado e o fluxo de execução do programa continua no próximo comando após o comando if-else.

```
if (expressão lógica)
{
    // conjunto de comandos 1
}
else
{
    // conjunto de comandos 2
}
// próximo comando após if-else
```

Figura 6. Formato completo de um comando if-else.

```
if (expressão lógica)
{
    // conjunto de comandos 1
}
// próximo comando após if-else
```

Figura 7. Formato simples de um comando if-else.

**Blocos de comandos** na linguagem C são representados entre os símbolos { e }. Um bloco de comandos representa a execução de um conjunto de comandos e pode possuir declarações locais de variáveis e constantes, as quais são válidas apenas dentro do próprio bloco de comandos. Note que o comando if-else possui 2 blocos de comandos (i.e., conjunto de comandos 1 e conjunto de comandos 2 da Figura 6). O primeiro bloco trata do resultado verdadeiro da expressão lógica, enquanto o segundo bloco de comandos trata do resultado falso da expressão lógica. No comando if-else, se um bloco de comandos possui um único comando, os símbolos { e } podem ser omitidos, ou seja, são opcionais (Figura 8).

```
if (expressão lógica)
    comando;
else
    comando;
// próximo comando após if-else
```

Figura 8. Comando if-else sem os símbolos { e }.

A **legibilidade** de um comando if-else (i.e., a sua facilidade de leitura e compreensão) depende diretamente da forma como este comando é apresentado no programa-fonte, ou seja, depende da sua **identação**. A identação mais indicada para o comando if-else é mostrada na Figura 6. Programadores C, no entanto, também usam outras alternativas de identação, conforme ilustrado na Figura 9. Esta alternativa garante a economia de uma linha de código por bloco de comandos, mas torna um pouco mais difícil a checagem dos pares de { e }. Para o compilador, a identação do comando if-else não influencia no código gerado para o programa executável. Assim, o comando if-else poderia ser escrito até em uma única linha. Porém, a escrita do comando if-else em uma única linha ou usando uma identação diferente das alternativas ilustradas nesta apostila pode dificultar a sua legibilidade e não é considerado uma boa prática de programação.

```
if (expressão lógica) {  
    // conjunto de comandos 1  
}  
else {  
    // conjunto de comandos 2  
}  
// próximo comando após if-else
```

Figura 9. Identação alternativa para o comando if-else.

O Programa 8 a seguir exemplifica o uso do comando condicional if-else. Este programa realiza a leitura dos dados de 3 funcionários (i.e., dados sobre o código, o nome, a idade, o sexo e o salário de cada funcionário) e contabiliza: (1) quantos funcionários possuem código inferior a 100 (i.e., códigos relativos a uma certa filial da empresa); (2) quantos funcionários chamam-se José; (3) quantos funcionários possuem idade superior a 60 anos; (4) quantos funcionários são do sexo feminino; e (5) quantos funcionários recebem um salário entre R\$ 1000,00 e R\$ 2000,00 reais. O resultado de todos os cálculos são no final mostrados na saída padrão.

**Programa 8**

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  #define TAMANHO 30 // tam. máx. do nome de um funcionário
6
7  int main(int argc, char *argv[])
8  {
9      int codFunc;      // código do funcionário
10     char nomeFunc[TAMANHO]; // nome do funcionário
11     int idadeFunc;     // idade do funcionário
12     char sexoFunc;     // sexo do funcionário, F e M
13     float salFunc;     // salário do funcionário
14
15     int codigo100 = 0; // quantos funcionários com código inferior a
100
16     int nomeJose = 0; // quantos funcionários chamam-se José
17     int idade60 = 0; // quantos funcionários com idade superior a 60
18     int mulheres = 0; // quantos funcionários são do sexo feminino
19     int salFaixa = 0; /* quantos funcionários recebem um salário entre
20                       R$ 1000,00 e R$ 2000,00 reais. */
21
22     /* entrada de dados do funcionário 1 e
23        cálculos intermediários */
24
25     printf("A seguir, entre com todos os dados do funcionario 1.\n");
26     printf("Digite o codigo: ");
27     scanf("%d", &codFunc);
28     printf("Digite o nome: ");
29     scanf("%s", &nomeFunc);
30     printf("Digite a idade: ");
31     scanf("%d", &idadeFunc);
32     printf("Digite o sexo [F ou M]: ");
33     scanf("%*c%c", &sexoFunc);
34     printf("Digite o salario (R$): ");
35     scanf("%f", &salFunc);
36     printf("\n\n");
37
38     if (codFunc < 100)
39     {
40         codigo100++;
41     }
```

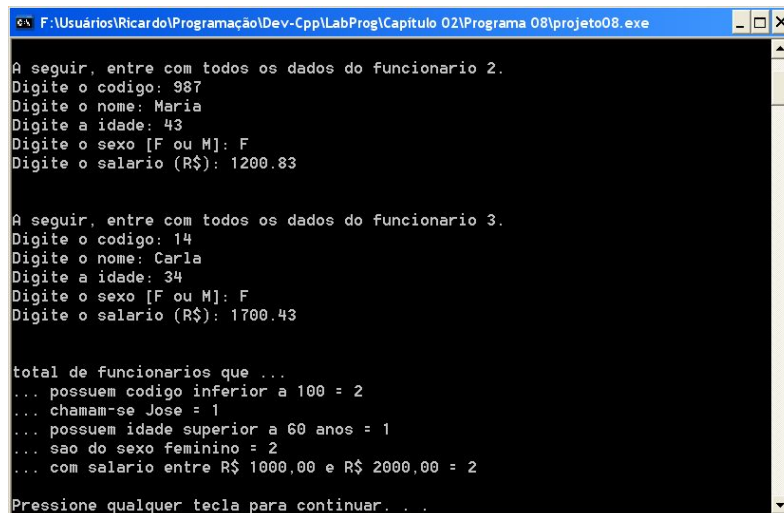
```
42
43     if (strcmp(nomeFunc,"Jose") == 0)
44     {
45         nomeJose++;
46     }
47
48     if (idadeFunc > 60)
49     {
50         idade60++;
51     }
52
53     if (sexoFunc == 'F')
54     {
55         mulheres++;
56     }
57
58     if (salFunc >= 1000.00 && salFunc <= 2000.00)
59     {
60         salFaixa++;
61     }
62
63     /* entrada de dados do funcionário 2 e
64        cálculos intermediários */
65
66     printf("A seguir, entre com todos os dados do funcionario 2.\n");
67     printf("Digite o codigo: ");
68     scanf("%d", &codFunc);
69     printf("Digite o nome: ");
70     scanf("%s", &nomeFunc);
71     printf("Digite a idade: ");
72     scanf("%d", &idadeFunc);
73     printf("Digite o sexo [F ou M]: ");
74     scanf("%c%c", &sexoFunc);
75     printf("Digite o salario (R$): ");
76     scanf("%f", &salFunc);
77     printf("\n\n");
78
79     if (codFunc < 100)
80         codigo100++;
81
82     if (strcmp(nomeFunc,"Jose") == 0)
83         nomeJose++;
84
85     if (idadeFunc > 60)
```



```
86     idade60++;
87
88     if (sexoFunc == 'F')
89         mulheres++;
90
91     if (salFunc >= 1000.00 && salFunc <= 2000.00)
92         salFaixa++;
93
94     /* entrada de dados do funcionário 3 e
95        cálculos intermediários */
96
97     printf("A seguir, entre com todos os dados do funcionario 3.\n");
98     printf("Digite o codigo: ");
99     scanf("%d", &codFunc);
100    printf("Digite o nome: ");
101    scanf("%s", &nomeFunc);
102    printf("Digite a idade: ");
103    scanf("%d", &idadeFunc);
104    printf("Digite o sexo [F ou M]: ");
105    scanf("%*c%c", &sexoFunc);
106    printf("Digite o salario (R$): ");
107    scanf("%f", &salFunc);
108    printf("\n\n");
109
110    if (codFunc < 100)
111        codigo100++;
112
113    if (strcmp(nomeFunc, "Jose") == 0)
114        nomeJose++;
115
116    if (idadeFunc > 60)
117        idade60++;
118
119    if (sexoFunc == 'F')
120        mulheres++;
121
122    if (salFunc >= 1000.00 && salFunc <= 2000.00)
123        salFaixa++;
124
125    // saída dos dados calculados
126
127    printf("total de funcionarios que ...\n");
128    printf("... possuem codigo inferior a 100 = %d\n", codigo100);
129    printf("... chamam-se Jose = %d\n", nomeJose);
```

```
130 printf("... possuem idade superior a 60 anos = %d\n", idade60);
131 printf("... sao do sexo feminino = %d\n", mulheres);
132 printf("... com salario entre 1000,00 e 2000,00 = %d\n\n", salFaixa);
133
134 // finalização do programa principal
135
136 system("PAUSE");
137 return 0;
138 }
```

Apesar do Programa 8 realizar a leitura dos dados de 3 funcionários, os dados de cada funcionário são necessários apenas até que cálculos intermediários sejam feitos, logo em seguida à leitura destes dados. Portanto, um mesmo conjunto de variáveis pode ser usado para ler os dados dos 3 funcionários, conforme a declaração de variáveis nas linhas 9 a 13. Note que a declaração de cada variável contador do programa (i.e., variáveis `codigo100`, `nomeJose`, `idade60`, `mulheres` e `salFaixa` declaradas nas linhas 15 a 19) inclui a inicialização do contador com o valor zero. Todos os comandos `if-else` ilustrados no Programa 8 estão no formato simples (i.e., sem a cláusula `else`). Enquanto nas linhas 38 a 61 os comandos `if-else` possuem os símbolos delimitadores de blocos de comandos (i.e., símbolos `{ e }`), apesar de terem um único comando, nas linhas 79 a 92 e 110 a 123 os comandos `if-else` são escritos de forma compacta, ou seja, sem os símbolos delimitadores de blocos. A lógica dos comandos condicionais é que se a expressão lógica for verdadeira, o respectivo contador é incrementado. Caso contrário, nada é feito. No final, os contadores possuem o total de funcionários que atendem ao seu critério. A Figura 10 ilustra um exemplo de execução do Programa 8.



```
F:\Usuários\Ricardo\Programação\Dev-Cpp\LabProg\Capitulo 02\Programa 08\projeto08.exe
A seguir, entre com todos os dados do funcionario 2.
Digite o codigo: 987
Digite o nome: Maria
Digite a idade: 43
Digite o sexo [F ou M]: F
Digite o salario (R$): 1200.83

A seguir, entre com todos os dados do funcionario 3.
Digite o codigo: 14
Digite o nome: Carla
Digite a idade: 34
Digite o sexo [F ou M]: F
Digite o salario (R$): 1700.43

total de funcionarios que ...
... possuem codigo inferior a 100 = 2
... chamam-se Jose = 1
... possuem idade superior a 60 anos = 1
... sao do sexo feminino = 2
... com salario entre R$ 1000,00 e R$ 2000,00 = 2

Pressione qualquer tecla para continuar. . .
```

Figura 10. Execução do Programa 8.

### 2.3.2 Comando switch

O **comando switch** é um comando condicional de seleção múltipla que facilita a comparação de igualdade entre uma variável (do tipo int ou char) e uma lista de constantes. O comando switch da linguagem C está relacionado ao comando caso de algoritmos. O formato geral de um comando switch é ilustrado na Figura 11.

O funcionamento deste comando é o seguinte. O valor da variável na cláusula switch é comparado com o valor da constante 1. Se o resultado desta comparação for verdadeiro (i.e., variável == constante 1), a seqüência de comandos 1 é executada até encontrar o comando break. Em especial, o **comando break** desvia o fluxo de execução para o próximo comando após o switch. Se o resultado da comparação anterior for falso (i.e., variável != constante 1), passa-se para a análise da constante 2. Assim, o valor da variável na cláusula switch é comparado com o valor da constante 2. Se o resultado desta comparação for verdadeiro (i.e., variável

== constante 2), a sequência de comandos 2 é executada até encontrar o comando break. Este processo segue de forma similar para as demais constantes (e.g., constante 3, constante 4, ...). Caso o valor da variável na cláusula switch não seja igual ao valor de nenhuma constante nas cláusulas case, a sequência de comandos padrão da cláusula default é executada e o comando switch é finalizado. Note que a cláusula default não é obrigatória em um comando switch. Assim, se a cláusula default não existir, nenhum comando é executado se o valor da variável na cláusula switch não for igual ao valor de nenhuma constante nas cláusulas case.

```
switch(variável)
{
    case constante1:
        // sequência de comandos 1
        break;
    case constante2:
        // sequência de comandos 2
        break;
    case constante3:
        // sequência de comandos 3
        break;
    case constante4:
        // sequência de comandos 4
        break;
    ...
    default:
        // opção padrão usada para valores diferentes
        // sequência de comandos padrão
}
// próximo comando após o switch
```

Figura 11. Formato geral do comando switch.

O Programa 9 a seguir exemplifica o uso do comando switch. Este programa realiza a leitura de uma opção do menu e de acordo com a opção escolhida o programa executa ações apropriadas, que no caso consistem simplesmente da exibição de uma mensagem na saída padrão. Neste programa, a variável na cláusula switch é do tipo char. Portanto, cada constante nas cláusulas case deve ser especificada entre aspas simples (e.g., 'I'). Isto não é necessário quan-

do a variável na cláusula switch é do tipo int. A Figura 12 ilustra um exemplo de execução do Programa 9.

### Programa 9

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[])
5  {
6      char opcao; // opção do menu
7
8      // leitura da opção
9      printf("Digite a opcao [I, R, M, P, O]: ");
10     scanf("%c", &opcao);
11     printf("\n");
12
13     // de acordo com a opção, escolhe a ação
14     // uso de comando condicional de seleção múltipla
15     switch(opcao)
16     {
17         case 'I':
18             // I para inserir um novo funcionário
19             printf("opcao escolhida: I\n\n");
20             break;
21         case 'R':
22             // R para remover um funcionário
23             printf("opcao escolhida: R\n\n");
24             break;
25         case 'M':
26             // M para modificar os dados de um funcionário
27             printf("opcao escolhida: M\n\n");
28             break;
29         case 'P':
30             // P para pesquisar os dados de um funcionário
31             printf("opcao escolhida: P\n\n");
32             break;
33         case 'O':
34             // O para ordenar o cadastro de funcionários
35             printf("opcao escolhida: O\n\n");
36             break;
37         default:
38             // opção padrão
```

```
39      // usada para valores diferentes dos anteriores
40      printf("opcao escolhida diferente de [I, R, M, P, O]\n");
41      printf("opcao escolhida = %c\n\n", opcao);
42  }
43
44      system("PAUSE");
45      return 0;
46  }
```

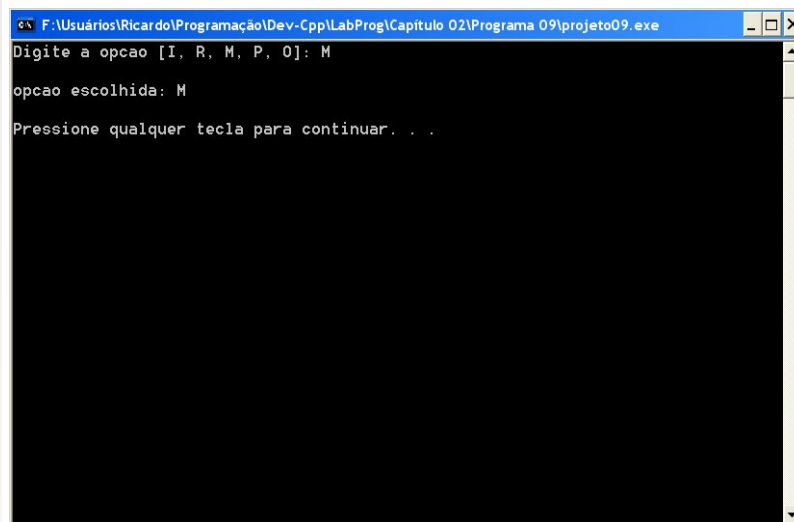


Figura 12. Execução do Programa 9.

### 2.3.3 Explicações Adicionais

Esta seção reforça a explicação de alguns dos conceitos estudados. Para tanto, serão usados exemplos de trechos de programas.

#### Comando if-else

O fluxo de execução de um comando condicional if-else é ilustrado na Figura 13 de acordo com o seu formato completo da Figura 6.

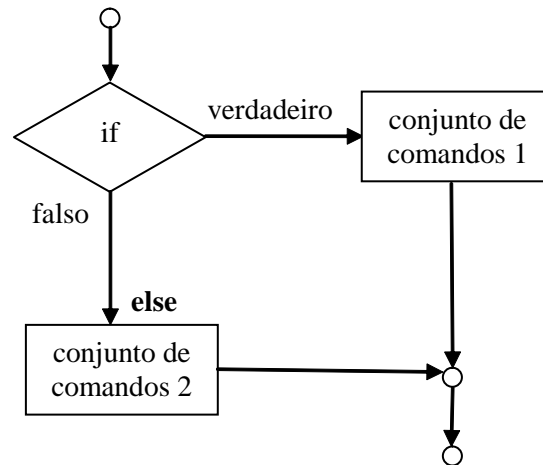


Figura 13. Representação gráfica do fluxo de execução de um comando if-else.

### Aninhamento de Comandos if-else

Um comando if-else pode ter dentro de seus blocos de comandos outros comandos if-else. Quando isto acontece, tem-se um **aninhamento** de comandos if-else. O trecho de programa a seguir ilustra esta situação. Uma cláusula else sempre pertence ao comando if-else anterior mais próximo que ainda não possui a cláusula else. Note como a indentação dos comandos if-else e o uso dos delimitadores de blocos de comandos (i.e., símbolos { e }) facilitam o entendimento do trecho a seguir (Figura 14).

```

...
if(salario > 3000.00)
{
    // seqüência de comandos 1
    // de (salario > 3000.00) verdadeiro
}
else if (sexo == 'F')
{
    // seqüência de comandos 2
    // de (sexo == 'F') verdadeiro
}
  
```

```

else if (idade < 53)
{
    // sequência de comandos 3
    // de (idade < 53) verdadeiro
}
...
  
```

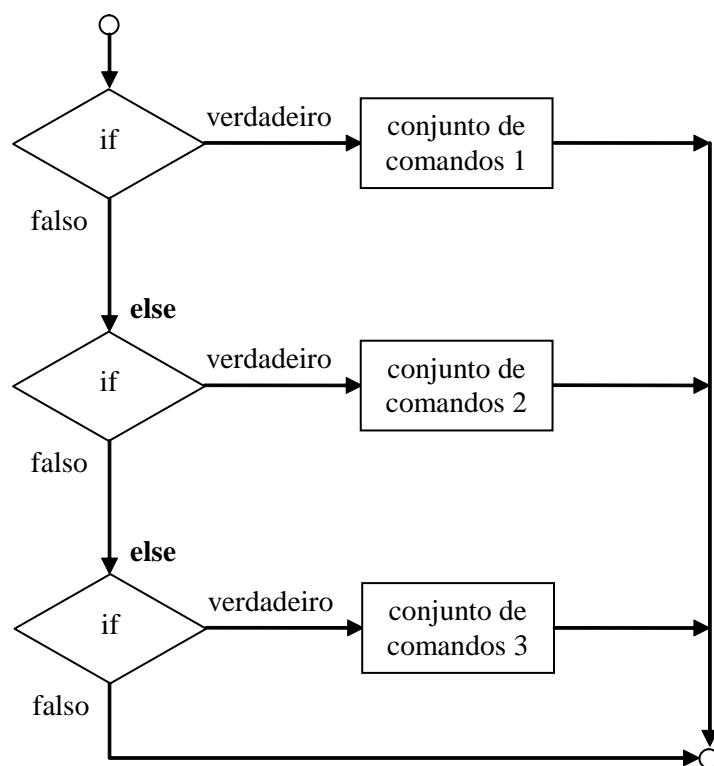


Figura 14. Representação gráfica do fluxo de execução de um comando if-else aninhado.

### Operador Condicional ?

A linguagem C possui um operador condicional reduzido na forma: **variável = expressão 1 ? expressão 2 : expressão 3**. Primei-



ramente, expressão 1 é avaliada. Se expressão 1 for verdadeira, variável recebe o valor de expressão 2, caso contrário, variável recebe o valor de expressão 3. O trecho de programa a seguir ilustra o uso do operador condicional ?.

```
...
aumentoSalarial = vendas > 110 ? 500.00 : 150.00 ;

/* o comando acima equivale ao comando
if (vendas > 110)
    aumentoSalarial = 500.00;
else aumentoSalarial = 150.00;
*/
...
```

### Comando switch

O fluxo de execução de um comando condicional switch é ilustrado na Figura 15 de acordo com o seu formato da Figura 11. Na Figura 10, “v == c1” significa que o valor da variável na cláusula switch é igual ao valor da constante 1, “v == c2” significa que o valor da variável na cláusula switch é igual ao valor da constante 2, etc.

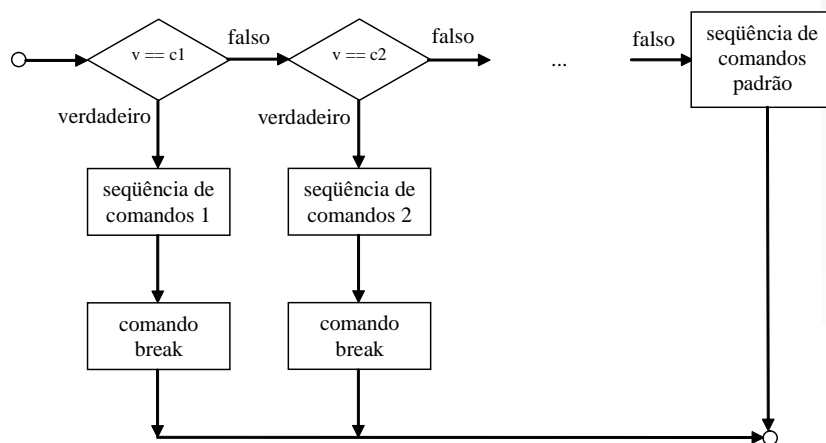


Figura 15. Representação gráfica do fluxo de execução de um comando switch.

### 2.3.4 Mapeamento de Algoritmo para Programa C

A seguir são apresentados os mapeamentos de 3 algoritmos para programas C.

#### Algoritmo 5-1 (apostila de “Construção de Algoritmos”)

```
1      { cálculo da média final de um aluno, feita pela média aritmética de
2      duas provas, e escrever mensagem indicando aprovação ou repro-
3      vação }
4      algoritmo
5          declare notaProva1, notaProva2, média: real
6
7          { leitura das notas }
8          leia(notaProva1, notaProva2)
9
10         { cálculo da média }
11         média ← (notaProva1 + notaProva2)/2
12
13         { resultados }
14         escreva(média)
15         se média >= 6 então
16             escreva(“Aprovado :...)”)
17         senão
18             escreva(“Reprovado :(...)”)
19         fim-se
20     fim-algoritmo
```

#### Programa 10 (equivalente ao algoritmo 5-1)

```
1      /* cálculo da média final de um aluno, feita pela média aritmética de
2      duas provas, e escrever mensagem indicando aprovação ou reprovação */
3
4      #include <stdio.h>
5      #include <stdlib.h>
6
7      int main(int argc, char *argv[])
8      {
9          float notaProva1, notaProva2, media;
10     }
```

```

11 // leitura das notas
12 printf("Digite a nota 1: ");
13 scanf("%f", &notaProva1);
14 printf("Digite a nota 2: ");
15 scanf("%f", &notaProva2);
16
17 // cálculo da média
18 media = (notaProva1 + notaProva2) / 2;
19
20 // resultados
21 printf("media = %.2f\n", media);
22 if (media >= 6)
23     printf("Aprovado :...\n\n");
24 else printf("Reprovado :(...\n\n");
25
26 system("PAUSE");
27 return 0;
28 }

```

### Algoritmo 5-3 (apostila de “Construção de Algoritmos”)

```

1 { determinar, dados os comprimentos dos lados de um triângulo, se o
2   triângulo pode ser formado (se existe) e qual o tipo dele (equilátero,
3   isósceles ou escaleno) }
4
5   algoritmo
6       declare
7           lado1, lado2, lado3: real
8       classificacao: literal
9
10      { leitura dos comprimentos dos lados }
11      leia(lado1, lado2, lado3)
12
13      se lado1 e" lado2 + lado3 ou lado2 e" lado1 + lado3 ou { existe? }
14          lado3 e" lado1 + lado2 então
15          escreva("Os lados fornecidos não formam um triângulo")
16      senão
17          se lado1 = lado2 e lado2 = lado3 então { lados iguais? }
18              classificacao ← "Equilátero"
19          senão
20              se lado1 = lado2 ou lado1 = lado3 ou { isósceles? }
21                  lado2 = lado3 então
22                  classificacao ← "Isósceles"

```

```

23          senão
24              classificação ← “Escaleno”
25          fim-se
26      fim-se
27
28      { escrita da classificação }
29      escreva(classificação)
30  fim-se
31 fim-algoritmo

```

### Programa 11 (equivalente ao algoritmo 5-3)

```

1  /* determinar, dados os comprimentos dos lados de um triângulo, se o
2     triângulo pode ser formado (se existe) e qual o tipo dele (equilátero,
3     isósceles ou escaleno) */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <string.h>
8
9  int main(int argc, char *argv[])
10 {
11     float lado1, lado2, lado3;
12     char classificacao[11] = “nenhuma”;
13
14     // leitura dos comprimentos dos lados
15     printf(“Digite o lado 1: “);
16     scanf(“%f”, &lado1);
17     printf(“Digite o lado 2: “);
18     scanf(“%f”, &lado2);
19     printf(“Digite o lado 3: “);
20     scanf(“%f”, &lado3);
21     printf(“\n”);
22
23     // cálculos
24
25     if ((lado1 >= lado2 + lado3) ||
26         (lado2 >= lado1 + lado3) ||
27         (lado3 >= lado1 + lado2))
28         printf(“Os lados fornecidos nao formam um triangulo\n”);
29     else
30     {
31         if ((lado1 == lado2) && (lado2 == lado3))

```

```
32     strcpy(classificacao,"Equilatero");
33     else
34     {
35         if ((lado1 == lado2) || (lado1 == lado3) || (lado2 == lado3))
36             strcpy(classificacao,"Isosceles");
37         else
38             strcpy(classificacao,"Escaleno");
39     }
40 }
41
42 // escrita da classificação
43 printf("classificacao = %s\n\n", classificacao);
44
45 system("PAUSE");
46 return 0;
47 }
```

#### Algoritmo 5-4 (apostila de “Construção de Algoritmos”)

```
1     { classificação da faixa etária segundo um critério arbitrário }
2
3     algoritmo
4         declare idade: inteiro
5
6         { leitura da idade }
7         leia(idade)
8
9         { classificação }
10        caso idade seja
11            0:         escreva("bebê")
12            1..10:     escreva("criança")
13            11..14:   escreva("pré-adolescente")
14            15..18:   escreva("adolescente")
15            19..120:  escreva("adulto")
16        senão
17            escreva("Idade inválida ou sem classificação definida")
18        fim-caso
19    fim-algoritmo
```

### Programa 12 (equivalente ao algoritmo 5-4)

```
1 // classificação da faixa etária segundo um critério arbitrário
2
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 int main(int argc, char *argv[])
7 {
8     int idade;
9
10    // leitura da idade
11    printf("Digite a idade: ");
12    scanf("%d", &idade);
13
14    // classificação
15    switch(idade)
16    {
17        case 0:
18            printf("bebe\n\n");
19            break;
20        case 1:
21        case 2:
22        case 3:
23        case 4:
24        case 5:
25        case 6:
26        case 7:
27        case 8:
28        case 9:
29        case 10:
30            printf("crianca\n\n");
31            break;
32        case 11:
33        case 12:
34        case 13:
35        case 14:
36            printf("pre-adolescente\n\n");
37            break;
38        case 15:
39        case 16:
40        case 17:
41        case 18:
```

```
42     printf("adolescente\n\n");
43     break;
44     case 19:
45     ... // todos os valores entre 20 e 119!
46     case 120:
47     printf("adulto\n\n");
48     break;
49     default:
50     printf("Idade invalida ou sem classificacao definida\n\n");
51     }
52
53     system("PAUSE");
54     return 0;
55 }
```

No Programa 12, deve-se observar a forma de se representar faixas de valores (por exemplo, 1..10) em uma cláusula case do comando switch. Cada diferente valor da faixa gera uma cláusula case sem nenhum comando (i.e., case valor: ). Apenas o último valor da faixa possui a sequência de comandos e o comando break. Quando omite-se o comando break em uma cláusula case, a execução prossegue para a próxima cláusula case. Por isso, na linha 20 (case 1) a execução prossegue para a linha 21 (case 2) que prossegue para a linha 22 (case 3) e assim sucessivamente até a linha 29 (case 10) que possui um comando printf( ), seguido de um comando break para interromper a execução do comando switch. Para faixas de valores muito amplas, pode-se considerar o uso de comandos if-else aninhados para diminuir o número de linhas de código e também facilitar a escrita dos intervalos usando operadores relacionais e lógicos.





## Unidade 3

---

### Comandos de Repetição

---



### 3.1 Comandos de Repetição

A linguagem C oferece estruturas para o **controle do fluxo de execução** de um programa. Assim, pode-se alterar a simples execução seqüencial dos comandos. Dentre as estruturas de controle de fluxo, o Capítulo 2 apresentou os comandos condicionais (ou de seleção), os quais permitem que partes de um programa sejam executadas (ou não executadas) de acordo com o resultado de uma expressão lógica. Exemplos de comandos condicionais são if-else e switch. Além de comandos condicionais, programas C usam freqüentemente estruturas que permitem repetir, uma ou mais vezes, partes de um programa. Estas estruturas são conhecidas como comandos de repetição.

Um **comando de repetição**, portanto, permite a execução de um conjunto de comandos diversas vezes. A parte do programa que é executada várias vezes em um comando de repetição é chamada de **laço de repetição**, simplesmente laço ou ainda loop. Os comandos presentes dentro do laço são denominados **comandos internos** ao comando de repetição. Um comando de repetição, mesmo quando possui diversos comandos internos, é considerado um único comando pela linguagem C.

Por um lado, **o uso de comandos de repetição facilita a escrita de código**, desde que um mesmo trecho não precise ser reescrito diversas vezes (e com pequenas modificações). Como exemplo, nós já codificamos a leitura dos dados de 3 funcionários sem usar comandos de repetição (Programa 2 do Capítulo 2). Como visto neste programa, as partes de leitura dos dados de cada funcionário são muito parecidas entre si, diferindo apenas na escolha das variáveis. O uso de um comando de repetição e de arranjos (conceito que será explicado no próximo capítulo) permite a escrita de um único trecho para realizar a leitura dos dados de 3 funcionários. Por outro lado, **sem o uso de comandos de repetição alguns problemas não podem ser resolvidos**. Por exemplo, a leitura dos dados de N funcionários, onde N é conhecido apenas em tempo de execução (i.e., quando o programa é executado e o usuário informa o valor de N), não pode ser realizada sem o uso de

comandos de repetição. Isto é uma grande motivação para entendermos o funcionamento dos comandos de repetição.

De forma geral, todo comando de repetição pode especificar:

- o seu **estado inicial** (ou estado anterior). Isto pode ser feito inicializando-se o valor de uma variável que controlará o número de repetições do laço. Esta variável é chamada de **variável de controle de laço**;
- o seu **critério de parada**. Isto é realizado por uma expressão lógica. Como exemplo, a expressão (contador == 5) ou a expressão (tecla == 'F' || tecla == 'f') podem ser o critério de parada de um comando de repetição quando a expressão for verdadeira. Assim, quando a expressão for verdadeira, o comando de repetição é interrompido e executa-se o próximo comando após ele. Alguns comandos de repetição usam, ao invés de um critério de parada, o seu critério oposto, ou seja, um **critério de continuação**. Por exemplo, a expressão (contador != 5) ou a expressão (tecla != 'F' && tecla != 'f') podem ser o critério de continuação de um comando de repetição quando a expressão for verdadeira. Assim, quando a expressão for verdadeira, o laço de repetição é executado novamente; e
- o seu **estado final** (ou posterior). Isto geralmente corresponde ao valor da variável de controle de laço.

Neste capítulo, você também aprenderá sobre os seguintes **conceitos** relacionados aos comandos de repetição da linguagem C:

comando while, comando do-while, comando for, comandos break e continue em um laço de repetição, laço infinito, contadores de tempo e comando de repetição sem comandos internos.

### 3.1.1 Comando while

O **comando while** é um comando de repetição que primeiramente testa o seu critério de continuação e somente depois, dependendo do resultado, prossegue na execução do laço de repetição. Este comando é freqüentemente utilizado quando não se conhece a priori (i.e., na escrita do programa) quantas vezes o laço de repetição será executado. Além disso, o comando while deve ser usado apenas quando a execução do laço de repetição for opcional. Uma vez que o critério de continuação do comando while é testado antes da execução do laço de repetição, se o critério de continuação já for falso no primeiro teste, o laço de repetição não será executado nenhuma vez.

O **formato** do comando **while** é ilustrado na Figura 1, enquanto a Figura 3 ilustra o **fluxo de execução** deste comando. O seu funcionamento é o seguinte. Enquanto a expressão lógica for verdadeira, o laço de repetição é executado. Se a expressão lógica for falsa, o comando while é interrompido e o comando seguinte ao while é executado. Para garantir que o comando while termine, em geral, o valor da variável de controle de laço se altera dentro do laço de repetição. Note que o laço de repetição é delimitado por um par de chaves (i.e., { e } que indicam um bloco de comandos). As chaves podem ser omitidas quando o laço de repetição possui um único comando (Figura 2). Note também que a expressão lógica que define o critério de continuação do comando while deve ser colocada sempre entre parênteses. Por fim, mas não menos importante, deve-se respeitar o formato sugerido para o comando while, visando a legibilidade do código. Portanto, use indentação nos comandos internos do while.

```
while (expressão lógica)
{
    // comandos internos
}
// comando seguinte ao while
```

Figura 1. Formato do comando while.

```

while (expressão lógica)
    comando;
// comando seguinte ao while
  
```

Figura 2. Formato do comando while com único comando.

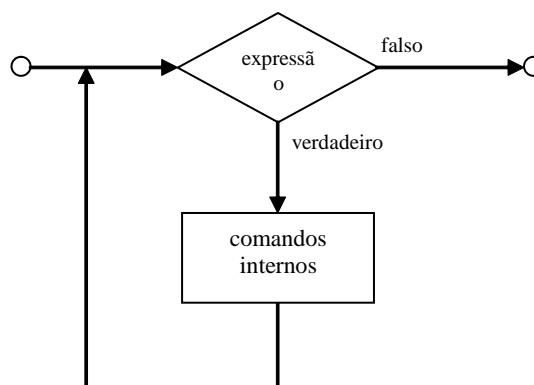


Figura 3. Fluxo de execução do comando while.

O Programa 1 a seguir exemplifica o uso do comando de repetição while. Este programa realiza a leitura dos dados de N funcionários (i.e., dados sobre o código, a idade e o salário de cada funcionário) e calcula a idade média e o salário médio dos N funcionários. O valor de N não é conhecido a priori, sendo determinado em função do número de leituras de dados. A repetição da leitura de dados pode continuar se o usuário digitar 'S' ou 's'. No final, o programa mostra apenas os dados calculados (i.e., o total de funcionários, a idade média e o salário médio) na saída padrão.

### Programa 1

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[])
5  {
6      int codFunc; // código do funcionário corrente sendo manipulado
7      int idadeFunc; // idade do funcionário corrente
8      float salFunc; // salário do funcionário corrente
9
  
```

```
10  int totalFunc;    // total de funcionários (valor de N)
11  int somaIdade = 0; // soma das idades dos funcionários
12  float somaSalario = 0; // soma dos salários dos funcionários
13
14  float idadeMedia; // idade média dos funcionários
15  float salarioMedio; // salário médio dos funcionários
16
17  char tecla;       // opção para leitura de novos dados
18
19  /* entrada de dados dos funcionários e
20   cálculos intermediários */
21
22  printf("Deseja inserir os dados de um novo funcionario?\n");
23  printf("[tecle S ou s para continuar]: ");
24  scanf("%c", &tecla);
25  printf("\n");
26
27  totalFunc = 0;
28  while (tecla == 'S' || tecla == 's')
29  {
30      // leitura dos dados do funcionário corrente
31      printf("A seguir, entre com todos os dados do funcionario.\n\n");
32      printf("Digite o codigo: ");
33      scanf("%d", &codFunc);
34      printf("Digite a idade: ");
35      scanf("%d", &idadeFunc);
36      printf("Digite o salario (R$): ");
37      scanf("%f", &salFunc);
38      printf("\n\n");
39      fflush(stdin);
40
41      // cálculos intermediários
42      totalFunc++;
43      somaIdade += idadeFunc;
44      somaSalario += salFunc;
45
46      // mudança da variável de controle de laço
47      printf("Deseja inserir os dados de um novo funcionario?\n");
48      printf("[tecle S ou s para continuar]: ");
49      scanf("%c", &tecla);
50      printf("\n");
51  }
52
53  // cálculo da idade média e do salário médio dos N funcionários
54
```

```
55     if (totalFunc > 0)
56     {
57         idadeMedia = (float) somaIdade / totalFunc;
58         salarioMedio = somaSalario / totalFunc;
59     }
60     else
61     {
62         idadeMedia = 0;
63         salarioMedio = 0;
64     }
65
66     // saída dos dados calculados
67
68     printf("Os dados calculados sao:\n");
69     printf("Total de funcionarios = %d\n", totalFunc);
70     printf("Idade media %.2f anos\n", idadeMedia);
71     printf("Salario medio %.2f reais\n", salarioMedio);
72     printf("\n");
73
74     // finalização do programa principal
75
76     system("PAUSE");
77     return 0;
78 }
```

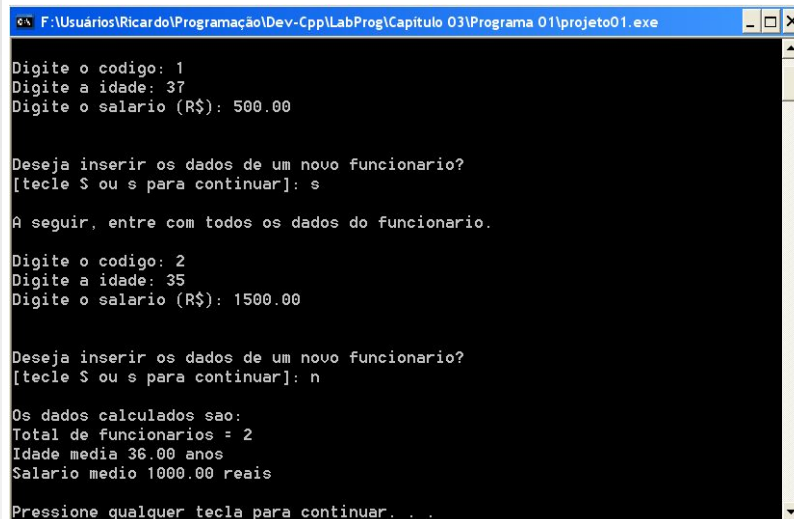
No Programa 1, um mesmo conjunto de variáveis é usado para ler os dados de todos os funcionários (linhas 6 a 8). Isto é possível porque os dados de um funcionário são lidos, logo em seguida são usados e depois não são mais necessários no programa. Na linha 10, a variável `totalFunc` é declarada para contabilizar o total de funcionários, ou seja, o valor de `N` citado no enunciado do Programa 1. A variável `tecla` declarada na linha 17 é uma variável de controle de laço. Esta variável irá armazenar a tecla digitada pelo usuário, a qual poderá indicar o término ou a continuação da leitura de dados. Nas linhas 22 a 25, o programa obtém a opção do usuário e armazena a opção na variável `tecla`. A linha 27 inicializa o valor da variável `totalFunc`, indicando que os dados de nenhum funcionário foram lidos.

O comando de repetição `while`, objeto de estudo do Programa 1, ocupa as linhas 28 a 51. A linha 28 possui o critério de continuação do comando `while`, ou seja, o comando executará o laço de repetição



se o usuário digitar 'S' ou 's'. O laço de repetição localiza-se nas linhas 29 a 51. Cada execução do laço realiza o seguinte. Primeiramente, os dados de um novo funcionário são lidos (linhas 30 a 39). Observe a utilização da função `fflush(stdin)` nesta parte do laço de repetição. O comando na linha 39 esvazia o buffer de entrada de dados do teclado (i.e., argumento `stdin`). Isto é necessário para eliminar a tecla <enter> digitada na leitura do salário do funcionário na linha 37, o qual pode interferir no próximo comando de leitura de dados (experimente remover esta linha do programa para verificar o que acontece!). Já nas linhas 41 a 44 do laço de repetição são realizados processamentos necessários para o cálculo do total de funcionários, da soma das idades e da soma dos salários. Por fim, as linhas 46 a 50 do laço de repetição obtém a nova opção do usuário (i.e., se o usuário deseja continuar ou não com a leitura de dados) e armazena a nova opção na variável `tecla`. Note que é necessária uma nova leitura de uma tecla para possivelmente alterar o critério de continuação do comando `while`.

O cálculo da idade média e do salário médio dos funcionários é realizado nas linhas 53 a 64. Este cálculo é similar ao efetuado anteriormente no Programa 2 do Capítulo 2. A principal diferença é o uso de um comando condicional `if-else` para testar a condição de nenhuma leitura de dados. Caso isto aconteça, o bloco de comandos da parte `else` é executado, atribuindo zero tanto à idade média dos funcionários quanto ao salário médio dos funcionários. Isto evita uma divisão por zero que pode ocasionar erro na execução do programa (linhas 57 e 58). O restante do programa já foi explicado anteriormente no Programa 2 do Capítulo 2. A Figura 4 ilustra um exemplo de execução do Programa 1.



```
F:\Usuários\Ricardo\Programação\Dev-Cpp\LabProg\Capítulo 03\Programa 01\projeto01.exe

Digite o código: 1
Digite a idade: 37
Digite o salário (R$): 500.00

Deseja inserir os dados de um novo funcionario?
[tecle S ou s para continuar]: s

A seguir, entre com todos os dados do funcionario.

Digite o código: 2
Digite a idade: 35
Digite o salário (R$): 1500.00

Deseja inserir os dados de um novo funcionario?
[tecle S ou s para continuar]: n

Os dados calculados são:
Total de funcionarios = 2
Idade média 36.00 anos
Salário médio 1000.00 reais

Pressione qualquer tecla para continuar. . .
```

Figura 4. Execução do Programa 1.

### 3.1.2 Comando do-while

O comando **do-while** é um comando de repetição que primeiramente executa o laço de repetição e em seguida testa o seu critério de continuação para determinar se executa novamente o laço de repetição. Este comando é frequentemente utilizado quando não se conhece a priori quantas vezes o laço de repetição será executado, mas sabe-se que o laço será executado pelo menos 1 vez. Portanto, o comando do-while deve ser usado apenas quando a execução do laço de repetição não for opcional.

O **formato** do comando **do-while** é ilustrado nas Figuras Figura 5 e Figura 6. O laço de repetição do comando do-while sempre é delimitado por chaves, independentemente do número de comandos internos (ou seja, do laço ter um único comando ou ter vários comandos). Ademais, o comando do-while sempre é finalizado com ponto-e-vírgula. Essa é a sua sintaxe (lembre-se desta diferença para o comando while). Note também que a expressão

lógica que define o critério de continuação do comando do-while deve ser colocada sempre entre parênteses (da mesma forma que no comando while). Sugere-se respeitar o formato para o comando do-while apresentado nas Figuras Figura 5 e Figura 6 de forma a garantir a legibilidade do código.

A Figura 7 mostra o **fluxo de execução** do comando do-while. Conforme já explicado, em primeiro lugar os comandos internos são executados. Em seguida, a expressão lógica, que representa o critério de continuação, é testada. Se o valor da expressão for verdadeiro, repete-se a execução dos comandos internos e novamente testa-se a expressão lógica. Se o valor da expressão for falso, o comando do-while é interrompido e o comando após ao do-while é executado. Para garantir que o comando do-while termine, em geral, o valor da variável de controle de laço se altera dentro do laço de repetição.

```
do
{
    // comandos internos
}
while (expressão lógica);
// comando seguinte ao do-while
```

Figura 5. Formato do comando do-while.

```
do
{
    comando;
}
while (expressão lógica);
// comando seguinte ao do-while
```

Figura 6. Formato do comando do-while com único comando.

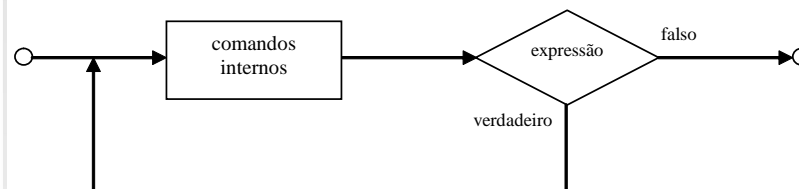


Figura 7. Fluxo de execução do comando do-while.

O Programa 2 a seguir exemplifica o uso do comando de repetição do-while. Este programa realiza a leitura dos dados de 1 ou mais funcionários (i.e., dados sobre o código, a idade e o salário de cada funcionário) e calcula a idade média e o salário médio dos funcionários. O total de funcionários não é conhecido a priori, sendo que serão lidos os dados de pelo menos 1 funcionário. A repetição da leitura de dados pode continuar se o usuário digitar 'S' ou 's'. No final, o programa mostra apenas os dados calculados (i.e., o total de funcionários, a idade média e o salário médio) na saída padrão.

### Programa 2

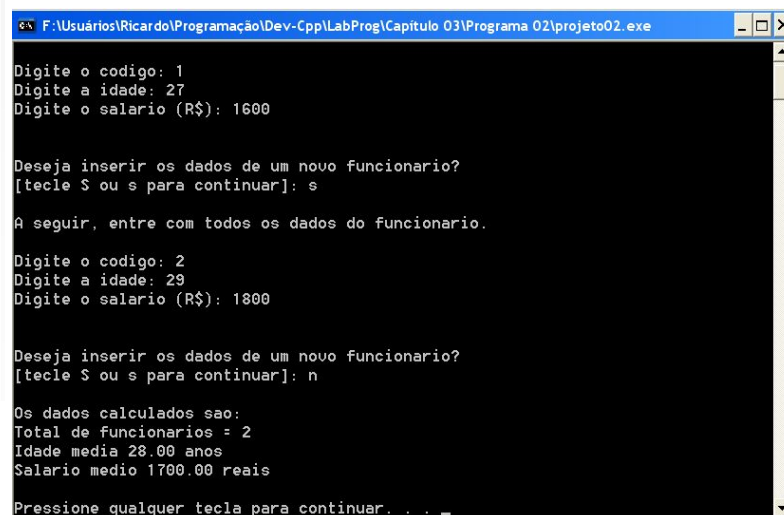
```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[])
5  {
6      int codFunc; // código do funcionário corrente sendo manipulado
7      int idadeFunc; // idade do funcionário corrente
8      float salFunc; // salário do funcionário corrente
9
10     int totalFunc; // total de funcionários (valor de N)
11     int somaIdade = 0; // soma das idades dos funcionários
12     float somaSalario = 0; // soma dos salários dos funcionários
13
14     float idadeMedia; // idade média dos funcionários
15     float salarioMedio; // salário médio dos funcionários
16
17     char tecla; // opção para leitura de novos dados
18
19     /* entrada de dados dos funcionários e
20        cálculos intermediários */
  
```

```
21
22     totalFunc = 0;
23     do
24     {
25         // leitura dos dados do funcionário corrente
26         printf("A seguir, entre com todos os dados do funcionario.\n\n");
27         printf("Digite o codigo: ");
28         scanf("%d", &codFunc);
29         printf("Digite a idade: ");
30         scanf("%d", &idadeFunc);
31         printf("Digite o salario (R$): ");
32         scanf("%f", &salFunc);
33         printf("\n\n");
34         fflush(stdin);
35
36         // cálculos intermediários
37         totalFunc++;
38         somaIdade += idadeFunc;
39         somaSalario += salFunc;
40
41         // mudança da variável de controle de laço
42         printf("Deseja inserir os dados de um novo funcionario?\n");
43         printf("[tecle S ou s para continuar]: ");
44         scanf("%c", &tecla);
45         printf("\n");
46     }
47     while (tecla == 'S' || tecla == 's');
48
49     // cálculo da idade média e do salário médio dos funcionários
50
51     if (totalFunc > 0)
52     {
53         idadeMedia = (float) somaIdade / totalFunc;
54         salarioMedio = somaSalario / totalFunc;
55     }
56     else
57     {
58         idadeMedia = 0;
59         salarioMedio = 0;
60     }
61
62     // saída dos dados calculados
63
64     printf("Os dados calculados sao:\n");
```

```
65     printf("Total de funcionarios = %d\n", totalFunc);
66     printf("Idade media %.2f anos\n", idadeMedia);
67     printf("Salario medio %.2f reais\n", salarioMedio);
68     printf("\n");
69
70     // finalização do programa principal
71
72     system("PAUSE");
73     return 0;
74 }
```

O Programa 2 possui uma lógica muito similar à lógica do Programa 1. A principal diferença consiste no uso do comando de repetição do-while (linhas 23 a 47). O uso deste comando tem duas implicações. A primeira implicação é que para pelo menos 1 funcionário os seus dados serão lidos. A segunda implicação é que a escrita do bloco de comandos que realiza a leitura da tecla (linhas 41 a 45) é feita somente dentro do laço, de forma distinta do que foi codificado no Programa 1 (linhas 22 a 25 e 46 a 50). Ou seja, o comando do-while para este exemplo facilitou a escrita do código (i.e., não houve necessidade de replicação de partes do código). Todos os demais detalhes do funcionamento do Programa 2 já foram explicados no Programa 1.



```

F:\Usuários\Ricardo\Programação\Dev-Cpp\LabProg\Capitulo 03\Programa 02\projeto02.exe
Digite o codigo: 1
Digite a idade: 27
Digite o salario (R$): 1600

Deseja inserir os dados de um novo funcionario?
[tecle $ ou s para continuar]: s

A seguir, entre com todos os dados do funcionario.

Digite o codigo: 2
Digite a idade: 29
Digite o salario (R$): 1800

Deseja inserir os dados de um novo funcionario?
[tecle $ ou s para continuar]: n

Os dados calculados sao:
Total de funcionarios = 2
Idade media 28.00 anos
Salario medio 1700.00 reais

Pressione qualquer tecla para continuar. . . .
```

Figura 8. Execução do Programa 2.

### 3.1.3 Comando for

O **comando for** é um comando de repetição usado especificamente quando já se sabe a priori quantas vezes o laço de repetição será executado. A quantidade de vezes que o laço será executado pode ser determinado por uma constante inteira (tal como 10, 15 ou 20 vezes) ou por uma variável inteira (por exemplo, int contador), cujo valor é conhecido antes do comando for. Portanto, a semântica do comando for é “faça o conjunto de comandos internos N vezes”, onde o valor de N é previamente conhecido. Em outras palavras, o comando for é usado quando se conhece o valor inicial e final da variável de controle de laço.

O **formato** do comando **for** é ilustrado na Figura 9. As três partes que formam o cabeçalho do comando for (isto é, inicialização, critério de continuação e incremento) são expressões. A inicialização é tipicamente um comando de atribuição. Nesta primeira parte, a variável de controle de laço é inicializada (e.g.,  $i = 0$  ou  $j = 1$ ). O critério de continuação é uma expressão lógica (e.g.,  $i < 10$  ou  $j < 5$  || !achou). Já a terceira parte, incremento, realiza a modificação automática no valor da variável de controle de laço. Em geral, esta parte envolve o incremento ou decremento da variável de controle de laço (e.g.,  $i++$  ou  $j--$ ). O comando for sempre coloca as suas três partes entre parênteses. Estas partes são separadas por ponto-e-vírgula dentro dos parênteses. Note que o laço de repetição é delimitado por um par de chaves (i.e.,  $\{ e \}$  que indicam um bloco de comandos). As chaves podem ser omitidas quando o laço de repetição possui um único comando (Figura 10). Deve-se respeitar o formato sugerido para o comando for, visando a legibilidade do código. Portanto, use indentação nos comandos internos do for.

Apesar das três partes que formam o cabeçalho do comando for serem comumente empregadas, nenhuma delas é obrigatória. Por exemplo, a variável de controle de laço já pode ter sido inicializada anteriormente no programa e, portanto a parte de inicialização é deixada vazia. Outro exemplo consiste em não modificar o valor da variável de controle de laço na parte de incremento, desde que

esta variável já pode ter o seu valor modificado dentro dos comandos internos do comando for.

```
for (inicialização; critério de continuação; incremento)
{
    // comandos internos
}
// comando seguinte ao for
```

Figura 9. Formato do comando for.

```
for (inicialização; critério de continuação; incremento)
    comando;
// comando seguinte ao for
```

Figura 10. Formato do comando for com único comando.

A Figura 11 ilustra o **fluxo de execução** do comando for. O funcionamento deste comando é o seguinte. Primeiramente, a parte de inicialização é executada. Esta parte é executada somente desta vez. Após, o critério de continuação é testado. Se o valor da expressão que representa este critério for verdadeiro, o laço de repetição é executado e em seguida também é executada a parte de incremento, retornando ao teste do critério de continuação. Se o valor da expressão que representa o critério de continuação for falso, o comando for é interrompido e o comando seguinte ao for é executado. De forma mais simples, considere  $i$  a variável de controle de laço inicializada com 0 (i.e.,  $i = 0$ ), o critério de continuação  $i < 3$  e a parte de incremento sendo  $i++$ . Neste exemplo, o comando for executará o laço de repetição 3 vezes (i.e., para os valores 0, 1 e 2 da variável  $i$ ).



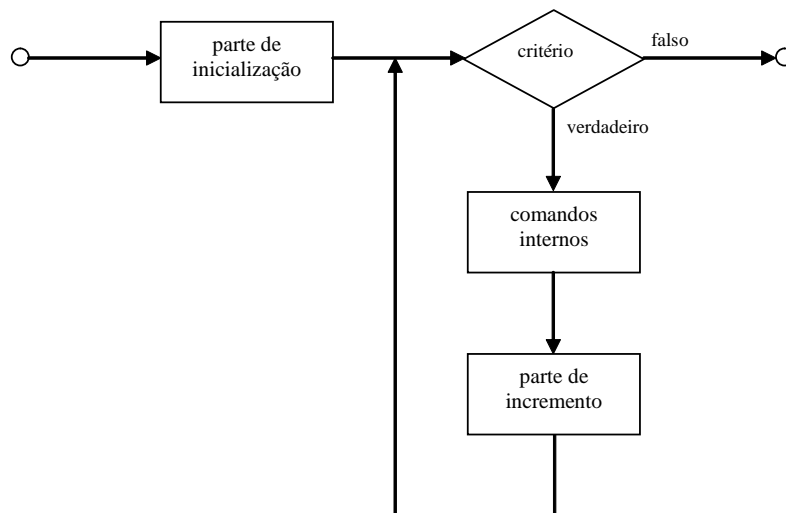


Figura 11. Fluxo de execução do comando for.

A seguir, a Figura 12 ilustra a equivalência entre um comando for e um comando while. Assim, espera-se um melhor entendimento do funcionamento do comando for. Note que apesar de equivalentes, o comando for é muito mais compacto, reduzindo a escrita de código.

```

for (inicialização; critério; incremento){ // comandos internos} // comando seguinte ao for
inicialização
while (critério){ // comando internos
incremento} // comando seguinte ao for
  
```

Figura 12. Equivalência entre os comandos for e while.

O Programa 3 a seguir exemplifica o uso do comando de repetição for. Este programa realiza a leitura dos dados de 5 funcionários (i.e., dados sobre o código, a idade e o salário de cada funcionário) e calcula a idade média e o salário médio dos funcionários. O total de funcionários, portanto, é conhecido a priori. No final, o programa mostra apenas os dados calculados (i.e., a idade média e o salário médio) e o total de funcionários na saída padrão.

### Programa 3

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[])
5  {
6      int codFunc; // código do funcionário corrente sendo manipulado
7      int idadeFunc; // idade do funcionário corrente
8      float salFunc; // salário do funcionário corrente
9
10     const int totalFunc = 5; // total de funcionários
11     int somaIdade = 0; // soma das idades dos funcionários
12     float somaSalario = 0; // soma dos salários dos funcionários
13
14     float idadeMedia; // idade média dos funcionários
15     float salarioMedio; // salário médio dos funcionários
16
17     int i; // contador temporário do comando for
18
19     /* entrada de dados dos funcionários e
20        cálculos intermediários */
21
22     for (i = 0; i < 5; i++)
23     {
24         // leitura dos dados do funcionário corrente
25         printf("A seguir, entre com todos os dados do funcionario.\n\n");
26         printf("Digite o codigo: ");
27         scanf("%d", &codFunc);
28         printf("Digite a idade: ");
29         scanf("%d", &idadeFunc);
30         printf("Digite o salario (R$): ");
31         scanf("%f", &salFunc);
32         printf("\n\n");
33         fflush(stdin);
34
35         // cálculos intermediários
36         somaIdade += idadeFunc;
37         somaSalario += salFunc;
38     }
39
40     // cálculo da idade média e do salário médio dos funcionários
41
```

```
42     idadeMedia = (float) somaIdade / totalFunc;  
43     salarioMedio = somaSalario / totalFunc;  
44  
45     // saída dos dados calculados  
46  
47     printf("Os dados calculados sao:\n");  
48     printf("Total de funcionarios = %d\n", totalFunc);  
49     printf("Idade media %.2f anos\n", idadeMedia);  
50     printf("Salario medio %.2f reais\n", salarioMedio);  
51     printf("\n");  
52  
53     // finalização do programa principal  
54  
55     system("PAUSE");  
56     return 0;  
57 }
```

O Programa 3 declara uma constante para armazenar o total de funcionários, desde que este total é conhecido a priori e não se altera no programa (linha 10). Além desta declaração, de novo há somente a declaração da variável de controle de laço *i* (linha 17). O comando de repetição *for* (linhas 22 a 38) realiza a leitura dos dados de 5 funcionários. Note que o uso do comando *for* reduziu o número de linhas necessário para codificar as repetidas leituras de dados e subseqüentes cálculos. Ou seja, o laço de repetição do comando *for* (linhas 23 a 38) é mais enxuto que os laços codificados nos programas anteriores usando os comandos *while* e *do-while*. Isto também se deve, em parte, ao fato que não é mais necessário pedir ao usuário para informar sobre a continuação da leitura dos dados (i.e., variável *tecla*). Pelo fato de se conhecer o total de funcionários, diferente de zero, não é mais necessário um comando *if-else* no cálculo da idade média e do salário médio dos funcionários. O restante do programa é idêntico aos programas anteriores. A Figura 13 ilustra um exemplo de execução do Programa 3.

```
F:\Usuários\Ricardo\Programação\Dev-Cpp\LabProg\Capitulo 03\Programa 03\projeto03.exe
Digite o código: 3
Digite a idade: 25
Digite o salário (R$): 1456.34

A seguir, entre com todos os dados do funcionario.

Digite o código: 4
Digite a idade: 37
Digite o salário (R$): 5678.34

A seguir, entre com todos os dados do funcionario.

Digite o código: 5
Digite a idade: 67
Digite o salário (R$): 8000.00

Os dados calculados são:
Total de funcionarios = 5
Idade media 45.20 anos
Salário medio 4314.85 reais

Pressione qualquer tecla para continuar. . .
```

Figura 13. Execução do Programa 3.

## 3.2 Explicações Adicionais

Esta seção reforça a explicação de alguns dos conceitos estudados. Para tanto, serão usados exemplos de trechos de programas.

### Lazo Infinito

O uso de um comando de repetição pode ocasionar a ocorrência de um **laço infinito**. Ou seja, o laço de repetição executa indefinidamente (i.e., nunca pára a sua execução) caso o critério de parada do comando de repetição nunca seja alcançado (ou verdadeiro) ou, inversamente, caso o critério de continuação do comando de repetição seja sempre alcançado. Isto tipicamente acontece quando o valor da variável de controle de laço não se altera dentro do laço ou, quando se altera o valor da variável de controle de laço, o seu valor não converge para interromper o laço. Outra situação típica é o uso de uma constante, por exemplo 1 (verdadeiro) para indicar que o laço de repetição sempre será executado.

A ocorrência de um laço infinito, em geral, indica um erro de lógica no programa, conforme exemplificado no primeiro trecho de código a seguir. Neste trecho, o critério de continuação do comando while sempre será verdadeiro, desde que a variável *i* é inicializada com o valor 1 e depois dentro do laço de repetição a variável *i* somente é incrementada. Nota-se, portanto, a definição incorreta do critério de continuação, que poderia ser, ao invés de (*i* > 10), algo como (*i* < 10). O segundo trecho de código, por sua vez, mostra o uso de uma constante para provocar um laço infinito nos comandos while, do-while e for. O seu uso é indicado em programas que realizam o monitoramento permanente de dados de dispositivos de entrada e saída. Assim, o programa, comumente chamado de daemon, executa ininterruptamente para verificar alguma condição no dispositivo. Portanto, deve-se ter atenção redobrada no uso de comandos de repetição.

```
...  
i = 1  
while (i > 0)  
{  
    printf("Digite a idade:");  
    scanf("%d", &idade);  
    i++;  
}  
...
```

```
...  
while (1)  
{  
    // comandos internos  
}  
...
```

```
...  
do  
{  
    // comandos internos  
}  
while (1);  
...  
for (;;)   
{  
    // comandos internos  
}  
...
```

### Comando de repetição sem comandos internos

O trecho de código a seguir ilustra um comando while que não possui nenhum comando interno. Neste exemplo, o ponto-e-vírgula depois do critério de continuação do comando while representa um comando vazio (i.e., um comando que não faz nada). De fato, toda a lógica deste comando está localizada no critério de continuação do comando while. Primeiramente, a variável `tecla` recebe o valor retornado pela função `getchar( )`. Esta função é usada para ler um novo caractere do teclado. Após, o valor da variável `tecla` é comparado com o caractere 'S'. Se o resultado for verdadeiro, nada é feito (i.e., comando vazio do laço), porém o critério de continuação é novamente testado e, portanto, a variável `tecla` recebe um novo caractere, o qual em seguida é comparado com o caractere 'S'. O comando while termina quando o usuário digitar o caractere 'S'. Comandos deste tipo podem ser usados para validar a entrada de dados, forçando que os dados estejam contidos em um subconjunto de caracteres. Note que um critério de continuação (ou de parada) de um comando de repetição pode usar o comando de atribuição dentro de uma expressão. Isto reduz a quantidade de código. O trecho de código com o segundo comando while é um comando equivalente ao primeiro comando while, mas que necessita de 3 linhas de código.

```
...  
while ( (tecla = getchar( )) != 'S' );  
...  
...  
  
tecla = getchar( );  
while (tecla != 'S')  
    tecla = getchar( );  
...
```

### Comando for

O comando for pode ser usado para implementar um **contador de tempo** (ou delay). Para isto, usa-se o comando for sem nenhum comando interno. Em geral, uma função delay(tempo) é disponibilizada, a qual é implementada com um comando for que varia de um valor inicial até um valor final calculado com base no argumento tempo (exemplo tempo\*100). O trecho a seguir ilustra este uso do comando for.

```
...  
for (i=0; i<tempo*100; i++);  
...
```

O comando for também pode efetuar a **chamada a funções** nas suas três partes. Por exemplo, no trecho a seguir o comando for possui na sua parte de inicialização a chamada a função f1( ), possui na parte de critério de continuação a chamada a função f2( ) e possui na sua parte de incremento a chamada a função f3( ). Em particular, uma variável cont recebe o valor da função f2( ) e se o valor for diferente de zero (verdadeiro) o comando for continua a sua execução. Este não é um uso comum de um comando for, mas demonstra a flexibilidade e poder deste comando. O conceito de funções é descrito em detalhes no Capítulo 4.

```
...  
for (cont = f1( ); cont = f2( ); cont = f3( ));  
...
```

O comando `for` pode trabalhar com **mais de uma variável de controle de laço**. Isto é muito útil, por exemplo, quando percorre-se strings em sentidos opostos ao mesmo tempo, ou seja, do começo para o fim da string e do fim para o começo da string. O trecho de código a seguir ilustra este uso do comando `for`. A função `strlen( )` retorna o tamanho de uma string. Como uma string ocupa as posições 0 até tamanho-1, a variável de controle de laço `fim` recebe `strlen(nome)-1`, onde `nome` é uma string. Note que para usar duas variáveis de controle de laço, tanto na parte de inicialização quanto na parte de incremento, foi usado o operador vírgula (i.e., com semântica “faz isto e isto”). A critério de continuação neste exemplo é “não terminou”. Assim, enquanto terminou `for` igual a zero (falso), o comando `for` continua.

```
...  
for (inicio=0, fim=strlen(nome)-1; !terminou; inicio++, fim--)  
{  
    ...  
    if (inicio > fim)  
        terminou = 1;  
    ...  
}  
...
```

### Comando `break` em um laço de repetição

O comando `for` ilustrado no trecho de código a seguir é idealizado para executar 100 vezes o laço de repetição. No entanto, se o comando `if (v==12)` for verdadeiro, o comando `break` é executado e por conseguinte o comando `for` é interrompido (i.e., desvia-se o fluxo de execução para o próximo comando após o `for`). A interrupção do comando `for` pelo comando `break` ocorre independentemente do número de vezes que o laço de repetição foi executado. De fato, o comando `break` é usado para representar um critério adicional de parada do comando `for`. Isto é muito útil, por exemplo, quando se pesquisa uma estrutura de `N` elementos, do primeiro ao último elemento, sendo que a pesquisa deve parar assim que uma condição `for` satisfeita, porém caso esta condição não seja



satisfeita, deve-se garantir que todos os elementos foram analisados. O comando `break` pode ser usado de forma similar conjuntamente aos comandos de repetição `while` e `do-while`. Assim, uma vez executado, o comando de repetição é interrompido.

```
...  
for (i=0; i<100; i++)  
{  
    // conjunto de comandos 1  
    ...  
    if (v == 12)  
        break;  
    ...  
    // conjunto de comandos 2  
}  
...
```

#### **Comando `continue` em um laço de repetição**

O comando `for` ilustrado no trecho de código a seguir é idealizado para executar 100 vezes todos os comandos do laço de repetição, ou seja, o conjunto de comandos 1, o teste condicional e o conjunto de comandos 2. No entanto, se o comando `if (v==12)` for verdadeiro, o comando `continue` é executado e por conseguinte o fluxo de execução é desviado para a parte de incremento do comando `for`. Assim, o conjunto de comandos 2 não é executado. Portanto, o comando `continue` força a ocorrência da próxima repetição (ou iteração) do laço a partir de um ponto deste laço.

O comando `continue` também pode ser utilizado de forma similar com os comandos `while` e `do-while`. No entanto, para estes comandos de repetição o comando `continue` desvia o fluxo de execução para o critério de continuação (do comando `while` ou `do-while`).

```
...  
for (i=0; i<100; i++)  
{  
    // conjunto de comandos 1  
    ...  
    if (v == 12)  
        continue;  
    ...  
    // conjunto de comandos 2  
}  
...
```

### 3.3 Mapeamento de Algoritmo para Programa C

A seguir são apresentados os mapeamentos de 4 algoritmos para programas C, com ênfase nos comandos de repetição while, do-while e for.

#### Algoritmo 6-2 (apostila de “Construção de Algoritmos”)

```
1      { algoritmo que apresenta uma saudação para alguns nomes }  
2  
3      algoritmo  
4          declare nome: literal  
5  
6          { primeiro nome a ser lido }  
7          leia(nome)  
8  
9          { repetição que termina quando o nome for “fim” }  
10         enquanto nome ≠ “fim” faça  
11             { escreva a saudação }  
12             escreva(“Olá,” nome, “, como vai?”)  
13  
14             { leitura do próximo nome }  
15             leia(nome)  
16         fim-enquanto  
17  
18         { despedida }  
19         escreva(“Não há mais ninguém? Então, tchau!”)  
20     fim-algoritmo
```

**Programa 4 (equivalente ao algoritmo 6-2)**

```
1 // algoritmo que apresenta uma saudação para alguns nomes
2
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 int main(int argc, char *argv[])
7 {
8     char nome[255];
9
10    // primeiro nome a ser lido
11    printf("Digite o nome: ");
12    scanf("%s", &nome);
13
14    // repetição que termina quando o nome for igual a "fim"
15    while (strcmp(nome,"fim")!=0)
16    {
17        // escreve a saudação
18        printf("Ola, %s, como vai?\n\n", nome);
19
20        // leitura do próximo nome
21        printf("Digite o nome: ");
22        scanf("%s", &nome);
23    }
24
25    // despedida
26    printf("Nao ha mais ninguem? Entao, tchau!\n\n");
27
28    // finalização do programa principal
29
30    system("PAUSE");
31    return 0;
32 }
```

Com relação ao Programa 4, deve-se notar a forma de mapeamento do critério de continuação do comando enquanto (i.e., nome “ fim”) no algoritmo no critério de continuação do comando while (i.e., strcmp(nome,”fim")!=0) no programa C. Vale lembrar que a linguagem C trabalha com strings como cadeias de caracteres que terminam em \0. Para manipular strings é necessário o uso de funções de biblioteca. A função strcmp( ) já foi explicada nesta apostila. Esta função retorna 0 se, no caso da linha

15, o valor da variável nome for igual a “fim”. Portanto, para o valor da variável nome ser diferente de “fim”, o valor retornado pela função strcmp( ) deve ser diferente de 0.

### Algoritmo 6-3 (apostila de “Construção de Algoritmos”)

```
1  { geração de uma tabela de conversão de graus Celsius para
Fahrenheit
2    de 0 a 10 (inclusive), de 0,5 em 0,5. }
3
4  algoritmo
5    declare celsius, fahrenheit: real
6
7    { geração da tabela }
8    celsius ← 0 { valor inicial }
9    faça
10     { calcula o valor convertido e escreve ambos }
11     fahrenheit ← celsius × 1,8 + 32
12     escreva(celsius, “<—>”, fahrenheit)
13
14     { passa para o próximo valor em Celsius }
15     celsius ← celsius + 0,5
16   enquanto celsius <= 10
17 fim-algoritmo
```

### Algoritmo 6-4 (apostila de “Construção de Algoritmos”)

```
1  { geração de uma tabela de conversão de graus Celsius para Fahrenheit
2    de 0 a 10 (inclusive), de 0,5 em 0,5. }
3
4  algoritmo
5    declare celsius, fahrenheit: real
6
7    { geração da tabela }
8    celsius ← 0 { valor inicial }
9    faça
10     { calcula o valor convertido e escreve ambos }
11     fahrenheit ← celsius × 1,8 + 32
12     escreva(celsius, “<—>”, fahrenheit)
13
14     { passa para o próximo valor em Celsius }
```

```

15      celsius ← celsius + 0,5
16      até celsius > 10
17      fim-algoritmo
  
```

### Programa 5 (equivalente aos algoritmo 6-3 e 6-4)

```

1  /* geração de uma tabela de conversão de graus Celsius para Fahrenheit
2     de 0 a 10 (inclusive), de 0,5 em 0,5. */
3
4  #include <stdio.h>
5  #include <stdlib.h>
6
7  int main(int argc, char *argv[])
8  {
9      float celsius, fahrenheit;
10
11     // geração da tabela
12     celsius = 0;
13
14     do
15     {
16         // calcula o valor convertido e escreve ambos
17         fahrenheit = celsius * 1.8 + 32;
18         printf("%f <—> %f\n", celsius, fahrenheit);
19
20         // passa para o próximo valor em Celsius
21         celsius += 0.5;
22     }
23     while (celsius <= 10);
24
25     system("PAUSE");
26     return 0;
27 }
  
```

Os Algoritmos 6-3 e 6-4 foram mapeados em uma mesmo programa C (Programa 5). Isto foi possível porque estes algoritmos são quase idênticos, diferindo apenas no uso dos comandos *faça-enquanto* e *faça-até*. Tais comandos possuem expressões lógicas complementares, ou seja, o comando *faça-enquanto* usa um critério de continuação e o comando *faça-até* usa um critério de parada. Por isto, no Algoritmo 6-3 a expressão lógica é (celsius d" 10), ao passo que no Algoritmo 6-4 a expressão lógica é (celsius > 10).

Ambos os comandos faça-enquanto e faça-até são mapeados no comando do-while da linguagem C, que possui um critério de continuação.

### Algoritmo 6-1 (apostila de “Construção de Algoritmos”)

```
1  { algoritmo para gerar uma tabela de conversão Celsius -> Fahrenheit
2    para valores de 0 a 40, de 1 em 1 }
3
4  algoritmo
5    declare celsius: inteiro  { sempre inteiro, neste caso }
6      Fahrenheit: real  { real, pois é obtido pela conversão }
7
8    { repetição para os valores indicados }
9    para celsius ← 0 até 40 faça
10      fahrenheit ← celsius * 1.8 + 32;
11      escreva(celsius, “ <—> “, fahrenheit)
12    fim-para
13  fim-algoritmo
```

### Programa 6 (equivalente ao algoritmo 6-1)

```
1  /* algoritmo para gerar uma tabela de conversão Celsius -> Fahrenheit
2    para valores de 0 a 40, de 1 em 1 */
3
4  #include <stdio.h>
5  #include <stdlib.h>
6
7  int main(int argc, char *argv[])
8  {
9    int celsius;    // sempre inteiro, neste caso
10   float fahrenheit; // real, pois é obtido pela conversão
11
12   // repetição para os valores indicados
13   for (celsius=0; celsius<41; celsius++)
14   {
15     fahrenheit = celsius * 1.8 + 32;
16     printf(“%d <—> %.2f\n”, celsius, fahrenheit);
17   }
18
19   system(“PAUSE”);
20   return 0;
21 }
```

## Unidade 4

---

### Ponteiros, Registros e Funções

---





Este capítulo irá tratar de conceitos mais avançados da linguagem C. Na seção 4.1 será apresentado o conceito de ponteiros. Na seção 4.2 serão apresentados os conceitos de registros, uniões e enumerações. Na seção 4.3 será descrito o uso do conceito de funções.

## 4.1 Ponteiros

Nesta seção, você aprenderá sobre os seguintes **conceitos** relacionados ao uso de ponteiros:

ponteiros, tipo de dados e variável ponteiro, esquema de endereçamento de memória, principais usos de ponteiros, declaração de ponteiros, operadores de ponteiros & e \*, uso da função `printf( )` com ponteiros, aritmética de ponteiros, ponteiros do tipo `void`, declaração de ponteiros com inicialização, atribuição e comparação de ponteiros, ponteiro `NULL` e alocação dinâmica de memória com uso das funções `malloc( )` e `free( )`.

O **tipo de dado ponteiro** permite o armazenamento de um endereço de memória. Assim, uma variável declarada como do tipo ponteiro sempre irá armazenar um valor que é um endereço de memória. Esta variável é chamada de **variável ponteiro** (ou simplesmente de **ponteiro**) e o seu valor é freqüentemente o endereço de uma outra variável. Portanto, em programação C diz-se que uma variável ponteiro aponta para outra variável.

A Figura 1 ilustra o uso da variável ponteiro chamada `pont_idade`. Esta variável armazena o valor 704, que corresponde ao endereço de memória da variável `idade`. Portanto, a variável `pont_idade` aponta para a variável `idade`. Um ponteiro pode ocupar desde 1 byte até 4 bytes, dependendo do tamanho do número usado para endereçar a memória. Nos exemplos a seguir, assumimos que o ponteiro ocupa apenas 1 byte.

Conforme ilustrado na Figura 1, cada byte da memória possui um endereço. No **esquema de endereçamento de memória**, os endereços de memória geralmente começam em 0 e terminam

no valor que representa o tamanho da memória menos 1. Nesta figura, note que a variável idade ocupa 2 bytes, ou seja, os endereços 704 e 705. Este é um típico exemplo de uma variável do tipo int. O ponteiro pont\_idade armazena o valor 704 que corresponde ao endereço inicial da variável idade. De forma geral, independentemente do tamanho que uma variável ocupa, um ponteiro sempre aponta para o endereço inicial desta variável (i.e., seu primeiro byte). Isto é possível desde que o tipo da variável apontada auxilia na leitura correta dos seus dados. Ou seja, se a variável ponteiro aponta para uma variável do tipo int, por exemplo, sabe-se que a variável ocupa 2 bytes (este exemplo é válido para uma plataforma específica. Revise a explicação sobre tipos de dados no capítulo 1, seção 1.5.2).

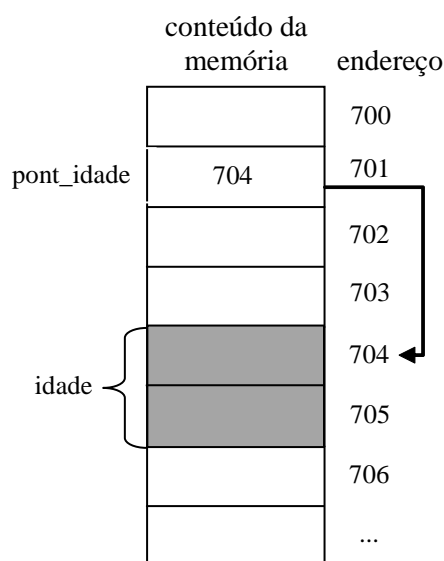


Figura 1. Exemplo do uso de uma variável ponteiro.

Ponteiros são recursos poderosos e importantes na programação em C. Há dois **usos principais de ponteiros**. O primeiro deles refere-se à alocação dinâmica de memória. Este tópico será abor-

dados na seção 4.4. Outro uso refere-se à passagem de parâmetros por referência para funções, o qual é tratado na seção 4.3.

O Programa 1 a seguir exemplifica o uso de ponteiros. Este programa trabalha com dados de dois funcionários (i.e., dados sobre o código e a idade de cada funcionário). O programa utiliza ponteiro para atribuir a idade do funcionário 1 para a idade do funcionário 2. Outros detalhes sobre a declaração e o uso de ponteiros também são exemplificados neste programa.

### Programa 1

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[])
5  {
6      int codFunc1 = 10; // código do funcionário 1
7      int idadeFunc1 = 35; // idade do funcionário 1
8      int codFunc2 = 11; // código do funcionário 2
9      int idadeFunc2;    // idade do funcionário 2
10
11     // declaração de um ponteiro
12     int *pont_idadeFunc; // ponteiro para a variável idadeFunc1
13
14     // impressão dos dados do funcionário 1
15     printf("Os dados do funcionario 1 sao:\n");
16     printf("codigo: %d\n", codFunc1);
17     printf("idade: %d\n", idadeFunc1);
18     printf("O endereco de idadeFunc1 e: %p\n", &idadeFunc1);
19     printf("\n");
20
21     // uso do ponteiro
22     pont_idadeFunc = &idadeFunc1;
23     idadeFunc2 = *pont_idadeFunc;
24
25     // impressão dos dados do funcionário 2
26     printf("Os dados do funcionario 2 sao:\n");
27     printf("codigo: %d\n", codFunc2);
28     printf("idade: %d\n", idadeFunc2);
29     printf("O endereco de idadeFunc2 e: %p\n", &idadeFunc2);
30     printf("\n");
31
```

```
32 // uso do ponteiro
33 *pont_idadeFunc = 20;
34
35 // impressão dos dados dos funcionários 1 e 2
36
37 printf("Os dados do funcionario 1 sao:\n");
38 printf("codigo: %d\n", codFunc1);
39 printf("idade: %d\n", idadeFunc1);
40 printf("O endereco de idadeFunc1 e: %p\n", &idadeFunc1);
41 printf("\n");
42
43 printf("Os dados do funcionario 2 sao:\n");
44 printf("codigo: %d\n", codFunc2);
45 printf("idade: %d\n", idadeFunc2);
46 printf("O endereco de idadeFunc2 e: %p\n", &idadeFunc2);
47 printf("\n");
48
49 system("PAUSE");
50 return 0;
51 }
```

No Programa 1, a **declaração do ponteiro** `pont_idadeFunc` é realizada na linha 12. Este ponteiro aponta para uma variável do tipo `int`, em particular neste programa para a variável `idadeFunc1`. A declaração de um ponteiro possui o formato **tipo \*variável**. Nesta declaração, o tipo deve ser o mesmo tipo da variável apontada pelo ponteiro. O caractere `*` indica que a variável é do tipo ponteiro.

O primeiro e o segundo uso de ponteiros são destacados nas linhas 22 e 23. A linha 22 inicializa o ponteiro `pont_idadeFunc` com o endereço inicial da variável `idadeFunc1`. Isto é realizado usando o **operador de ponteiro &**. O operador `&` pode ser lido como “o endereço de”. Assim, a linha 22 pode ser lida como “o ponteiro `pont_idadeFunc` recebe o endereço de `idadeFunc1`”. A Figura 2 ilustra esta atribuição no item (a). Desde que a variável `idadeFunc1` ocupa os endereços `0022FF70` e `0022FF71`, `pont_idadeFunc` recebe o valor `0022FF70`.

A linha 23 atribui um valor para a variável `idadeFunc2`. Este valor é o conteúdo do endereço apontado pelo ponteiro `pont_idadeFunc`. Isto é realizado usando o **operador de ponteiro**

\*. O operador \* pode ser lido como “no endereço apontado por”. Assim, a linha 23 pode ser lida como “idadeFunc2 recebe o valor que está armazenado no endereço apontado por pont\_idadeFunc”. A Figura 2 ilustra esta atribuição no item (b). Desde que o valor que está armazenado no endereço apontado por pont\_idadeFunc é 35, idadeFunc2 recebe 35.

O terceiro uso de ponteiros é destacado na linha 33. Nesta linha, o valor 20 será armazenado no endereço apontado por pont\_idadeFunc. Portanto, \*pont\_idadeFunc está se referindo ao conteúdo do endereço apontado. A Figura 3 ilustra esta atribuição no item (c). Desde que o endereço armazenado em pont\_idadeFunc é 022FF70, o conteúdo do endereço 022FF70 recebe o valor 20.

Por fim, vale destacar a **impressão de endereços na função printf( )**. O endereço de idadeFunc1 é impresso nas linhas 18 e 40, enquanto que o endereço de idadeFunc2 é impresso nas linhas 29 e 46. Para imprimir um endereço, a função printf( ) usa o especificador de formato %p (note que é necessário usar o símbolo % e a letra p quando se imprime ponteiros). A Figura 4 ilustra um exemplo de execução do Programa 1.

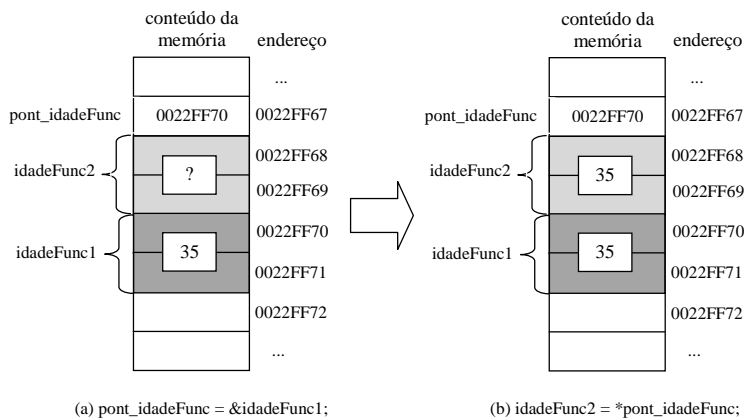


Figura 2. Conteúdo da memória após a execução das linhas 22 e 23.

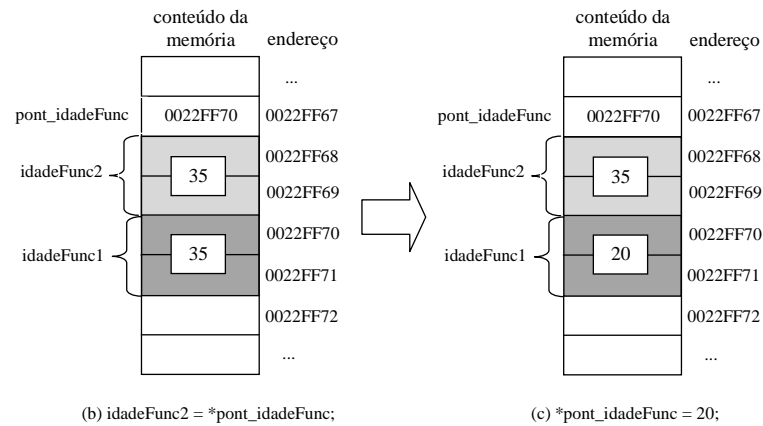


Figura 3. Conteúdo da memória após a execução da linha 33.

```

F:\Usuários\Ricardo\Programação\Dev-Cpp\LabProg\Capitulo 04\Programa 01\projeto01.exe
Os dados do funcionario 1 sao:
codigo: 10
idade: 35
O endereco de idadeFunc1 e: 0022FF70

Os dados do funcionario 2 sao:
codigo: 11
idade: 35
O endereco de idadeFunc2 e: 0022FF68

Os dados do funcionario 1 sao:
codigo: 10
idade: 20
O endereco de idadeFunc1 e: 0022FF70

Os dados do funcionario 2 sao:
codigo: 11
idade: 35
O endereco de idadeFunc2 e: 0022FF68

Pressione qualquer tecla para continuar. . .
  
```

Figura 4. Exemplo de execução do Programa 1.

## 4.2 Registros

Nesta seção, você aprenderá sobre os seguintes **conceitos** relacionados ao uso de registros:

estruturas, campos de uma estrutura, operador de acesso a campos ponto, operador de acesso a campos seta, definição de novos tipos com typedef, estruturas aninhadas, campos de bits, união e enumeração.

Um **registro** agrupa um conjunto de variáveis correlacionadas, as quais se referem a uma mesma entidade ou a um mesmo objeto do mundo real. Em vários exemplos de programas estudados nesta apostila, dados de funcionários, tais como código, idade, sexo e salário, foram lidos, processados e escritos usando diferentes variáveis. Por exemplo, para tratar da idade do funcionário foi usada a variável `idadeFunc`, enquanto para tratar do salário do funcionário foi usada a variável `salFunc`. O conceito de registro permite agrupar todos estes dados em uma única estrutura. Assim, pode-se criar um registro funcionário, o qual é composto de código, idade, sexo e salário. Cada parte de um registro é denominada de **campo do registro**.

O uso de registros facilita a organização dos dados e melhora a legibilidade dos programas. Portanto, seu uso é fortemente indicado sempre que um conjunto de variáveis relacionadas entre si for identificado. Na linguagem C, registros são chamados de **estruturas** (ou **struct**), e a partir deste momento, nós vamos referenciá-los desta forma.

O Programa 2 a seguir exemplifica o uso de estruturas. Este programa adapta o Programa 1 do Capítulo 1 para o uso de estruturas. O programa realiza a manipulação dos dados de 2 funcionários (i.e., código, idade, sexo e salário) e depois mostra os dados destes funcionários na saída padrão. Detalhes sobre a declaração e a utilização de estruturas são exemplificados neste programa.

## Programa 2

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[])
5  {
6
7      /* definição da estrutura de funcionários
8       com os campos codFunc, idadeFunc, sexoFunc e salFunc. */
9
10     struct funcionario
11     {
12         int codFunc; // código do funcionário
13         int idadeFunc; // idade do funcionário
14         char sexoFunc; // sexo do funcionário, M (masculino) e F (femini-
15         no)
16         float salFunc; // salário do funcionário
17     }; // atenção ao uso do ponto-e-vírgula
18
19     // declaração de duas variáveis do tipo estrutura
20     struct funcionario funcionario1;
21     struct funcionario funcionario2;
22
23     // entrada de dados do funcionário 1
24
25     printf("A seguir, entre com todos os dados do funcionario.\n\n");
26     printf("Digite o codigo: ");
27     scanf("%d", &funcionario1.codFunc);
28     printf("Digite a idade: ");
29     scanf("%d", &funcionario1.idadeFunc);
30     printf("Digite o sexo [F ou M]: ");
31     scanf("%c%c", &funcionario1.sexoFunc);
32     printf("Digite o salario (R$): ");
33     scanf("%f", &funcionario1.salFunc);
34     printf("\n");
35
36     // saída de dados do funcionario 1 para a tela (monitor de vídeo)
37
38     printf("Os dados do funcionario 1 sao:\n\n");
39     printf("Codigo: %d\n", funcionario1.codFunc);
40     printf("Idade: %d\n", funcionario1.idadeFunc);
41     printf("Sexo: %c\n", funcionario1.sexoFunc);
```



```

41     printf("Salario (R$): %.2f\n", funcionario1.salFunc);
42     printf("\n");
43
44     // atribuição dos dados do funcionário 1 para o funcionario 2
45     funcionario2 = funcionario1;
46
47     // saída de dados do funcionario 2 para a tela (monitor de vídeo)
48
49     printf("Os dados do funcionario 2 sao:\n\n");
50     printf("Codigo: %d\n", funcionario2.codFunc);
51     printf("Idade: %d\n", funcionario2.idadeFunc);
52     printf("Sexo: %c\n", funcionario2.sexoFunc);
53     printf("Salario (R$): %.2f\n", funcionario2.salFunc);
54     printf("\n");
55
56     // finalização do programa principal
57
58     system("PAUSE");
59     return 0;
60 }

```

No Programa 2, a **declaração da estrutura** funcionario é realizada nas linhas 10 a 16. A declaração desta estrutura possui o formato ilustrado na Figura 5. A linha 10 possui a identificação da estrutura pela palavra reservada **struct**, seguida do nome da estrutura (i.e., identificador). Toda a estrutura é delimitada por um par de chaves, dentro do qual são definidos os seus campos. Cada campo é definido de forma similar a uma declaração de variável. Ou seja, o formato de cada campo é **tipo nome\_campo ;**. Especial atenção deve ser dada ao uso do ponto-e-vírgula no final da declaração de cada campo e também no final da estrutura. No Programa 2, a estrutura funcionario possui 4 campos, denominados codFunc, idadeFunc, sexoFunc e salFunc.

```

struct nome_estrutura
{
    definição do campo 1 -----> definição do campo
    definição do campo 2             tipo nome_campo;
    ...
    definição do campo n
};

```

Figura 5. Declaração de uma estrutura e de seus campos.

A declaração de uma estrutura apenas define o seu formato, ou seja, define um novo tipo de dados. Este novo tipo de dados pode ser então usado na declaração de variáveis. De fato, para armazenar dados usando uma estrutura, é necessária a **declaração de uma variável do tipo da estrutura**. As linhas 19 e 20 ilustram a declaração das variáveis funcionario1 e funcionario2 como estruturas funcionario. O formato da declaração de uma variável do tipo estrutura é **struct nome\_estrutura nome\_variável ;**.

A Figura 6 ilustra graficamente o relacionamento entre a estrutura funcionario e a variável funcionario1. Cada retângulo representa um campo, ou seja, a estrutura funcionario define os campos codFunc, idadeFunc, sexoFunc e salFunc. Já a variável funcionario1 armazena os valores 121, 37, F e 1000.00, respectivamente nestes campos.

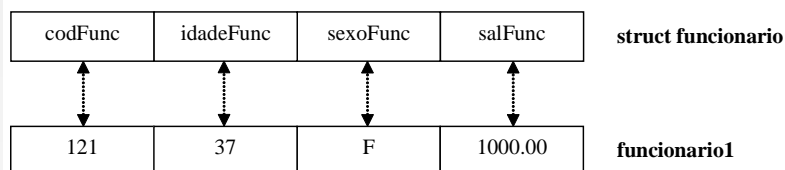


Figura 6. Relacionamento entre uma estrutura e uma variável de seu tipo.

O **acesso aos dados de uma variável do tipo estrutura** é realizado de duas formas. A primeira forma refere-se ao acesso aos **dados de todos os campos da variável**. Isto é realizado por meio do uso do nome da variável. Por exemplo, na linha 45 é realizada a atribuição de todos os dados da variável funcionario1 (i.e., dados de todos os seus campos) para a variável funcionario2. O resultado desta atribuição é ilustrado na Figura 7.

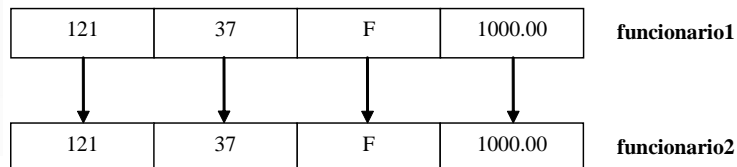
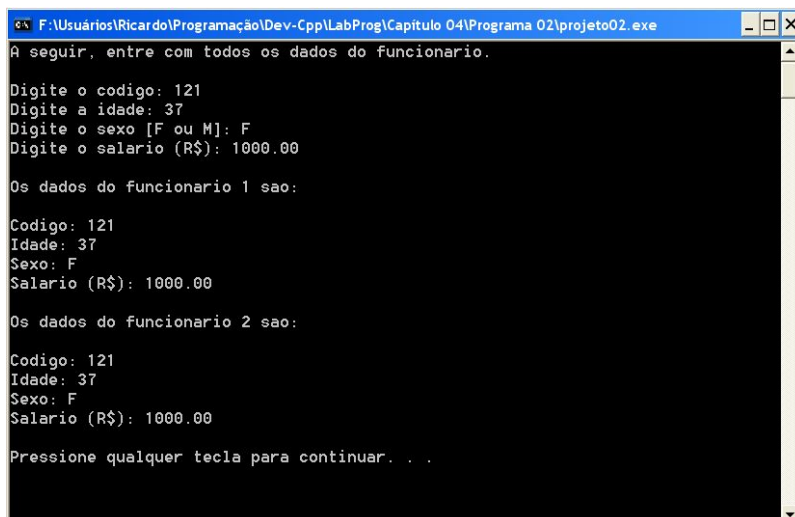


Figura 7. Atribuição dos dados de funcionario1 para funcionario2.

A segunda forma refere-se ao uso do **operador** de acesso a campos **ponto** (identificado pelo símbolo `.`). Este operador permite o **acesso individual a cada campo** de uma variável do tipo estrutura. O formato de seu uso é **nome\_variável.nome\_campo**. Note que não há espaço em branco entre o nome da variável e o nome do campo. Por exemplo, para acessar o valor do campo `codFunc` da variável do tipo estrutura `funcionario1`, utiliza-se `funcionario1.codFunc`, produzindo o resultado 121 conforme a Figura 7.

Em muitas situações, tais como a leitura e a escrita de dados, não é possível usar a primeira forma de acesso aos dados de uma variável do tipo estrutura. Nestas situações, os dados devem ser especificados campo a campo. No Programa 2, a leitura dos dados do `funcionario1` é realizada, campo a campo, nas linhas 26, 28, 30 e 32. De forma similar, o operador ponto também é necessário na impressão dos dados do `funcionario1`, a qual é realizada nas linhas 38 a 41, e na impressão dos dados do `funcionario2`, a qual é realizada nas linhas 50 a 53. A Figura 8 ilustra um exemplo de execução do Programa 2.



```
F:\Usuários\Ricardo\Programação\Dev-Cpp\LabProg\Capitulo 04\Programa 02\projeto02.exe
A seguir, entre com todos os dados do funcionario.

Digite o codigo: 121
Digite a idade: 37
Digite o sexo [F ou M]: F
Digite o salario (R$): 1000.00

Os dados do funcionario 1 sao:

Codigo: 121
Idade: 37
Sexo: F
Salario (R$): 1000.00

Os dados do funcionario 2 sao:

Codigo: 121
Idade: 37
Sexo: F
Salario (R$): 1000.00

Pressione qualquer tecla para continuar. . .
```

Figura 8. Exemplo de execução do Programa 2.

O Programa 3 a seguir exemplifica o uso de ponteiros para estruturas. Este programa realiza a manipulação dos dados de 2 funcionários (i.e., código, idade, sexo e salário). Inicialmente os dados do funcionário 1 são lidos e mostrados na tela usando uma variável do tipo estrutura. Em seguida, os dados do funcionário 2 são inicializados utilizando como base os dados do funcionário 1. Isto é feito por meio de um ponteiro para estrutura. Os dados do funcionário 1 também são alterados usando um ponteiro para estrutura. Ambos os dados de funcionário 1 e de funcionário 2 são então mostrados na tela após a manipulação destes com o uso de ponteiros.

### Programa 3

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[])
5  {
6
7      /* definição da estrutura de funcionários
8       com os campos codFunc, idadeFunc, sexoFunc e salFunc. */
9
10     struct funcionario
11     {
12         int codFunc; // código do funcionário
13         int idadeFunc; // idade do funcionário
14         char sexoFunc; // sexo do funcionário, M (masculino) e F (femini-
15         no)
16         float salFunc; // salário do funcionário
17     }; // atenção ao uso do ponto-e-vírgula
18
19     // declaração de variáveis
20     struct funcionario funcionario1; // variável do tipo estrutura
21     struct funcionario funcionario2; // variável do tipo estrutura
22     struct funcionario *pontFunc; // ponteiro para variável do tipo estrutura
23
24     // entrada de dados do funcionário 1
25     printf("A seguir, entre com todos os dados do funcionario.\n\n");
26     printf("Digite o codigo: ");
```

```
27 scanf("%d", &funcionario1.codFunc); // note o uso do operador &
28 printf("Digite a idade: ");
29 scanf("%d", &funcionario1.idadeFunc); // note o uso do operador
&
30 printf("Digite o sexo [F ou M]: ");
31 scanf("%*c%c", &funcionario1.sexoFunc); // note o uso do operador &
32 printf("Digite o salario (R$): ");
33 scanf("%f", &funcionario1.salFunc); // note o uso do operador &
34 printf("\n");
35
36 // saída de dados do funcionario 1 para a tela (monitor de vídeo)
37
38 printf("Os dados do funcionario 1 sao:\n\n");
39 printf("Codigo: %d\n", funcionario1.codFunc);
40 printf("Idade: %d\n", funcionario1.idadeFunc);
41 printf("Sexo: %c\n", funcionario1.sexoFunc);
42 printf("Salario (R$): %.2f\n", funcionario1.salFunc);
43 printf("\n");
44
45 // uso de ponteiro para estrutura
46
47 pontFunc = &funcionario1;
48
49 // uso do operador seta para a manipulação de campos
50
51 funcionario2.codFunc = 99;
52 funcionario2.idadeFunc = pontFunc->idadeFunc;
53 funcionario2.sexoFunc = pontFunc->sexoFunc;
54 funcionario2.salFunc = pontFunc->salFunc * 1.5;
55
56 // saída de dados do funcionario 2 para a tela (monitor de vídeo)
57
58 printf("Os dados do funcionario 2 sao:\n\n");
59 printf("Codigo: %d\n", funcionario2.codFunc);
60 printf("Idade: %d\n", funcionario2.idadeFunc);
61 printf("Sexo: %c\n", funcionario2.sexoFunc);
62 printf("Salario (R$): %.2f\n", funcionario2.salFunc);
63 printf("\n");
64
65 // alteração do conteúdo de alguns campos de funcionário 1
66 // usando ponteiro
67
68 pontFunc->codFunc = 79; // (*pontFunc).codFunc = 79;
69 pontFunc->idadeFunc = 25; // (*pontFunc).idadeFunc = 25;
```

```
70
71 // saída de dados do funcionario 1 para a tela (monitor de vídeo)
72
73 printf("Os dados do funcionario 1 sao:\n\n");
74 printf("Codigo: %d\n", funcionario1.codFunc);
75 printf("Idade: %d\n", funcionario1.idadeFunc);
76 printf("Sexo: %c\n", funcionario1.sexoFunc);
77 printf("Salario (R$): %.2f\n", funcionario1.salFunc);
78 printf("\n");
79
80 // finalização do programa principal
81
82 system("PAUSE");
83 return 0;
84 }
```

O Programa 3 declara o ponteiro pontFunc para o tipo estrutura funcionário na linha 21. Esta declaração é similar a qualquer outra declaração de ponteiros, ou seja, possui o mesmo formato (tipo \*nome\_ponteiro;). Note que o tipo de dados na linha 21 é struct funcionario. A atribuição de um endereço de uma variável do tipo estrutura para um ponteiro também é similar a qualquer outra atribuição de endereço para ponteiros. Portanto, usa-se o operador de ponteiro &. Na linha 47, pontFunc recebe o endereço inicial da variável estrutura funcionario1.

Já o acesso ao conteúdo de uma variável do tipo estrutura por meio de um ponteiro é realizado utilizando-se o **operador** de acesso a campos **seta** (símbolo ->). Por exemplo, na linha 52, o campo idadeFunc de funcionario2 recebe o valor do campo idadeFunc de funcionario1, desde que pontFunc aponta para funcionario1. Mais especificamente, o campo idadeFunc de funcionario2 recebe o conteúdo do campo idadeFunc do endereço apontado por pontFunc. A Figura 9 ilustra os valores dos campos de funcionario1 e funcionario2 após a execução do Programa 3 até a linha 55.

121	37	F	1000.00	<b>funcionario1</b>
99	37	F	1500.00	<b>funcionario2</b>

Figura 9. Atribuição dos dados de funcionario1 para funcionario2.

Note que o uso do operador seta substitui o uso conjunto do operador de ponteiro \* e do operador de acesso a campos ponto (símbolo .) para manipular o conteúdo do endereço apontado pelo ponteiro. Portanto, por exemplo, escrever `pontFunc->idadeFunc` é equivalente a escrever `(*pontFunc).idadeFunc`, como destacado na linha 69. O uso de `(*pontFunc).idadeFunc` também é válido, porém muito pouco usual e requer os parênteses para alterar a precedência dos operadores. É importante destacar que não é possível usar conjuntamente os operadores \* e seta. Por exemplo, `*pontFunc->codFunc` não é válido. A Figura 10 ilustra um exemplo de execução do Programa 3.

```
F:\Usuários\Ricardo\Programação\Dev-Cpp\LabProg\Capitulo 04\Programa 03\projeto03....
A seguir, entre com todos os dados do funcionario.

Digite o codigo: 121
Digite a idade: 37
Digite o sexo [F ou M]: F
Digite o salario (R$): 1000.00

Os dados do funcionario 1 sao:

Codigo: 121
Idade: 37
Sexo: F
Salario (R$): 1000.00

Os dados do funcionario 2 sao:

Codigo: 99
Idade: 37
Sexo: F
Salario (R$): 1500.00

Os dados do funcionario 1 sao:

Codigo: 79
Idade: 25
Sexo: F
Salario (R$): 1000.00

Pressione qualquer tecla para continuar. . .
```

Figura 10. Exemplo de execução do Programa 3.

## 4.3 Funções

Nesta seção, você aprenderá sobre os seguintes **conceitos** relacionados aos usos de funções:

conceito de função, função main( ), funções adicionais, cabeçalho e corpo de uma função, tipo de retorno de uma função, escopo e declarações locais, comando return, fluxo de execução, parâmetros formais e argumentos, argumentos e passagem de parâmetros por valor, argumentos e passagem de parâmetros por referência, protótipos de funções, passagem de estruturas para funções e uso de funções em expressões.



Um programa C é formado por um conjunto de funções. Todo programa possui uma **função main( )**, pela qual inicia-se a execução do programa. A função **main( )** pode chamar outras **funções adicionais**, as quais também podem chamar outras funções e assim sucessivamente. A função **main( )** já foi explicada no Capítulo 1. Nesta seção, o foco é na declaração e uso de funções adicionais.

Uma **função** é uma unidade autônoma e independente do programa que agrupa um conjunto de comandos para produzir um resultado específico. Uma função deve conter dentro dela todas as declarações de variáveis e todos os comandos necessários para produzir o resultado. Adicionalmente, uma função geralmente necessita de dados externos a ela, os quais são passados para a função por meio de parâmetros.

Conforme discutido na disciplina de “Construção de Algoritmos”, sub-rotinas (i.e., funções na linguagem C) possuem três utilidades principais: agrupar comandos e facilitar a organização funcional do programa, evitar a escrita repetida de grupos de comandos que devem ser executados diversas vezes em um programa e separar conjuntos de comandos que possuem propósitos específicos diferentes.

A **declaração de uma função** em C segue o formato ilustrado na Figura 11. A declaração de uma função especifica o que será executado pela função quando ela for invocada em algum ponto do programa. Uma função é composta de duas partes: o cabeçalho da função e o corpo da função. O **cabeçalho da função** consiste na definição de seu tipo, seguido do nome da função e de uma lista de parâmetros entre parênteses. Já o **corpo da função** é delimitado por { e }, dentro dos quais devem ser especificadas todas as variáveis locais e a sequência de comandos da função.

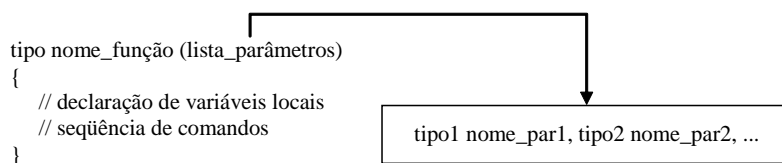


Figura 11. Formato de uma função em C.

O **tipo da função** indica o tipo de valor do retorno da função. Por exemplo, se uma função for do tipo `int`, ela retornará valores tais como 10, -5 e 5000, enquanto se uma função for do tipo `caractere`, ela retornará valores tais como 'a', 'M' e 'x'. O tipo de retorno da função pode ser qualquer tipo de dados básico da linguagem C (i.e., `char`, `int`, `float`, `double`, ponteiro). Uma função que retorna um valor pode ser utilizada em uma expressão, tal como `5 + raizQuadrada(3)`. Uma função pode, opcionalmente, não retornar nenhum valor. Neste caso, o seu tipo deve ser **void**. Este conceito corresponde ao conceito de procedimento descrito na disciplina de “Construção de Algoritmos”. Uma função do tipo `void` não pode ser utilizada em expressões. Por exemplo, considere o comando `a = a + 3 + f1(2);`. Caso a função `f1(x)` seja do tipo `void`, ela não poderá ser usada na expressão `a + 3 + f1(2)`, pois ela não retorna nenhum valor. Ao invés, uma função do tipo `void` é equivalente a um novo comando. Portanto, no exemplo, podemos usar o seguinte comando: `f1(2);`. A função será executada, processará dados e depois o fluxo de execução retornará ao próximo comando.

O **nome de uma função** deve seguir as mesmas regras de formação de nomes de variáveis e constantes. Assim, nomes tais como “calculaFatorial”, “raizQuadrada” e “calculaMediaProvas” são válidos. Lembre-se que os nomes das funções não devem coincidir com nomes já existentes de variáveis, constantes, tipos de dados, ou qualquer outra declaração dentro do programa. A escolha do nome de uma função deve refletir a sua funcionalidade. Fica claro nos exemplos de nomes citados neste parágrafo a funcionalidade das 3 funções.

A **lista de parâmetros** de uma função é colocada entre parênteses. Esta lista é composta por um conjunto de declarações da forma **tipo nome\_parâmetro**, as quais são separadas por vírgula. A lista de parâmetros de uma função pode conter 1 parâmetro, 2 parâmetros, 10 parâmetros, ou mais parâmetros. Além disso, a lista de parâmetros também pode ser vazia. Quando a lista de parâmetros é vazia, a palavra reservada **void** deve ser colocada dentro dos parênteses.

Um **parâmetro** pode ser usado para passar dados para uma função (i.e., parâmetro de entrada de dados), para retornar dados processados pela função (i.e., parâmetro de retorno de dados), ou com ambas funcionalidades de entrada e de retorno de dados. O uso de parâmetros é extremamente importante porque permite que a função interaja com dados externos de forma controlada. Desta forma, não é indicado, por exemplo, que uma função lide com variáveis globais (i.e., com variáveis visíveis em todo o programa e que podem ser alteradas em qualquer parte) em detrimento do uso de parâmetros. Dentro do corpo de uma função, um parâmetro pode ser usado da mesma forma que uma variável.

O **corpo da função** contém a declaração de variáveis e constantes locais, ou seja, variáveis e constantes que somente podem ser utilizadas dentro do corpo da função. Portanto, o escopo destas declarações é local à função (i.e., **escopo local**). Após as declarações, uma sequência de comandos é usada para atingir o propósito da função. Um comando especial que pode ser utilizado dentro do corpo da função é o comando **return**. Este comando encerra a execução da função e retorna um valor compatível com o tipo da função. A sua forma é **return valor** ou **return expressão**. Enquanto que para uma função que possui tipo é indicado o uso de pelo menos um comando return, uma função do tipo void não pode conter este comando.

O Programa 4 ilustra diversos aspectos do uso de funções. O programa realiza a manipulação dos dados de 2 funcionários (i.e., código, idade, sexo e salário). O programa realiza as seguintes ações: (1) leitura dos dados de 2 funcionários; (2) cálculo do maior salário dentre os salários dos 2 funcionários; (3) cálculo do aumento salarial para os 2 funcionários; e (4) troca dos dados entre os 2 funcionários (com exceção do código), ou seja, troca da idade, do sexo e do salário entre os 2 funcionários. Para cada uma destas ações, o programa imprime mensagens na tela. As Figuras Figura 12 e Figura 13 ilustram um exemplo e execução do Programa 4.

Você não precisa entender o Programa 4 completamente neste momento. Nós vamos estudar o programa por partes, e en-

tender detalhadamente cada aspecto do uso de funções. Ao final da explicação de todas as partes, você deverá ser capaz de compreender todo o código do programa e o seu funcionamento.

#### Programa 4

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  /* definição da estrutura de funcionários
5   com os campos codFunc, idadeFunc, sexoFunc e salFunc. */
6  struct funcionario
7  {
8      int codFunc; // código do funcionário
9      int idadeFunc; // idade do funcionário
10     char sexoFunc; // sexo do funcionário, M (masculino) e F (feminino)
11     float salFunc; // salário do funcionário
12 }; // atenção ao uso do ponto-e-vírgula
13
14 // protótipos de funções
15 void mensagens(void);
16 void leituraDadosFunc1(int *codigo, int *idade,
17     char *sexo, float *salario);
18 void leituraDadosFunc2(struct funcionario *F);
19 void imprimeDadosFunc(struct funcionario F);
20 float maiorSalario(float salario1, float salario2);
21 float aumentoSalarial(float salario);
22 void trocaDadosFunc(struct funcionario *funcionario1,
23     struct funcionario *funcionario2);
24
25 // função principal com início da execução do programa
26
27 int main(int argc, char *argv[])
28 {
29     // declaração
30     struct funcionario funcionario1;
31     struct funcionario funcionario2;
32     float salario;
33
34     mensagens();
35
36     // leitura dos dados de 2 funcionários
```

```
37     printf("Leitura e escrita inicial dos dados\n\n");
38     leituraDadosFunc1(&funcionario1.codFunc, &funcionario1.idadeFunc,
39                     &funcionario1.sexoFunc, &funcionario1.salFunc);
40     leituraDadosFunc2(&funcionario2);
41
42     // escrita dos dados de 2 funcionários
43     imprimeDadosFunc(funcionario1);
44     imprimeDadosFunc(funcionario2);
45
46     // cálculo do maior salário
47     printf("Calculo do maior salario\n\n");
48     salario = maiorSalario(funcionario1.salFunc, funcionario2.salFunc);
49     printf("O maior salario = %.2f\n\n", salario);
50
51     // aumento salarial para os 2 funcionários
52     printf("Aumento salarial para os 2 funcionarios\n\n");
53     funcionario1.salFunc = aumentoSalarial(funcionario1.salFunc);
54     funcionario2.salFunc = aumentoSalarial(funcionario2.salFunc);
55     imprimeDadosFunc(funcionario1);
56     imprimeDadosFunc(funcionario2);
57
58     // troca dados entre funcionarios
59     printf("Troca de dados entre os funcionarios\n\n");
60     trocaDadosFunc(&funcionario1, &funcionario2);
61     imprimeDadosFunc(funcionario1);
62     imprimeDadosFunc(funcionario2);
63
64     mensagens();
65
66     system("PAUSE");
67     return 0;
68 }
69
70 // imprime na tela um conjunto de mensagens
71
72 void mensagens(void)
73 {
74     printf("Ola este eh um programa que usa funcoes!\n");
75     printf("Lembre-se sempre de diferenciar o tipo de chamada\n");
76     printf("Existe passagem de parametro por valor e\n");
77     printf("passagem de parametro por referencia\n");
78     printf("\n");
79 }
80
```

```
81  /* esta função lê os dados de um funcionário e
82     retorna-os usando passagem de parâmetro por referência */
83
84  void leituraDadosFunc1(int *codigo,int *idade, char *sexo, float *salario)
85  {
86      printf("A seguir, entre com todos os dados do funcionario.\n\n");
87      printf("Digite o codigo: ");
88      scanf("%d", codigo);
89      printf("Digite a idade: ");
90      scanf("%d", idade);
91      printf("Digite o sexo [F ou M]: ");
92      scanf("%*c%c", sexo);
93      printf("Digite o salario (R$): ");
94      scanf("%f", salario);
95      printf("\n");
96  }
97
98  /* esta função tem a mesma funcionalidade da função leituraDadosFunc1,
99     porém utiliza uma estrutura como parâmetro */
100
101  void leituraDadosFunc2(struct funcionario *F)
102  {
103      printf("A seguir, entre com todos os dados do funcionario.\n\n");
104      printf("Digite o codigo: ");
105      scanf("%d", &F->codFunc);
106      printf("Digite a idade: ");
107      scanf("%d", &F->idadeFunc);
108      printf("Digite o sexo [F ou M]: ");
109      scanf("%*c%c", &F->sexoFunc);
110      printf("Digite o salario (R$): ");
111      scanf("%f", &F->salFunc);
112      printf("\n");
113  }
114
115  // esta função imprime na tela os dados de um funcionário
116
117  void imprimeDadosFunc(struct funcionario F)
118  {
119      printf("Os dados do funcionario %d sao:\n\n", F.codFunc);
120      printf("Codigo: %d\n", F.codFunc);
121      printf("Idade: %d\n", F.idadeFunc++);
122      printf("Sexo: %c\n", F.sexoFunc);
123      printf("Salario (R$): %.2f\n", F.salFunc);
124      printf("\n");
```

```
125     }
126
127 // esta função retorna o maior salário dentre o salário de 2 funcionários
128
129 float maiorSalario(float salario1, float salario2)
130 {
131     if (salario1 > salario2)
132         return salario1;
133     else return salario2;
134 }
135
136 // esta função retorna o novo salário de um funcionário
137
138 float aumentoSalarial(float salario)
139 {
140     return (salario * 1.5);
141 }
142
143 /* esta função troca os dados de 2 funcionários
144    com exceção do código do funcionário */
145
146 void trocaDadosFunc(struct funcionario *funcionario1,
147                     struct funcionario *funcionario2)
148 {
149     struct funcionario temp; // variável temporária para troca de dados
150     int codigo1, codigo2;
151
152     codigo1 = funcionario1->codFunc;
153     codigo2 = funcionario2->codFunc;
154
155     temp = *funcionario1;
156     *funcionario1 = *funcionario2;
157     *funcionario2 = temp;
158
159     funcionario1->codFunc = codigo1;
160     funcionario2->codFunc = codigo2;
161 }
```

```
F:\Usuários\Ricardo\Programação\Dev-Cpp\LabProg\Capitulo 04\Programa 04\projeto...
Ola este eh um programa que usa funcoes!
Lembre-se sempre de diferenciar o tipo de chamada
Existe passagem de parametro por valor e
passagem de parametro por referencia

Leitura e escrita inicial dos dados

A seguir, entre com todos os dados do funcionario.

Digite o codigo: 121
Digite a idade: 37
Digite o sexo [F ou M]: F
Digite o salario (R$): 1000.00

A seguir, entre com todos os dados do funcionario.

Digite o codigo: 18
Digite a idade: 48
Digite o sexo [F ou M]: M
Digite o salario (R$): 2000.00

Os dados do funcionario 121 sao:

Codigo: 121
Idade: 37
Sexo: F
Salario (R$): 1000.00

Os dados do funcionario 18 sao:

Codigo: 18
Idade: 48
Sexo: M
Salario (R$): 2000.00

Calculo do maior salario

O maior salario = 2000.00
```

Figura 12. Exemplo de execução do Programa 4 (parte 1).



```
F:\Usuários\Ricardo\Programação\Dev-Cpp\LabProg\Capitulo 04\Programa 04\projeto...
Aumento salarial para os 2 funcionarios
Os dados do funcionario 121 sao:
Codigo: 121
Idade: 37
Sexo: F
Salario (R$): 1500.00
Os dados do funcionario 18 sao:
Codigo: 18
Idade: 48
Sexo: M
Salario (R$): 3000.00
Troca de dados entre os funcionarios
Os dados do funcionario 121 sao:
Codigo: 121
Idade: 48
Sexo: M
Salario (R$): 3000.00
Os dados do funcionario 18 sao:
Codigo: 18
Idade: 37
Sexo: F
Salario (R$): 1500.00
Ola este eh um programa que usa funcoes!
Lembre-se sempre de diferenciar o tipo de chamada
Existe passagem de parametro por valor e
passagem de parametro por referencia
Pressione qualquer tecla para continuar. . .
```

Figura 13. Exemplo de execução do Programa 4 (parte 2).

### Fluxo de execução

O Programa 4 usa 7 funções adicionais: `mensagens()`, `leituraDadosFunc1()`, `leituraDadosFunc2()`, `imprimeDadosFunc()`, `maiorSalario()`, `aumentoSalarial()`, e `trocaDadosFunc()`. Estas funções são chamadas a partir da função `main()`, a qual define a seqüência de execução do programa.

O fluxo de execução do programa inicia-se no primeiro comando da função `main()`, ou seja, na linha 34. Nesta linha é efetuada a chamada para a função `mensagens()`. Neste momento, o fluxo de execução é desviado para o primeiro comando da função `mensagens()`, ou seja, é desviado para a linha 74. Os comandos desta função são executados seqüencialmente até que se encontre o seu fim, indicado pelo símbolo `}`, na linha 79. Ao encontrar o fim da função `mensagens()`, o fluxo de execução é retornado ao comando subsequente à chamada desta função em `main()`. Portanto, o fluxo de execução retorna para a linha 37 (note que as linhas 35 e 36 não possuem comandos).

A linha 37 apenas imprime uma mensagem na tela. Na linha 38, é executada uma chamada à função `leituraDadosFunc1()`. Novamente, o fluxo de execução é desviado para o primeiro comando de uma função. Neste caso, o fluxo é desviado para a linha 86, que é o primeiro comando da função `leituraDadosFunc1()`. Os comandos desta função são executados até o seu fim, na linha 96. Ao encontrar o fim da função `leituraDadosFunc1()`, o fluxo de execução é retornado ao comando subsequente à chamada desta função em `main()`. Portanto, o fluxo de execução retorna para a linha 40 (note que a chamada da função `leituraDadosFunc1()` ocupa as linhas 38 e 39).

De forma similar, a cada chamada de uma função, o fluxo de execução do programa é desviado para o primeiro comando da função, e é retornado para o comando subsequente da chamada da função quando o fim da função for encontrado. A Figura 14 ilustra o fluxo de execução do Programa 4 até a linha 47 da função `main()`.

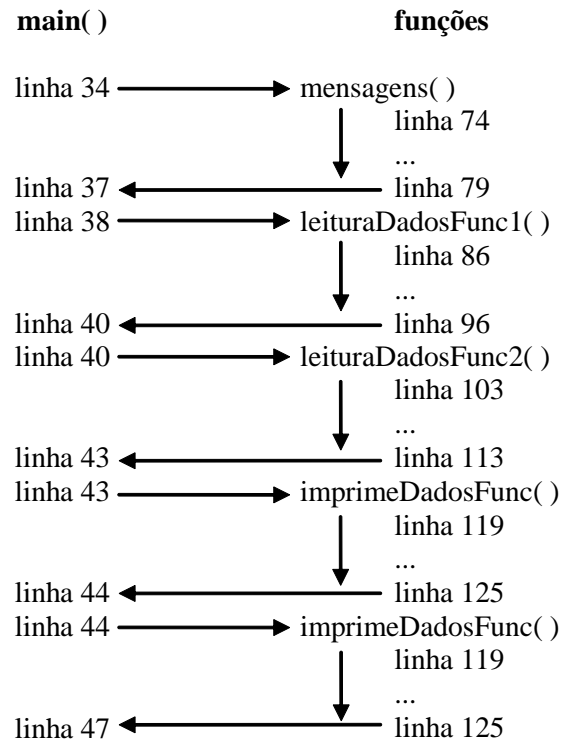


Figura 14. Fluxo de execução do Programa 4 até a linha 44.

Além do fim da função (i.e., símbolo `}`), outra forma de retorno do fluxo de execução de uma função para o comando subsequente à sua chamada é por meio do **comando `return`**. Quando o comando `return` é executado dentro de uma função, a execução desta função é interrompida imediatamente no ponto em que o comando aparece. Por exemplo, na linha 48 o fluxo de execução é desviado para a função `maiorSalario( )` durante o cálculo da parte direita do comando de atribuição. O primeiro comando desta função é executado na linha 131. Se as linhas 132 ou 133 forem executadas, dependendo do resultado do comando condicional, a execução da função `maiorSalario( )` é interrompida, e o fluxo de execução retorna para a linha 48 para que o comando de atribuição seja finalizado. A Figura 15 ilustra o fluxo de execução do Progra-

ma 4 da linha 48 para a função maiorSalario( ), considerando que o comando condicional da 131 produziu um resultado verdadeiro.

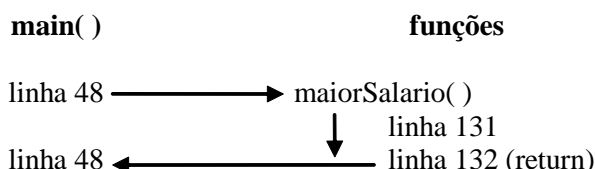


Figura 15. Fluxo de execução do Programa 4 da linha 48 para maiorSalario( ).

### Valor de retorno de uma função

Funções declaradas do tipo void não retornam nenhum valor. Portanto, estas funções não possuem o comando return e os seus comandos são executados seqüencialmente desde o primeiro comando após { até o fim da função. Exemplos deste tipo de função são: mensagens( ), leituraDadosFunc1( ), leituraDadosFunc2( ), imprimeDadosFunc( ) e trocaDadosFunc( ). Funções do tipo void não podem ser utilizadas em expressões desde que não retornam valor. As linhas 43 e 64 são alguns exemplos de chamadas de função do tipo void. Vale lembrar que funções declaradas do tipo void correspondem ao conceito de procedimento, conforme estudado na disciplina de “Construção de Algoritmos”.

Quando uma função possui um tipo, o valor retornado pela função é determinado pelo comando return. Por exemplo, na função maiorSalario( ), a linha 132 possui o comando return salario1, indicando que a função retornará o valor armazenado na variável salario1. Note que a função maiorSalario( ) é do tipo float, o mesmo tipo do parâmetro salario1. Uma função pode possuir um ou mais comandos return, tal como a função maiorSalario( ). Entretanto, somente um único comando return é executado dentro de uma função. Caso o comando return seja omitido em uma função que possui tipo, um valor padrão é retornado quando a função atinge o seu fim. Como este valor não é padronizado, a ausência de um comando return não é uma boa prática de programação, podendo introduzir problemas na execução do programa. Portanto, sempre utilize o comando return com funções que possuem tipo.

### Argumentos e passagem de parâmetros por valor

Na chamada de uma função, os nomes colocados dentro dos parênteses e separados por vírgulas são chamados de **argumentos**. Um argumento é usado para passar dados para uma função. Por exemplo, na linha 43, `funcionario1` é o único argumento da chamada da função `imprimeDadosFunc()`. Este argumento passa para a função `imprimeDadosFunc()` o valor da variável `funcionario1`. Já na linha 48, `funcionario1.salFunc` e `funcionario2.salFunc` são os 2 argumentos da chamada da função `maiorSalario()`. Isto significa que os valores destas 2 variáveis são passados para a função `maiorSalario()`.

Para receber os dados dos argumentos, uma função possui uma **lista de parâmetros formais**. Cada **parâmetro formal** define o tipo de dados que o argumento deve possuir na chamada da função. Além disso, cada parâmetro formal possui um nome, o qual pode ser diferente do nome do argumento. O número de argumentos na chamada de uma função e o número de parâmetros formais declarados nesta função deve ser o mesmo. Além disso, a ordem é importante, desde que o primeiro argumento corresponda ao primeiro parâmetro formal, o segundo argumento corresponda ao segundo parâmetro formal, e assim sucessivamente. No Programa 4, a função `maiorSalario()` declara 2 parâmetros formais do tipo `float`, chamados `salario1` e `salario2`. Assim, na chamada desta função na linha 48, são passados 2 argumentos do tipo `float`. O argumento `funcionario1.salFunc` corresponde ao parâmetro formal `salario1`, enquanto que o argumento `funcionario2.salFunc` corresponde ao parâmetro formal `salario2`.

Na linguagem C, a **passagem de parâmetros** sempre é feita **por valor**. Em outras palavras, o valor do argumento sempre é **copiado** para o parâmetro formal correspondente. O parâmetro formal age dentro de uma função como uma variável local e, portanto, pode ser utilizado em expressões, atribuições e comparações. Por exemplo, suponha que o argumento `funcionario1` possua os valores de seus campos conforme representados na Figura 16. Na chamada da função `imprimeDadosFunc()` da linha 43, es-

tes valores são copiados para o parâmetro formal F. O fato dos dados serem copiados faz com que qualquer alteração no parâmetro formal dentro da função não seja refletida no argumento. Ou seja, se um parâmetro formal for alterado, tal como é feito na linha 121 por meio de `F.idadeFunc++` para 38, o seu argumento correspondente não será alterado, ou seja, `funcionario1.idadeFunc` permanecerá com o valor de 37. Isto é ilustrado na Figura 17.

No Programa 4, os seguintes protótipos ilustram a declaração de funções com passagem de parâmetros por valor.

```
19 void imprimeDadosFunc(struct funcionario F);
20 float maiorSalario(float salario1, float salario2);
21 float aumentoSalarial(float salario);
```

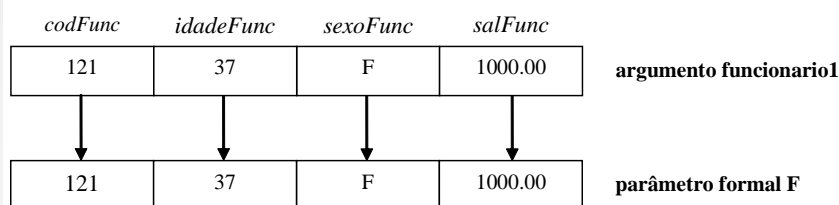


Figura 16. Relacionamento entre o argumento `funcionario1` e o parâmetro formal `F`.

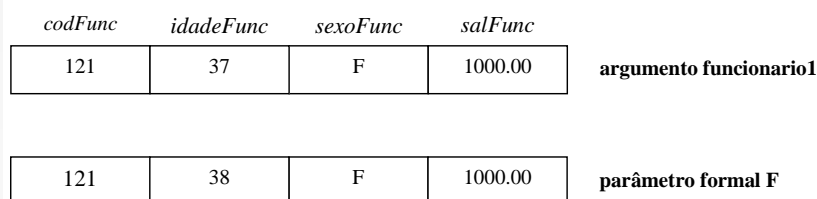


Figura 17. Alteração de um parâmetro formal sem alterar seu argumento.

### Argumentos e passagem de parâmetros por referência

Em algoritmos, o uso da palavra reservada **var** antes de um parâmetro formal define que este parâmetro é passado por referência. Em uma **passagem de parâmetros por referência**, o argumento e o parâmetro formal correspondem ao mesmo endereço de memória e, portanto, qualquer alteração realizada no parâmetro formal dentro da função é refletida no argumento. A passagem de parâmetros por referência é usada quando deseja-se retornar valo-

res para fora de uma função, além do valor de retorno da própria função. Os valores retornados são valores que foram resultantes de algum processamento ou de leitura de dados.

A linguagem C não possui um mecanismo automático para a passagem de parâmetros por referência. Para simular esta passagem, utiliza-se ponteiros, ou seja, passa-se por valor o endereço de um argumento para um parâmetro formal. Dentro da função, esse endereço pode ser utilizado pelo parâmetro formal para alterar o valor do argumento. Em outras palavras, qualquer alteração feita pelo ponteiro dentro da função é refletida diretamente no valor do argumento correspondente.

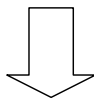
Por exemplo, no Programa 4, a função `leituraDadosFunc1` possui 4 parâmetros formais declarados como ponteiros: `*codigo`, `*idade`, `*sexo`, `*salario` (linha 84). Desta forma, estes parâmetros receberão como valor um endereço. Na chamada da função `leituraDadosFunc1` na linha 38, são passados os endereços dos 4 argumentos, respectivamente: `&funcionario1.codFunc`, `&funcionario1.idadeFunc`, `&funcionario1.sexoFunc` e `&funcionario1.salFunc`. A linha 88 passa para a função `scanf( )` o valor de `codigo`, que corresponde ao endereço do argumento `funcionario1.codFunc`. Assim, a leitura de um valor pela função `scanf( )` é armazenada em `funcionario1.codFunc`. Isto é realizado de forma similar para todos os demais parâmetros. A Figura 18 ilustra este exemplo.

No Programa 4, os seguintes protótipos ilustram a declaração de funções com passagem de parâmetros por referência. Note o uso do operador `*` em cada parâmetro passado por referência.

```
16 void leituraDadosFunc1(int *codigo, int *idade
17                          char *sexo, float *salário);
18 void leituraDadosFunc2(struct funcionario *F);
22 void trocaDadosFunc(struct funcionario *funcionario1,
23                      struct funcionario *funcionario2);
```

<i>codFunc</i> endereço 70	<i>idadeFunc</i> endereço 74	<i>sexoFunc</i> endereço 78	<i>salFunc</i> endereço 79	argumentos para funcionario1
<i>codigo</i>	<i>idade</i>	<i>sexo</i>	<i>salario</i>	parâmetros formais
70	74	78	79	

Relacionamento entre os argumentos e os parâmetros na chamada da função



Conteúdo dos argumentos após a leitura de dados

<i>codFunc</i> endereço 70	<i>idadeFunc</i> endereço 74	<i>sexoFunc</i> endereço 78	<i>salFunc</i> endereço 79	argumentos para funcionario1
121	37	F	1000.00	
<i>codigo</i>	<i>idade</i>	<i>sexo</i>	<i>salario</i>	parâmetros formais
70	74	78	79	

Figura 18. Exemplo de relacionamento entre argumentos e parâmetros formais.

### Protótipos

No Programa 4, os protótipos das 7 funções adicionais são declarados nas linhas 15 a 23. O conceito de protótipo já foi explicado no Capítulo 1. Vale lembrar que um protótipo de uma função corresponde somente ao seu cabeçalho, e que é finalizado com ponto-e-vírgula (símbolo ;). Os protótipos de função devem ser colocados antes do início da função `main()`, ou seja, antes das chamadas das funções adicionais.

### Passagem de estruturas para funções

No Programa 4, funcionários são armazenados como estruturas compostas por 4 campos, ou seja, os campos `codFunc`,



idadeFunc, sexoFunc e salFunc. A funcionalidade de leitura dos dados dos funcionários é implementada de 2 formas diferentes. Na função `leituraDadosFunc1( )`, cada campo é passado como um argumento distinto. Este uso já foi discutido anteriormente neste Capítulo. Por outro lado, na função `leituraDadosFunc2( )`, os dados do funcionário são passados em um único argumento do tipo estrutura. Esta segunda forma facilita a passagem dos dados, mas deve ser utilizada apenas quando todos os campos forem utilizados dentro da função.

Na chamada da função `leituraDadosFunc2` na linha 40, é passado o endereço da variável `funcionario2` do tipo estrutura (i.e., `&funcionario2`). Isto permite a alteração de valores dos campos que compõem esta estrutura. Como a passagem é feita por ponteiros e o argumento é uma estrutura, o acesso aos campos desta estrutura dentro da função é realizado usando o operador seta (símbolo `->`). Note que o parâmetro formal `F` da função `leituraDadosFunc2( )` recebe o endereço inicial da variável `funcionario2`. Para acessar cada campo pela função `scanf( )` dentro do corpo da função, deve ser passado o endereço do campo, o qual é indicado pelo símbolo `&`. O seguinte exemplo pode ser encontrado na linha 105: `scanf("%d", &F->codFunc)`. Isto é equivalente a `scanf("%d", &(*F).codFunc)`.

Outro exemplo de passagem de uma estrutura completa para uma função é realizado na linha 43. Na chamada da função `imprimeDadosFunc`, o argumento do tipo estrutura é passado por valor sem utilizar ponteiros. Desta forma, o acesso aos campos da estrutura dentro do corpo da função é realizado usando o operador ponto (símbolo `.`). O seguinte exemplo pode ser encontrado na linha 120: `printf("Codigo: %d\n", F.codFunc)`.

### **Função `trocaDadosFunc( )`**

A função `trocaDadosFunc( )` do Programa 4 necessita de uma explicação separada. Esta função tem como objetivo trocar os dados dos campos `idadeFunc`, `sexoFunc` e `salFunc` entre 2 funcionários. Portanto, após a troca, os dados iniciais destes campos

do funcionário 1 estarão armazenados em funcionário 2, e vice-versa.

Esta função possui 2 parâmetros formais que são ponteiros para a estrutura funcionario. Portanto, na sua chamada na linha 60, os seus argumentos são passados usando o operador de ponteiro & (i.e., são passados os seus endereços). As linhas 149 e 150 realizam declarações de variáveis locais, cujo escopo é o corpo da função, ou seja, estas variáveis são visíveis e podem ser usadas apenas dentro da função trocaDadosFunc( ). Nas linhas 152 e 153, são armazenados os códigos iniciais das variáveis funcionario1 e funcionario2. Isto previne a perda destes dados nas trocas realizadas nas linhas 155 a 157. Na linha 155, os dados de funcionario1 (i.e., de todos os seus campos) são atribuídos a uma variável temporária. Após, na linha 156, os dados de funcionario1 recebem os dados de funcionario2 (i.e., todos os campos são copiados). Por fim, na linha 157, os dados de funcionario2 recebem os dados de temp, que possui os valores iniciais de funcionario1. Note que, para acessar os dados de uma variável do tipo estrutura usando ponteiros, foi usado o operador de ponteiros \*. Nas linhas 159 e 160 os campos codFunc de funcionario1 e de funcionario2 são atualizados com os seus valores originais.

Agora que todas as partes do Programa 4 foram explicadas, você deve reler o programa e tentar entendê-lo como um todo!

## 4.4 Explicações Adicionais

Esta seção reforça a explicação de alguns dos conceitos estudados. Para tanto, serão usados exemplos de trechos de programas.

### Símbolos usados com ponteiros

Para usar ponteiros, são utilizados os operadores \* e &. Estes símbolos possuem diferentes significados na linguagem C, de-

pendendo do contexto. Portanto, um mesmo símbolo pode representar diferentes conceitos. O símbolo `*` em uma expressão aritmética representa multiplicação. Este símbolo, quando usado na declaração de variáveis, especifica uma variável do tipo ponteiro. Já em comandos de atribuição, este símbolo antecedendo uma variável refere-se a um ponteiro, o qual indica o conteúdo do endereço apontado. De forma similar, o símbolo `&` pode representar o operador bit-a-bit ou pode representar o endereço de uma variável.

### Aritmética de ponteiros

Ponteiros armazenam endereços de memória. Com ponteiros, pode-se usar os operadores aritméticos de adição e de subtração. Estes operadores aritméticos modificam o conteúdo de um ponteiro, aumentando ou diminuindo o valor do endereço armazenado. O trecho de código a seguir exemplifica o uso de operadores aritméticos com ponteiros. O ponteiro `pont_idade` aponta para uma variável do tipo `int`. Supondo que o tipo `int` ocupe 2 bytes, cada vez que `pont_idade` é incrementado, seu endereço é acrescido do tamanho de 2 bytes. Este é um aspecto crucial na aritmética de ponteiros. O incremento e o decremento do conteúdo de ponteiros é feito de acordo com o tamanho do tipo base apontado pelo ponteiro. Desta forma, deve-se sempre indicar o tipo apontado por um ponteiro na sua declaração.

Em ocasiões especiais, pode-se declarar um **ponteiro do tipo void** (i.e., declarado como `void *pont_void`). Este tipo de ponteiro pode ser usado para apontar para diferentes tipos de dados em um mesmo programa. Porém, o controle do aumento e do decremento de um ponteiro void deve ser realizado sem usar aritmética de ponteiros.

```
...
int *pont_idade;           // ponteiro do tipo int
...
pont_idade = &idade;       // assume que endereço de idade é 704
pont_idade++;              // conteúdo de pont_idade recebe 706
pont_idade+=3;             // conteúdo de pont_idade recebe 712
...
```

### Declaração de ponteiros com inicialização

Um ponteiro pode ser inicializado juntamente com a sua declaração. O trecho abaixo inicializa o ponteiro `pont_idade` com o endereço de memória 704. Note que, em uma declaração de variáveis, `*pont_idade` não se refere ao conteúdo do endereço apontado. Em contrapartida, em uma linha de comando, tal como em `idade = *pont_idade`, `*pont_idade` refere-se ao conteúdo do endereço apontado.

```
...  
int *pont_idade = 704; // declaração e inicialização  
...  
idade = *pont_idade;  
...
```

### Atribuição de ponteiros

Um ponteiro pode receber o valor de outro ponteiro de forma similar ao que ocorre em um comando de atribuição com outros tipos de variáveis. Isto significa que um ponteiro receberá o endereço de memória armazenado em outro ponteiro. O trecho a seguir ilustra um exemplo de atribuição. Como resultado final, `pont_idade2` irá armazenar o endereço 704. Portanto, após o comando de atribuição, ambos os ponteiros apontam para o mesmo endereço.

```
...  
int *pont_idade1 = 704;  
int *pont_idade2;  
...  
pont_idade2 = pont_idade1;  
...
```

### Comparação de ponteiros

De forma similar à atribuição de ponteiros, ponteiros podem ser comparados entre si usando operadores relacionais e de igualdade. Neste caso, compara-se os endereços armazenados nos ponteiros. O trecho a seguir ilustra a comparação de dois ponteiros usando o operador relacional `>` para decidir a execução de um trecho de programa.

```
...  
if (pont_idade1 > pont_idade2)  
{  
    // sequência de comandos  
}  
...
```

### Ponteiro nulo

Quando um ponteiro não aponta para nenhum endereço válido, o ponteiro é inicializado com o endereço 0. Esta é uma convenção usada por programadores C. Para facilitar o seu uso, a linguagem C define a constante `NULL` para representar o endereço 0. O trecho de programa a seguir ilustra a inicialização do ponteiro `pont_idade` com o valor `NULL`.

Sempre que um ponteiro ficar temporariamente sem uso, deve-se atribuir `NULL` a ele. Caso contrário, erros podem acontecer durante a execução do programa, os quais podem inclusive travar o computador. Por exemplo, um ponteiro que armazena um endereço arbitrário pode estar apontando para uma área de memória que contém o próprio conjunto de instruções do programa. A modificação do conteúdo apontado por este ponteiro irá causar a perda da instrução e, por conseguinte, a execução incorreta do programa ou o travamento do programa. Previne-se este erro atribuindo o valor `NULL` ao ponteiro sem uso.

```
...  
int *pont_idade;  
...  
pont_idade = NULL; // pont_idade = 0  
...
```

### Alocação dinâmica de memória

Quando uma variável de qualquer tipo é declarada, uma porção de espaço da memória é reservada para armazenar o conteúdo desta variável. Para a grande maioria das situações, sabe-se na escrita do programa o tamanho necessário de memória que deve ser alocado para os dados. No entanto, em algumas situações, desconhece-se a quantidade de memória que os dados irão ocupar. Assim, a declaração de variáveis conforme já estudamos não se aplica. Nestas situações, deve-se alocar memória em tempo de execução do programa.

A linguagem C oferece funções específicas para o gerenciamento de memória alocada dinamicamente. O uso destas funções é realizado por meio de ponteiros. A função `malloc( )` aloca uma porção de memória de tamanho `n`, onde `n` é passado como parâmetro. Esta função retorna o endereço da porção de memória alocada, a qual deve ser armazenada em um ponteiro. De forma oposta, a função `free( )` libera uma porção de memória alocada dinamicamente, ou seja, esta porção pode ser novamente usada em alocações subsequentes. Para tanto, deve-se passar como parâmetro da função `free( )` o endereço inicial da porção de memória alocada. Este endereço é frequentemente armazenado em um ponteiro.

O trecho de programa abaixo ilustra o uso das funções `malloc( )` e `free( )`. Neste trecho, `pont_idade` é um ponteiro do tipo `int`. Portanto, na função `malloc( )`, solicita-se a alocação de uma porção de memória do tamanho de um tipo `int`. Este tamanho é determinado pela função `sizeof( )`. O teste condicional verifica se a alocação de memória ocorreu com sucesso, ou seja, se `pont_idade` recebeu um valor diferente de 0 (i.e., `NULL`). O uso da função `free( )` é ilustrado no final do trecho do programa.

```
...
int *pont_idade;
...
pont_idade = malloc(sizeof(int));
if (!pont_idade) // pont_idade == NULL
{
    printf("Erro na alocação de memória\n");
    exit(1);
}
else // pont_idade != NULL
{
    printf("O valor de pont_idade é: %p\n", pont_idade);
}
...
// libera a memória quando pont_idade não for mais usado no programa
free(pont_idade);
...
```

### Erros comuns com ponteiros

São dois os erros mais comuns que ocorrem no uso de ponteiros. O primeiro deles refere-se a não inicialização de um ponteiro antes de seu primeiro uso. No trecho abaixo, o ponteiro `pont_idade` não foi inicializado e, portanto, pode conter qualquer endereço de memória. Este endereço provavelmente será um endereço inválido. O comando de atribuição poderá causar um erro de execução do programa.

```
...
int idade;
int *pont_idade;
idade = *pont_idade;
...
```

O segundo erro refere-se ao esquecimento do uso do operador `&`. Assim, em um comando de atribuição, a ausência deste operador faz com que a variável ponteiro receba o valor armazenado na variável do lado direito do comando de atribuição, e não o endereço desta variável. No trecho de programa a seguir, o comando correto seria `pont_idade = &idade`.

```
...  
int idade = 10;  
int *pont_idade;  
pont_idade = idade; // pont_idade recebe 10  
...
```

### Declaração conjunta de estruturas e variáveis

Usualmente, a declaração de uma estrutura e das variáveis do tipo desta estrutura é realizada separadamente. Por exemplo, no Programa 3, enquanto a declaração da estrutura funcionario é realizada nas linhas 10 a 16, a declaração das variáveis funcionario1 e funcionario2 deste tipo de estrutura é feita nas linhas 19 e 20. Entretanto, é possível efetuar a declaração conjunta de uma estrutura e de suas variáveis. No primeiro trecho de programa a seguir, as variáveis funcionario1 e funcionario2 são declaradas conjuntamente com a estrutura funcionario. Já no segundo trecho, o nome da estrutura é omitido, desde que somente nesta parte do programa esta estrutura é usada.

```
...  
struct funcionario  
{  
    int codFunc; // código do funcionário  
    int idadeFunc; // idade do funcionário  
    char sexoFunc; // sexo do funcionário, M (masculino) e F (feminino)  
    float salFunc; // salário do funcionário  
} funcionario1, funcionario2;  
...
```

```
...  
struct  
{  
    int codFunc; // código do funcionário  
    int idadeFunc; // idade do funcionário  
    char sexoFunc; // sexo do funcionário, M (masculino) e F (feminino)  
    float salFunc; // salário do funcionário  
} funcionario1, funcionario2;  
...
```



### Uso de typedef para simplificar a declaração de variáveis do tipo estrutura

A palavra reservada typedef permite a criação de novos tipos de dados. O seu formato é **typedef tipo novo\_tipo;**. Assim, um novo tipo de dados é criado com base em um tipo existente, e pode ser usado na declaração de variáveis. Por exemplo, o trecho de programa a seguir cria um novo tipo de dados chamado tipoCodigo que corresponde a um número inteiro e declara a variável codFunc deste tipo.

```
...  
typedef int tipoCodigo;  
tipoCodigo codFunc;  
...
```

O uso de typedef simplifica a declaração de variáveis do tipo estrutura. No trecho de programa a seguir, funcionario é um novo tipo de dados que corresponde a uma estrutura (com os campos codFunc, idadeFunc, sexoFunc e salFunc). Note que funcionario não é uma variável, e sim o nome de um novo tipo de dados. A declaração de variáveis do tipo estrutura pode ser realizada com base neste novo tipo de dados, sem o uso da palavra struct. Isto melhora a legibilidade do programa.

```
...  
typedef struct  
{  
    int codFunc; // código do funcionário  
    int idadeFunc; // idade do funcionário  
    char sexoFunc; // sexo do funcionário, M (masculino) e F (feminino)  
    float salFunc; // salário do funcionário  
} funcionario;  
...  
funcionario funcionario1, funcionario2;
```

## Estruturas aninhadas

Uma estrutura pode conter campos que também são estruturas. Quando isto ocorre, tem-se uma estrutura aninhada. O trecho de programa a seguir ilustra a declaração da estrutura `funcionario`, que possui um dos seus campos (i.e., `endFunc`) como uma estrutura `endereco`. O trecho também ilustra o acesso ao campo `cidade` do campo `endFunc` de `funcionario1`. Para cada estrutura aninhada, acrescenta-se o uso do operador ponto.

```
...
struct endereco
{
    char rua[40];
    int numero;
    char complemento[10];
    char cidade[35];
    char estado[2]; // sigla do estado
    char CEP[8];
};

struct funcionario
{
    int codFunc; // código do funcionário
    int idadeFunc; // idade do funcionário
    char sexoFunc; // sexo do funcionário, M (masculino) e F (feminino)
    float salFunc; // salário do funcionário
    struct endereco endFunc; // endereço do funcionário
} funcionario1, funcionario2;
...
printf("O funcionario 1 mora na cidade: %s", funcionario1.endFunc.cidade);
...
```

## Uso de operadores relacionais com estruturas

O uso de operadores relacionais (tais como, `>`, `==` e `<=`) somente é possível quando aplicado aos campos de uma estrutura. Portanto, estes operadores não podem ser usados com variáveis do tipo estrutura. O trecho a seguir ilustra um uso correto e um uso incorreto de operadores relacionais com estruturas.

```
...
struct funcionario
{
    int codFunc; // código do funcionário
    int idadeFunc; // idade do funcionário
    char sexoFunc; // sexo do funcionário, M (masculino) e F (feminino)
    float salFunc; // salário do funcionário
} funcionario1, funcionario2;

...
if (funcionario1.idadeFunc < 18) // CORRETO
{
    ...
}

...
if (funcionario2 > funcionario1) // INCORRETO - ERRO!
{
    ...
}

...
```

### Campos de bits

Campos de bits consistem em uma adaptação de estruturas para representar bits. Nesta adaptação, cada campo possui um tamanho em bits. Assim, vários campos podem compartilhar um mesmo byte. Portanto, campos de bits permitem melhorar a alocação de espaço de uma estrutura quando os campos podem ser representados por bits.

No trecho de programa a seguir, 1 byte é usado para representar um campo de bits chamado portaSerial. Este byte é compartilhado da seguinte forma: os 3 primeiros bits são usados para acionar a porta serial; o quarto bit é usado para restabelecer a porta; o quinto e o sexto bits são usados para verificar a paridade; e os últimos dois bits restantes não são utilizados. Caso portaSerial fosse declarada como uma estrutura, e assumindo que cada um dos seus campos fosse declarado como um caractere, ela ocuparia 4 bytes ao invés de 1 byte. Note que a declaração de um campo de bits é quase idêntica à declaração de uma estrutura, diferindo apenas no uso de dois pontos e tamanho (i.e., : **tamanho**) após a definição de cada campo. O uso do modificador unsigned permite

aproveitar todos os bits, desde que não é necessário representar o sinal para números negativos.

```
...
struct portaSerial
{
    unsigned int acionaPorta: 3; // 3 bits para acionar porta
    unsigned int reset: 1;      // 1 bit para restabelecer a porta
    unsigned int paridade: 2;   // bits de paridade
    unsigned int vazio: 2;      // bits não utilizados
} portaA;
...
portaA = 0xAF; // atribui o valor hexadecimal AF a variável porta
               // note o uso de 0x para indicar base 16 (hexadecimal)
               // isso corresponde a (10101111)2
               // portanto, acionaPorta recebe 101, reset recebe 0,
               // paridade recebe 11 e vazio recebe 11
...
```

## União

União consiste em uma variação de estruturas. Em uma união, os diversos campos que a compõem compartilham a mesma porção de espaço de memória. Por exemplo, no trecho a seguir, os campos RA, codFunc e CPF compartilham a mesma porção de espaço. O espaço alocado pela união corresponde ao tamanho do seu maior campo. No exemplo corrente, a união pessoa alocará 12 bytes, que serão compartilhados pelos três campos (Figura 19). O uso de união, portanto, permite economia na alocação de espaço de memória.

A união é utilizada quando uma entidade do mundo real pode assumir diferentes significados no programa. Por exemplo, uma pessoa pode ser representada pelo seu RA se ela for um estudante, ou pode ser representada pelo seu código se ela for um funcionário ou ainda pode ser representada pelo seu CPF caso ela não seja estudante ou funcionário. No trecho a seguir, é declarada a variável p1 do tipo união pessoa. Em diferentes partes do programa, esta variável pode estar armazenando dados diferentes, em função do tipo de pessoa.

O acesso aos campos de uma união é realizado da mesma forma que o acesso aos campos de uma estrutura, ou seja, por meio do operador de acesso a campos ponto (símbolo `.`). Note que, após a execução do primeiro comando de atribuição, `p1` armazenará o valor 118987623 referente ao código de um funcionário. Após a execução do segundo comando de atribuição, `p1` armazenará o valor 3876 referente ao RA de um estudante. Finalmente, `p1` receberá o valor 15015015015 após a execução da função `strcpy()`, o qual é referente ao campo CPF. Como a mesma porção de memória é compartilhada pela união `pessoa`, após uma atribuição a um dos campos, o valor atribuído anteriormente a qualquer outro campo é perdido. Assim, se após a função `strcpy()` for realizada a escrita de `p1.RA`, um resultado inconsistente será produzido.

```
...
union pessoa
{
    int RA;          // RA de um estudante
    long int codFunc; // código de um funcionário
    char CPF[12];    // CPF de uma pessoa
} p1;
...
p1.codFunc = 118987623;
...
p1.RA = 3876;
...
strcpy(p1.CPF, "15015015015");
...
```

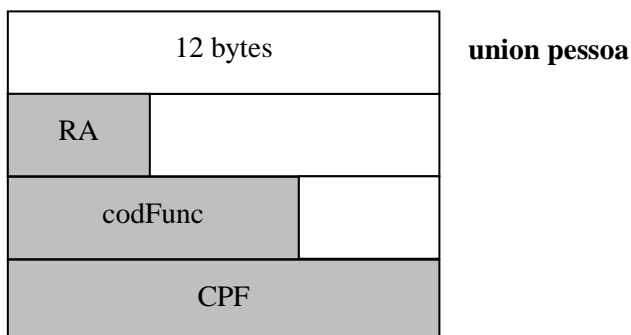


Figura 19. Esquema de compartilhamento de memória na união `pessoa`.

## Enumeração

Uma enumeração define um conjunto de constantes para representar valores que uma variável pode assumir. Cada constante é representada internamente por um número inteiro, sendo que a primeira constante recebe o valor 0, e as demais são incrementadas de 1 em 1. No trecho de programa a seguir, a enumeração tipo é declarada como o conjunto de constantes motorista (valor 0), secretario (valor 1), docente (valor 2) e outros (valor 3). O campo tipoFunc das variáveis funcionario1 e funcionario2 somente podem assumir as constantes definidas na enumeração. O trecho de programa também ilustra a atribuição de valores ao campo tipoFunc destas variáveis. Note que a atribuição é feita sem o uso de aspas duplas, pois as constantes representam valores inteiros e não strings.

```
...
enum tipo {motorista, secretario, docente, outros};
...
struct funcionario
{
    int codFunc;    // código do funcionário
    int idadeFunc; // idade do funcionário
    char sexoFunc; // sexo do funcionário, M (masculino) e F (feminino)
    float salFunc; // salário do funcionário
    enum tipo tipoFunc; // tipo do funcionário
} funcionario1, funcionario2;
...
funcionario1.tipoFunc = motorista;
funcionario2.tipoFunc = docente;
...
if (funcionario1.tipoFunc == funcionario2.tipoFunc)
{
    ...
}
...
```

### Uso de funções em expressões

Funções que retornam um valor, ou seja, funções que possuem um tipo, podem ser usadas em expressões. O trecho de programa abaixo ilustra o uso de 2 funções em um comando de atribuição. A parte direita deste comando é uma expressão. O corpo das funções é omitido no exemplo.

```
...  
x = f1(2,4) + f2(8);  
...  
int f1(int a, int b)  
{  
    // sequência de comandos  
}  
  
int f2(int c)  
{  
    // sequência de comandos  
}
```

Por outro lado, funções que não retornam um valor, ou seja, funções do tipo void, não podem ser usadas em expressões. O trecho de programa abaixo ilustra o uso incorreto de 2 funções void em um comando de atribuição.

```
...  
x = f1(2,4) + f2(8); // ERRO. As funções não retornam valor.  
...  
void f1(int a, int b)  
{  
    // sequência de comandos  
}  
  
void f2(int c)  
{  
    // sequência de comandos  
}
```

## Funções, declarações locais e declarações globais

Em um programa, o escopo de uma variável, ou seja, a abrangência da visibilidade da variável dentro do programa e, por conseguinte, o possível uso de seus valores no programa, depende do local que a variável foi declarada. Uma variável global, que é declarada externamente a função `main()` e também externamente a qualquer função adicional, pode ser usada em qualquer parte do programa. Em outras palavras, uma variável global pode ser usada dentro de qualquer função. Já uma variável declarada dentro de uma função possui escopo local. Assim, esta variável somente pode ser usada dentro do corpo da função e não é visível em outras partes do programa. O trecho de programa a seguir ilustra o uso de variáveis globais e locais e indica um erro no uso de variáveis locais.

```
...  
float valor1; // variável global visível em qualquer parte  
...  
void f1(int a, int b)  
{  
    int valor2;  
    valor2 = valor1 + 1; // CORRETO. valor1 possui escopo global  
    ...  
}  
  
void f2(int c)  
{  
    int valor3;  
    valor3 = valor2 - 1; // ERRO: valor2 possui escopo local a f1()  
    ...  
}  
...
```

Quando variáveis em diferentes escopos, por exemplo escopo global e local, possuem o mesmo nome, dentro de uma função é sempre escolhida a variável que possui o escopo local. O trecho de programa a seguir ilustra esta situação.



```

...
int valor = 60; // variável global visível em qualquer parte
...
void f1(int a, int b)
{
    int valor = 2;
    printf("valor: %d", valor); // será impresso o valor 2
    ...
}
...

```

## 4.5 Mapeamento de Algoritmo para Programa C

A seguir são apresentados os mapeamentos de 7 algoritmos para programas C, com ênfase em ponteiros, estruturas e funções.

### Algoritmo 7-1 (apostila de “Construção de Algoritmos”)

```

1      { dadas duas variáveis, uma inteira e outra literal, escrever os
2      endereços de memória na qual estão armazenadas }
3
4      algoritmo
5          declare
6              valorReal: real
7              valorLiteral: literal
8              endereçoReal: ↑ real    { endereços de variáveis reais }
9              endereçoLiteral: ↑ literal { endereços de variáveis literais }
10
11         { armazenamento dos endereços }
12         endereçoReal ← &valorReal
13         endereçoLiteral ← &valorLiteral
14
15         { escrita do resultado }
16         escreva("A variável real está no endereço:", endereçoReal)
17         escreva("A variável literal está no endereço:", endereçoLiteral)
18     fim-algoritmo

```

### Programa 5 (equivalente ao algoritmo 7-1)

```

1  /* dadas duas variáveis, uma inteira e outra literal, escrever os
2     endereços de memória na qual estão armazenadas */
3
4  #include <stdio.h>
5  #include <stdlib.h>
6
7  #define TAM 250
8
9  int main(int argc, char *argv[])
10 {
11     float valorReal;
12     char valorLiteral[TAM];
13
14     float *enderecoReal; // endereços de variáveis reais
15     char *enderecoLiteral; // endereços de variáveis literais
16
17     // armazenamento dos endereços
18     enderecoReal = &valorReal;
19     enderecoLiteral = &valorLiteral;
20
21     // escrita do resultado
22     printf("A variavel real esta no endereco: %p\n", enderecoReal);
23     printf("A variavel literal esta no endereco: %p\n", enderecoLiteral);
24
25     system("PAUSE");
26     return 0;
27 }

```

### Algoritmo 7-2 (apostila de “Construção de Algoritmos”)

```

1 {dada uma variável, modificar seu conteúdo usando apenas seu endereço}
2
3 algoritmo
4     declare valor: inteiro      { um inteiro comum }
5     ponteiro: ↑ inteiro { um ponteiro para inteiro }
6
7     { atribuição de um dado inicial }
8     valor ← 100
9
10    { determinação do endereço da variável }

```

```
11      ponteiro ← &valor
12
13      { uso do ponteiro para mexer na memória }
14      ↑ ponteiro ← 200
15
16      { verificação do valor armazenado em valor }
17      escreva(valor) { 200!!! }
18  fim-algoritmo
```

**Programa 6 (equivalente ao algoritmo 7-2)**

```
1 //dada uma variável, modificar seu conteúdo usando apenas seu endere-
ço
2
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  int main(int argc, char *argv[])
7  {
8      int valor;    // um inteiro comum
9      int *ponteiro; // um ponteiro para inteiro
10
11      // atribuição de um dado inicial
12      valor = 100;
13
14      // determinação do endereço da variável
15      ponteiro = &valor;
16
17      // uso do ponteiro para mexer na memória
18      *ponteiro = 200;
19
20      // verificação do valor armazenado em valor
21      printf("valor = %d\n", valor);
22
23      system("PAUSE");
24      return 0;
25  }
```

### Algoritmo 7-3 (apostila de “Construção de Algoritmos”)

```
1 { exemplo de modificação de variáveis diferentes usando o mesmo con-
junto
2     de comandos }
3
4     algoritmo
5         declare valor1, valor2: inteiro
6         i, númeroPassos: inteiro
7         ponteiro: ↑ inteiro
8
9         { atribuições iniciais }
10        valor1 ← 0
11        valor2 ← 0
12
13        { repetição }
14        leia(númeroPassos)
15        para i ← 1 até númeroPassos
16            { escolha da variável }
17            se i % 5 = 0 então
18                ponteiro ← &valor1
19            senão
20                ponteiro ← &valor2
21            fim-se
22
23            { comandos de manipulação da variável escolhida }
24            ponteiro ← ↑ ponteiro + i
25            se ↑ ponteiro % 2 = 1 então
26                ↑ ponteiro ← ↑ ponteiro - 1
27            fim-se
28        fim-para
29
30        { escritas }
31        escreva(valor1, valor2)
32    fim-algoritmo
```

**Programa 7 (equivalente ao algoritmo 7-3)**

```
1  /* exemplo de modificação de variáveis diferentes usando
2     o mesmo conjunto de comandos */
3
4  #include <stdio.h>
5  #include <stdlib.h>
6
7  int main(int argc, char *argv[])
8  {
9      int valor1, valor2;
10     int i, numeroPassos;
11     int *ponteiro;
12
13     // atribuições iniciais
14     valor1 = 0;
15     valor2 = 0;
16
17     // repetição
18
19     printf("Digite o numero de passos: ");
20     scanf("%d", &numeroPassos);
21
22     for (i = 1; i <= numeroPassos; i++)
23     {
24         // escolha da variável
25         if (i % 5 == 0)
26             ponteiro = &valor1;
27         else
28             ponteiro = &valor2;
29
30         // comandos de manipulação da variável escolhida
31
32         *ponteiro += i;
33
34         if (*ponteiro % 2 == 1)
35             (*ponteiro)—;
36     }
37
38     // escritas
39     printf("valor 1 = %d\n", valor1);
40     printf("valor 2 = %d\n", valor2);
41
```

```
42    system("PAUSE");
43    return 0;
44 }
```

### Algoritmo 8-1 (apostila de “Construção de Algoritmos”)

```
1  { determinar, em  $R^3$ , qual de dois pontos é o mais próximo da origem }
2
3  algoritmo
4      declare
5          ponto1, ponto2: registro
6              x, y, z: real
7          fim-registro
8          distância1, distância2: real
9
10     { leitura das coordenadas dos pontos }
11     leia(ponto1.x, ponto1.y, ponto1.z)
12     leia(ponto2.x, ponto2.y, ponto2.z)
13
14     { cálculo das distâncias }
15     distância1  $\leftarrow$  raiz(pot(ponto1.x, 2) + pot(ponto1.y, 2)
+16pot(ponto1.z, 2))
17     distância2  $\leftarrow$  raiz(pot(ponto2.x, 2) + pot(ponto2.y, 2) +
18pot(ponto2.z, 2))
19
20     { comparação e resultado }
21     se distância1 < distância2 então
22         escreva("O primeiro ponto é o mais próximo")
23     senão
24         se distância2 < distância1 então
25             escreva("O segundo ponto é o mais próximo")
26         senão
27             escreva("Os pontos equidistam da origem")
28         fim-se
29     fim-se
30 fim-algoritmo
```

**Programa 8 (equivalente ao algoritmo 8-1)**

```
1 // determinar, em R3, qual de dois pontos é o mais próximo da origem
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <math.h> // para usar função sqrt( )
6
7 int main(int argc, char *argv[])
8 {
9     struct ponto
10    {
11        float x, y, z;
12    };
13
14    struct ponto ponto1, ponto2; // declara ponto1 e ponto2 como pontos
15    float distancia1, distancia2;
16
17    // leitura das coordenadas dos pontos
18
19    printf("Digite a coordenada x de ponto 1: ");
20    scanf("%f", &ponto1.x);
21    printf("Digite a coordenada y de ponto 1: ");
22    scanf("%f", &ponto1.y);
23    printf("Digite a coordenada z de ponto 1: ");
24    scanf("%f", &ponto1.z);
25    printf("Digite a coordenada x de ponto 2: ");
26    scanf("%f", &ponto2.x);
27    printf("Digite a coordenada y de ponto 2: ");
28    scanf("%f", &ponto2.y);
29    printf("Digite a coordenada z de ponto 2: ");
30    scanf("%f", &ponto2.z);
31
32    // cálculo das distâncias
33
34    distancia1 = sqrt((ponto1.x * ponto1.x) +
35                      (ponto1.y * ponto1.y) +
36                      (ponto1.z * ponto1.z));
37
38    distancia2 = sqrt((ponto2.x * ponto2.x) +
39                      (ponto2.y * ponto2.y) +
40                      (ponto2.z * ponto2.z));
41
42    // comparação e resultado
```

```
43
44     if (distancia1 < distancia2)
45         printf("O primeiro ponto e o mais proximo\n");
46     else if (distancia2 < distancia1)
47         printf("O segundo ponto e o mais proximo\n");
48     else printf("Os pontos equidistam da origem\n");
49
50     system("PAUSE");
51     return 0;
52 }
```

### Algoritmo 8-3 (apostila de “Construção de Algoritmos”)

```
1 { dados o nome, o preço e o tipo dos vinhos (indicados aqui por “T” para
2   tinto, “B” para branco ou “R” para rosê, descrever o vinho mais caro;
3   não são considerados vinhos de preços iguais;
4   fim dos dados indicado por nome = “fim” }
5
6 algoritmo
7     tipo tVinho: registro
8         nome,
9         tipo: literal
10        preço: real
11        fim-registro
12
13 declare
14     vinho, vinhoCaro: tVinho
15
16 { repetição para leitura dos dados }
17 vinhoCaro.preço ← -1 {para forçar a troca na primeira vez }
18 faça
19     { dados }
20     leia(vinho.nome, vinho.preço, vinho.tipo)
21
22     { verificação do maior preço }
23 se vinho.preço > vinhoCaro.preço e vinho.nome <> “fim” então
24     vinhoCaro ← vinho { copia tudo }
25 fim-se
26 até vinho.nome = “fim”
27
28 { apresentação do resultado }
29 se vinhoCaro.preço = -1 então
```



```
29         escreva("Nenhum vinho foi apresentado.")
30     senão
31         escreva(vinhoCaro.nome, vinhoCaro.preço, vinhoCaro.tipo)
32     fim-se
33 fim-algoritmo
```

### Programa 9 (equivalente ao algoritmo 8-3)

```
1 /* dados o nome, o preço e o tipo dos vinhos (indicados aqui por "T" para
2    tinto, "B" para branco ou "R" para rosê, descrever o vinho mais caro;
3    não são considerados vinhos de preços iguais;
4    fim dos dados indicado por nome = "fim"
5    */
6
7 #include <stdio.h>
8 #include <stdlib.h>
9
10 int main(int argc, char *argv[])
11 {
12     struct tVinho
13     {
14         char nome[20];
15         char tipo;
16         float preco;
17     };
18
19     struct tVinho vinho, vinhoCaro;
20
21     // repetição para leitura dos dados
22
23     vinhoCaro.preco = -1; // para forçar a troca na primeira vez
24
25     do
26     {
27         // dados
28         printf("Digite o nome do vinho: ");
29         gets(vinho.nome);
30         printf("Digite o preço do vinho: ");
31         scanf("%f", &vinho.preco);
32         fflush(stdin);
33         printf("Digite o tipo do vinho: ");
34         scanf("%c", &vinho.tipo);
```

```

35    fflush(stdin);
36    printf("\n");
37
38    // verificação do maior preço
39    if ((vinho.preco > vinhoCaro.preco) &&
40        (strcmp(vinho.nome, "fim")!=0))
41        vinhoCaro = vinho;
42    } while (strcmp(vinho.nome, "fim")!=0);
43
44    // apresentação do resultado
45
46    if (vinhoCaro.preco == -1)
47        printf("Nenhum vinho foi apresentado.\n");
48    else
49    {
50        printf("Nome do vinho: %s\n", vinhoCaro.nome);
51        printf("Preço do vinho: %.2f\n", vinhoCaro.preco);
52        printf("Tipo do vinho: %c\n", vinhoCaro.tipo);
53    }
54
55    system("PAUSE");
56    return 0;
57    }

```

### Algoritmo 9-1 (apostila de “Construção de Algoritmos”)

```

1    { a partir de dois números racionais, calcular e apresentar a soma e a
2      multiplicação de ambos }
3
4    { definição do tipo }
5    tipo tRacional: registro
6
7        numerador, denominador: inteiro
8        fim-registro
9
10   procedimento simplifiqueRacional(var racional: tRacional)
11   { modifica um racional para que fique na forma mais simples }
12       declare valor1, valor2, resto: inteiro
13
14       { cálculo do MDC }
15       valor1 ← racional.numerador
16       valor2 ← racional.denominador
17       faça
18           resto ← valor1 % valor2
19           valor1 ← valor2

```

```

19      valor2 ← resto
20      até resto = 0      { resultado do MDC fica em valor1 }
21
22      { simplificação da razão }
23      racional.numerador ← racional.numerador/valor1
24      racional.denominador ← racional.denominador/valor1
25  fim-procedimento
26
27  algoritmo
28      declare número1, número2, resultado: tRacional
29
30      { obtenção dos números }
31      leia(número1.numerador, número1.denominador)
32      leia(número2.numerador, número2.denominador)
33
34      { cálculo da soma }
35  resultado.numerador ← número1.numerador * número2.denominador +
36      número1.denominador * número2.numerador
37  resultado.denominador ← número1.denominador * número2.denominador
38      simplifiqueRacional(resultado)
39
40      { escrita da soma }
41      escreva(resultado.numerador, "/", resultado.denominador)
42
43      { cálculo do produto }
44      resultado.numerador ← número1.numerador * número2.numerador
45  resultado.denominador ← número1.denominador * número2.denominador
46      simplifiqueRacional(resultado)
47
48      { escrita do produto }
49      escreva(resultado.numerador, "/", resultado.denominador)
50  fim-algoritmo

```

### Programa 10 (equivalente ao algoritmo 9-1)

```

1  /* a partir de dois números racionais,
2     calcular e apresentar a soma e a multiplicação de ambos */
3
4  #include <stdio.h>
5  #include <stdlib.h>
6
7  // definição do tipo de dados tRacional
8  typedef
9  struct
10 {

```

```

11     int numerador;
12     int denominador;
13 } tRacional;
14
15 // protótipos de funções
16 void simplifiqueRacional(tRacional *racional);
17
18 int main(int argc, char *argv[])
19 {
20     tRacional numero1, numero2, resultado;
21
22     // obtenção dos números }
23     printf("Digite o numerador do numero 1: ");
24     scanf("%d", &numero1.numerador);
25     printf("Digite o denominador do numero 1: ");
26     scanf("%d", &numero1.denominador);
27     printf("Digite o numerador do numero 2: ");
28     scanf("%d", &numero2.numerador);
29     printf("Digite o denominador do numero 2: ");
30     scanf("%d", &numero2.denominador);
31
32     // cálculo da soma
33     resultado.numerador = numero1.numerador * numero2.denominador
+
34         numero1.denominador * numero2.numerador;
35     resultado.denominador = numero1.denominador *
numero2.denominador;
36     simplifiqueRacional(&resultado);
37
38     // escrita da soma
39     printf("resultado da soma: %d / %d\n",
40         resultado.numerador, resultado.denominador);
41
42     // cálculo do produto
43     resultado.numerador = numero1.numerador * numero2.numerador;
44     resultado.denominador = numero1.denominador *
numero2.denominador;
45     simplifiqueRacional(&resultado);
46
47     // escrita do produto
48     printf("resultado do produto: %d / %d\n",
49         resultado.numerador, resultado.denominador);
50
51     system("PAUSE");

```

```

52     return 0;
53 }
54
55 void simplifiqueRacional(tRacional *racional)
56 {
57     // modifica um racional para que fique na forma mais simples
58     int valor1, valor2, resto;
59
60     // cálculo do MDC
61     valor1 = racional->denominador;
62     valor2 = racional->numerador;
63     do
64     { resto = valor1 % valor2;
65       valor1 = valor2;
66       valor2 = resto;
67     } while (resto != 0); // resultado do MDC fica em valor1
68
69     // simplificação da razão
70     racional->numerador = racional->numerador / valor1;
71     racional->denominador = racional->denominador / valor1;
72 }

```

### Algoritmo 9-2 (apostila de “Construção de Algoritmos”)

```

1  { algoritmo simples (e sem função específica) que ilustra o formato
2  e as consequências de sub-rotinas com passagem de parâmetros por
3  valor e por referência }
4
5  procedimento escrevaInteiroEFracionario(valor: real)
6  { escreve as partes inteira e fracionária de um valor }
7
8      escreva(“Parte inteira:”, int(valor))
9      valor ← valor – int(valor)
10     escreva(“Parte fracionária:”, valor)
11 fim-procedimento
12
13 procedimento transformeEmPositivo(var valor: real)
14 { transforma o valor de um número real em positivo }
15
16     se valor < 0 então
17         valor ← -valor { transforma em positivo }
18     fim-se
19 fim-procedimento

```

```
20
21 algoritmo
22     declare número: real
23
24     { o valor “especial” 999 indica o término }
25     faça
26         leia(número)
27
28         escrevaInteiroEFracionario(número)
29
30         escreva(“O módulo é:”)
31         transformeEmPositivo(número)
32         escreva(número)
33     enquanto número ≠ 999
34 fim-algoritmo
```

### Programa 11 (equivalente ao algoritmo 9-2)

```
1  /* algoritmo simples (e sem função específica) que ilustra
2   o formato e as consequências de funções com passagem de
3   parâmetros por valor e por referência (i.e., ponteiros) */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <math.h> // uso da função modf( )
8
9  // protótipos de funções
10 void escrevaInteiroEFracionario(float valor);
11 void transformeEmPositivo(float *valor);
12
13 int main(int argc, char *argv[])
14 {
15     float numero;
16
17     // o valor “especial” 999 indica o término
18     do
19     { printf(“Digite um numero: “);
20       scanf(“%f”, &numero);
21       fflush(stdin);
22       escrevaInteiroEFracionario(numero);
23       printf(“O modulo eh: “);
24       transformeEmPositivo(&numero);
25       printf(“%.2f\n\n”, numero);
```

```
26     }
27     while (numero != 999);
28
29     system("PAUSE");
30     return 0;
31 }
32
33 void escrevaInteiroEFracionario(float valor)
34 {
35     double parte_fracionaria, parte_inteira;
36
37     /* obtenção da parte inteira e da parte fracionaria
38     do parâmetro formal valor. Para isto, usa-se modf( ).
39     A função modf( ) retorna a parte fracionaria
40     de seu primeiro parâmetro. O segundo parâmetro recebe
41     a parte inteira de seu primeiro parâmetro */
42     parte_fracionaria = modf(valor, &parte_inteira);
43
44     // escreve as partes inteira e fracionária de um valor
45     printf("Parte inteira: %.2f\n", parte_inteira);
46     printf("Parte fracionaria: %.2f\n", parte_fracionaria);
47 }
48
49 void transformeEmPositivo(float *valor)
50 {
51     // transforma o valor de um número real em positivo
52     if ((*valor) < 0)
53         *valor = -(*valor);
54 }
```





## Unidade 5

---

Vetores, Matrizes e Arranjos de Registros

---



Este capítulo irá tratar de arranjos. Na seção 5.1 será apresentado o conceito de arranjo unidimensional. Na seção 5.2 é discutido o conceito de arranjo bidimensional. Na seção 5.3 será apresentado o uso de arranjos de estruturas.

Neste capítulo, você aprenderá sobre os seguintes **conceitos** relacionados a arranjos:

arranjo, elementos de arranjos, índices de arranjos, acesso a elementos de arranjos, espaço alocado para arranjos, vetor, matriz, relacionamento entre os conceitos de arranjos e ponteiros, arranjos de estruturas, inicialização de arranjos na declaração, arranjos não dimensionados, arranjos multidimensionais, indexação de ponteiros, e vetores e matrizes como argumentos de funções.

## 5.1 Arranjos Unidimensionais

Um **arranjo unidimensional**, também chamado de **vetor** ou simplesmente **arranjo**, é uma estrutura que permite armazenar um conjunto de elementos de um mesmo tipo de dados. Os **elementos** de um arranjo são armazenados contiguamente na memória, ou seja, ocupam endereços de memória adjacentes. A Figura 1 ilustra um arranjo de inteiros, contendo 5 elementos. Assumindo que o tipo de dados `int` ocupa 4 bytes, o primeiro elemento está armazenado no endereço 484, o segundo elemento está armazenado no endereço 488, e assim sucessivamente.

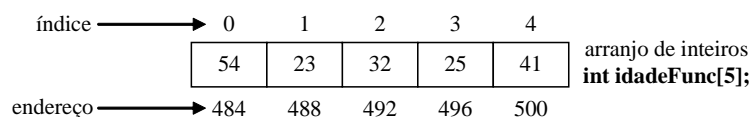


Figura 1. Exemplo de uma variável do tipo arranjo unidimensional.

Um arranjo pode ser utilizado em substituição ao uso de diversas variáveis de um mesmo tipo. Por exemplo, o arranjo de inteiros da Figura 1 pode armazenar as idades de 5 funcionários.

Desta forma, ao invés de declarar `idadeFunc1`, `idadeFunc2`, `idadeFunc3`, `idadeFunc4` e `idadeFunc5`, é necessário declarar uma única **variável arranjo**, chamada de `idadeFunc`. Cada elemento de `idadeFunc` refere-se à idade de um funcionário diferente.

A **declaração** de uma variável arranjo é realizada de acordo com o seguinte formato: **tipo nome\_variável[tamanho]**. Toda declaração de arranjo inicia-se com a definição do tipo de dados dos elementos do arranjo (i.e., tipo). Na seqüência, é especificado o nome da variável arranjo (i.e., nome\_variável). O número de elementos que o arranjo pode armazenar é indicado entre colchetes. Portanto, tamanho é um valor numérico que especifica este número de elementos. A declaração é finalizada com ponto-e-vírgula. Na Figura 1, a variável `idadeFunc` é declarada como um arranjo de inteiros de tamanho 5 (i.e., **`int idadeFunc[5];`**).

A declaração de uma variável arranjo faz com que seja alocada uma área de memória de tamanho suficiente para armazenar todos os seus elementos. O espaço de memória alocado para um arranjo, em bytes, é determinado pela fórmula: **espaço alocado = tamanho \* sizeof(tipo)**. A função **`sizeof( )`** retorna a quantidade de bytes que um tipo de dados ou uma variável ocupa. Na Figura 1, o espaço alocado para a variável arranjo `idadeFunc` é 20 bytes, ou seja,  $5 * 4$ .

Um arranjo, além de agrupar um conjunto de variáveis de um mesmo tipo, facilita o acesso aos dados dos seus diferentes elementos. O **acesso a um elemento** de um arranjo é feito usando um **índice**. Este índice é um valor inteiro que indica a posição do elemento no arranjo. Conforme ilustrado na Figura 1, as posições variam de 0 até o valor do número de elementos menos 1. Portanto, para o arranjo de 5 elementos `idadeFunc`, os seus elementos ocupam as posições 0 a 4. Cada elemento é acessado conforme o seguinte formato: **nome\_variável[índice]**. Por exemplo, para acessar o primeiro elemento de `idadeFunc`, usa-se `idadeFunc[0]`. O valor retornado é 54. Já para acessar o terceiro elemento de `idadeFunc`, usa-se `idadeFunc[2]`. O valor retornado é 32. Para acessar o último elemento de `idadeFunc`, usa-se a posição `tamanho-1`, ou seja, `idadeFunc[4]`. O valor retornado é 41. Um cuidado com o uso de índices em arranjos é que a linguagem C não realiza a verificação de

limites. Por exemplo, o uso de `idadeFunc[18]` não gera erro durante a compilação do programa. Porém, na execução do programa, haverá um acesso a uma posição de memória indevida e, por conseguinte, ocorrerá um erro que possivelmente abortará a execução do programa. Portanto, a verificação de limites deve ser realizada pelo programador.

O Programa 1 a seguir exemplifica o uso de arranjos. Este programa realiza a leitura dos dados de 5 funcionários (i.e., dados sobre o código, a idade e o salário de cada funcionário) e calcula a idade média e o salário médio dos funcionários. O total de funcionários, portanto, é conhecido a priori. No final, o programa mostra, além dos dados calculados (i.e., a idade média e o salário médio), o total de funcionários e os dados de cada um dos funcionários.

### Programa 1

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define TAMANHO 5
5
6  int main(int argc, char *argv[])
7  {
8      int codFunc[TAMANHO]; // código dos funcionários
9      int idadeFunc[TAMANHO]; // idade dos funcionários
10     float salFunc[TAMANHO]; // salário dos funcionários
11
12     const int totalFunc = TAMANHO; // total de funcionários
13     int somaIdade = 0; // soma das idades dos funcionários
14     float somaSalario = 0; // soma dos salários dos funcionários
15
16     float idadeMedia; // idade média dos funcionários
17     float salarioMedio; // salário médio dos funcionários
18
19     int i; // contador temporário do comando for
20
21     // leitura dos dados dos funcionários
22
23     for (i = 0; i < TAMANHO; i++)
24     {
25         printf("A seguir, entre com todos os dados do funcionario %d.\n", i+1);
```

```
26     printf("Digite o codigo: ");
27     scanf("%d", &codFunc[i]);
28     printf("Digite a idade: ");
29     scanf("%d", &idadeFunc[i]);
30     printf("Digite o salario (R$): ");
31     scanf("%f", &salFunc[i]);
32     printf("\n");
33     fflush(stdin);
34 };
35
36 // cálculo intermediários
37
38 for (i = 0; i < TAMANHO; i++)
39 {
40     somaIdade += idadeFunc[i];
41     somaSalario += salFunc[i];
42 }
43
44 // cálculo da idade média e do salário médio dos funcionários
45
46 idadeMedia = (float) somaIdade / totalFunc;
47 salarioMedio = somaSalario / totalFunc;
48
49 // saída dos dados calculados
50
51 printf("Os dados calculados sao:\n");
52 printf("Total de funcionarios = %d\n", totalFunc);
53 printf("Idade media %.2f anos\n", idadeMedia);
54 printf("Salario medio %.2f reais\n", salarioMedio);
55 printf("\n");
56
57 // escrita dos dados dos funcionários
58
59 for (i = 0; i < TAMANHO; i++)
60 {
61     printf("funcionario %i\n", i+1);
62     printf("Codigo: %d\n", codFunc[i]);
63     printf("Idade: %d\n", idadeFunc[i]);
64     printf("Salario: %.2f\n", salFunc[i]);
65 };
66
67 // finalização do programa principal
68
69 system("PAUSE");
70 return 0;
71 }
```

No Programa 1, o tamanho dos arranjos é definido pela constante TAMANHO na linha 4. As linhas 8 a 10 declaram 3 arranjos, os quais armazenam respectivamente o código, a idade e o salário dos funcionários. Enquanto os dois primeiros arranjos possuem elementos do tipo int, o terceiro arranjo armazena elementos do tipo float.

A leitura dos dados dos funcionários é realizada nas linhas 23 a 34 com o auxílio do comando for. Este comando controla o índice dos arranjos, variando de 0 até o TAMANHO – 1, com incremento de 1. Assim, a cada execução do laço, os dados de um funcionário são lidos nas linhas 27, 29 e 31 e são armazenados em suas posições correspondentes em cada um dos arranjos. Note que o acesso ao endereço de um elemento de um arranjo é similar ao acesso ao endereço de uma variável simples. Por exemplo, `&codFunc[i]` corresponde ao endereço do elemento na posição `i` do arranjo `codFunc`. O comando for facilita o acesso a todos os elementos de um arranjo e, portanto, deve ser utilizado neste caso.

Outro uso do comando for para ler todos os dados dos elementos de um arranjo é realizado nas linhas 38 a 42. Note que nas linhas 40 e 41, o valor de um elemento de um arranjo é usado em uma expressão. Por exemplo, `salFunc[i]` realiza o acesso ao valor do elemento na posição `i` do arranjo `salFunc`.

O uso dos arranjos `codFunc`, `idadeFunc` e `salFunc` permite armazenar os dados de todos os 5 funcionários durante todo o programa. Como resultado, é possível imprimir estes dados com o auxílio do comando for nas linhas 59 a 65. Note que na linha 61 o número do funcionário cujos dados estão sendo exibidos na tela é acrescido de 1 unidade ao valor da variável de controle de laço. Isto é necessário porque o índice de um arranjo inicia-se em 0, enquanto que o número do funcionário inicia-se em 1. As Figuras Figura 2 e Figura 3 ilustram um exemplo de execução do Programa 1.

```
F:\Usuários\Ricardo\Programação\Dev-Cpp\LabProg\Capitulo 05\Programa 02\projeto02.exe
A seguir, entre com todos os dados do funcionario 1.
Digite o codigo: 100
Digite a idade: 54
Digite o salario (R$): 1000.00

A seguir, entre com todos os dados do funcionario 2.
Digite o codigo: 101
Digite a idade: 23
Digite o salario (R$): 1200.00

A seguir, entre com todos os dados do funcionario 3.
Digite o codigo: 102
Digite a idade: 32
Digite o salario (R$): 1400.00

A seguir, entre com todos os dados do funcionario 4.
Digite o codigo: 103
Digite a idade: 25
Digite o salario (R$): 800.00

A seguir, entre com todos os dados do funcionario 5.
Digite o codigo: 104
Digite a idade: 41
Digite o salario (R$): 600.00
```

Figura 2. Exemplo de execução do Programa 1: entrada de dados.

```
F:\Usuários\Ricardo\Programação\Dev-Cpp\LabProg\Capitulo 05\Programa 02\projeto02.exe
Os dados calculados são:
Total de funcionarios = 5
Idade media 35.00 anos
Salario medio 1000.00 reais

funcionario 1
Codigo: 100
Idade: 54
Salario: 1000.00

funcionario 2
Codigo: 101
Idade: 23
Salario: 1200.00

funcionario 3
Codigo: 102
Idade: 32
Salario: 1400.00

funcionario 4
Codigo: 103
Idade: 25
Salario: 800.00

funcionario 5
Codigo: 104
Idade: 41
Salario: 600.00

Pressione qualquer tecla para continuar. . .
```

Figura 3. Exemplo de execução do Programa 1: resultados.



Um aspecto muito importante na linguagem C é o **relacionamento entre os conceitos de arranjos e ponteiros**. O nome de um arranjo sem o seu índice representa o endereço do primeiro elemento do arranjo. Este endereço pode ser atribuído a um ponteiro, o qual pode ser usado para acessar os elementos do arranjo. Uma vez ocorrida esta atribuição, a aritmética de ponteiros permite o acesso a qualquer um dos elementos. O Programa 2 adapta o Programa 1 para acessar os elementos dos arranjos `codFunc`, `idadeFunc` e `salFunc` usando ponteiros.

### Programa 2

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define TAMANHO 5
5
6  int main(int argc, char *argv[])
7  {
8      int codFunc[TAMANHO]; // código dos funcionários
9      int idadeFunc[TAMANHO]; // idade dos funcionários
10     float salFunc[TAMANHO]; // salário dos funcionários
11
12     int *pont_codFunc; // ponteiro para o arranjo codFunc
13     int *pont_idadeFunc; // ponteiro para o arranjo idadeFunc
14     float *pont_salFunc; // ponteiro para o arranjo salFunc
15
16     const int totalFunc = TAMANHO; // total de funcionários
17     int somaIdade = 0; // soma das idades dos funcionários
18     float somaSalario = 0; // soma dos salários dos funcionários
19
20     float idadeMedia; // idade média dos funcionários
21     float salarioMedio; // salário médio dos funcionários
22
23     int i; // contador temporário do comando for
24
25     // inicializar os ponteiros
26     pont_codFunc = codFunc; // pont_codFunc = &codFunc[0]
27     pont_idadeFunc = idadeFunc; // pont_idadeFunc = &idadeFunc[0]
28     pont_salFunc = salFunc; // pont_salFunc = &salFunc[0]
29
30     // leitura dos dados dos funcionários
```

```
31
32     for (i = 0; i < TAMANHO; i++)
33     {
34         printf("A seguir, entre com todos os dados do funcionario %d.\n", i+1);
35         printf("Digite o codigo: ");
36         scanf("%d", &pont_codFunc++);
37         printf("Digite a idade: ");
38         scanf("%d", &pont_idadeFunc++);
39         printf("Digite o salario (R$): ");
40         scanf("%f", &pont_salFunc++);
41         printf("\n");
42         fflush(stdin);
43     };
44
45     // inicializar os ponteiros
46     pont_codFunc = codFunc; // pont_codFunc = &codFunc[0]
47     pont_idadeFunc = idadeFunc; // pont_idadeFunc = &idadeFunc[0]
48     pont_salFunc = salFunc; // pont_salFunc = &salFunc[0]
49
50     // cálculo intermediários
51
52     for (i = 0; i < TAMANHO; i++)
53     {
54         somaIdade += *pont_idadeFunc;
55         somaSalario += *pont_salFunc;
56         pont_idadeFunc++;
57         pont_salFunc++;
58     }
59
60     // cálculo da idade média e do salário médio dos funcionários
61
62     idadeMedia = (float) somaIdade / totalFunc;
63     salarioMedio = somaSalario / totalFunc;
64
65     // saída dos dados calculados
66
67     printf("Os dados calculados sao:\n");
68     printf("Total de funcionarios = %d\n", totalFunc);
69     printf("Idade media %.2f anos\n", idadeMedia);
70     printf("Salario medio %.2f reais\n", salarioMedio);
71     printf("\n");
72
73     // inicializar os ponteiros
74     pont_codFunc = codFunc; // pont_codFunc = &codFunc[0]
```

```
75     pont_idadeFunc = idadeFunc; // pont_idadeFunc = &idadeFunc[0]
76     pont_salFunc = salFunc;    // pont_salFunc = &salFunc[0]
77
78     // escrita dos dados dos funcionários
79
80     for (i = 0; i < TAMANHO; i++)
81     {
82         printf("funcionario %i\n", i+1);
83         printf("Codigo: %d\n", *pont_codFunc);
84         printf("Idade: %d\n", *pont_idadeFunc);
85         printf("Salario: %.2f\n", *pont_salFunc);
86         pont_codFunc++;
87         pont_idadeFunc++;
88         pont_salFunc++;
89     };
90
91     // finalização do programa principal
92
93     system("PAUSE");
94     return 0;
95 }
```

No Programa 2, os ponteiros para os arranjos codFunc, idadeFunc e salFunc são declarados nas linhas 12 a 14. Estes ponteiros recebem os endereços iniciais destes arranjos nas linhas 26 a 28. Por exemplo, na linha 26 o comando `pont_codFunc = codFunc` atribui ao ponteiro `pont_codFunc` o endereço do primeiro elemento do arranjo `codFunc` (`&codFunc[0]`).

O acesso aos elementos dos arranjos é realizado por meio de ponteiros. Por exemplo, na linha 36 é passado para a função `scanf( )` o valor de `pont_codFunc`, que corresponde ao endereço do elemento corrente no arranjo `codFunc`. Note que não é utilizado o operador de ponteiro `&`, desde que `pont_codFunc` já armazena um endereço. Na primeira vez que o laço das linhas 34 a 43 é executado, `pont_codFunc` contém o endereço de `codFunc[0]`. Isto significa que o valor lido pela função `scanf( )` será armazenado neste elemento, ou seja, será armazenado em `codFunc[0]`. Logo após a leitura deste elemento, o ponteiro `pont_codFunc` é incrementado, de forma que ele passa a armazenar o endereço de `codFunc[1]`, que é o próximo elemento do arranjo `codFunc`. O incremento, con-

forme já explicado na aritmética de ponteiros, é realizado de acordo com o tipo base, no caso, o tipo `int`.

Uma vez que os ponteiros `pont_codFunc`, `pont_idadeFunc` e `pont_salFunc` são incrementados a cada execução do laço nas linhas 34 a 43, ao final da execução do comando `for` estes ponteiros não possuem mais o endereço do primeiro elemento dos arranjos `codFunc`, `idadeFunc` e `salFunc`, respectivamente. Na verdade, estes ponteiros armazenam endereços fora dos limites dos arranjos (i.e., endereços após o endereço do último elemento de cada arranjo). Assim, antes dos ponteiros serem usados novamente, eles devem ser inicializados, conforme efetuado nas linhas 46 a 48 e 74 a 76.

O acesso ao conteúdo dos elementos dos arranjos também é realizado usando ponteiros. Uma vez que o ponteiro possui o endereço do elemento corrente, o acesso aos dados deste elemento é feito por meio do operador de ponteiros `*`. A forma de uso deste operador é a mesma que a discutida no Capítulo 4. Exemplos de uso do operador de ponteiros `*` podem ser encontrados nas linhas 54 e 55 e nas linhas 83 a 85. Na linha 54, assumindo que `i` possui o valor 1, `pont_idadeFunc` armazena o endereço do segundo elemento do arranjo `idadeFunc`. Assim, `*pont_idadeFunc` retorna o conteúdo de `idadeFunc[1]`.

## 5.2 Arranjos Bidimensionais

Um **arranjo bidimensional**, comumente chamado de **matriz**, é uma estrutura de dados que possui duas dimensões. A primeira dimensão representa as linhas da matriz enquanto a segunda dimensão representa as colunas da matriz. Por isso, um arranjo bidimensional também é chamado de matriz linha-coluna. A Figura 4 ilustra uma matriz com 4 linhas e 6 colunas. Cada linha refere-se a um funcionário diferente. Já cada coluna refere-se a um salário mensal de um funcionário, nos meses de janeiro a junho (i.e., primeira coluna indica o mês de janeiro, segunda coluna indica o mês de fevereiro, e assim sucessivamente).

		colunas					
		0	1	2	3	4	5
linhas	0	1000.00	1000.00	1500.47	1500.47	1500.47	1600.00
	1	1300.00	1300.00	1300.00	1300.00	1300.00	1300.00
	2	2000.00	2000.00	2670.27	2670.27	2670.27	2670.27
	3	1300.00	1300.00	1300.00	1300.00	1300.00	1300.00

matriz de números do tipo float: **float salFunc[4][6];**

Figura 4. Matriz salFunc.

A **declaração** de uma variável matriz é realizada de acordo com o seguinte formato: **tipo nome\_variável[tamanho\_dimensão1][tamanho\_dimensão2];**. Toda declaração de matriz inicia-se com a definição do tipo de dados dos seus elementos. Portanto, todos os elementos de uma matriz são do mesmo tipo de dados. Na sequência, é especificado o nome da variável matriz. A dimensão 1 da variável matriz é representada pelo primeiro par de colchetes. Dentro destes colchetes, deve ser especificado o seu número de linhas. A dimensão 2, por sua vez, é representada pelo segundo par de colchetes. Dentro destes colchetes, deve ser especificado o número de colunas da variável matriz. A declaração é finalizada com ponto-e-vírgula. Na Figura 4, a variável salFunc é declarada como uma matriz de números do tipo float com 4 linhas e 6 colunas (i.e., **float salFunc[4][6];**).

A declaração de uma variável matriz faz com que seja alocada uma área de memória de tamanho suficiente para armazenar todos os seus elementos. O espaço de memória alocado para uma matriz, em bytes, é determinado pela fórmula: espaço alocado = tamanho da dimensão 1 \* tamanho da dimensão 2 \* sizeof(tipo). Assumindo que o tipo de dados float ocupa 8 bytes, o espaço alocado para a variável matriz salFunc da Figura 4 é 192 bytes, ou seja,  $4 * 6 * 8$ .

O **acesso a um elemento** de uma matriz é feito usando 2 índices, um índice para as linhas e outro índice para as colunas. Para acessar o elemento armazenado na linha  $i$ , coluna  $j$  da matriz, usa-se o seguinte formato: **nome\_variável[i][j]**. Da mesma forma que para vetores, os valores do índice das linhas variam de 0 até o valor do número de linhas menos 1. Similarmente, os valores do índice das colunas variam de 0 até o valor do número de colunas menos 1. A Figura 5 ilustra a representação dos elementos da matriz `salFunc` em termos de seus índices. Note que os elementos variam de `salFunc[0][0]` a `salFunc[3][5]`. Assim, na Figura 4, para acessar o elemento da primeira linha e primeira coluna de `salFunc`, usa-se `salFunc[0][0]`. O valor retornado é 1000.00. Para acessar o elemento da primeira linha e quarta coluna de `salFunc`, usa-se `salFunc[0][3]`. O valor retornado é 1500.47. Para acessar o elemento na terceira linha e sexta coluna de `salFunc`, usa-se `salFunc[2][5]`. O valor retornado é 2670.27. Para acessar o último elemento, usa-se `salFunc[3][5]`. O valor retornado é 1300.00. Vale lembrar que a linguagem C não verifica os limites de índices em matrizes. Portanto, o uso de `salFunc[10][18]` vai gerar um erro em tempo de execução, e não em tempo de compilação.

		colunas					
		0	1	2	3	4	5
linhas	0	[0][0]	[0][1]	[0][2]	[0][3]	[0][4]	[0][5]
	1	[1][0]	[1][1]	[1][2]	[1][3]	[1][4]	[1][5]
	2	[2][0]	[2][1]	[2][2]	[2][3]	[2][4]	[2][5]
	3	[3][0]	[3][1]	[3][2]	[3][3]	[3][4]	[3][5]

matriz de números do tipo float: **float salFunc[4][6];**

Figura 5. Representação dos elementos de uma matriz em termos de seus índices.

Os **elementos** de uma matriz são armazenados contigualmente na memória, ou seja, ocupam endereços de memória adjacentes. Estes elementos são organizados da seguinte forma. Inicialmente são armazenados todos os elementos da primeira linha da

matriz. Em seguida, são armazenados todos os elementos da segunda linha da matriz, e assim sucessivamente. A Figura 6 ilustra o armazenamento em memória de parte da matriz salFunc da Figura 4. Assumindo que o tipo de dados float ocupa 8 bytes, o primeiro elemento de salFunc (salFunc[0][0]) está armazenado no endereço 484, o segundo elemento (salFunc[0][1]) está armazenado no endereço 492, o terceiro elemento (salFunc[0][2]) está armazenado no endereço 500, e assim sucessivamente de 8 em 8 bytes. O último elemento da primeira linha (salFunc[0][5]) está armazenado no endereço 524, ao passo que o primeiro elemento da segunda linha (salFunc[1][0]) está armazenado no endereço 532. Note que o primeiro elemento da segunda linha é o sétimo elemento de salFunc.

índices							
[0,0]		[0,5]	[1,0]		[1,5]	[2,0]	
1000.00	...	1600.00	1300.00	...	1300.00	2000.00	...
endereço							
484	...	524	532	...	580	588	...

Figura 6. Representação física da matriz salFunc.

O Programa 3 a seguir exemplifica o uso de matriz. Este programa realiza a leitura de 6 salários para 4 funcionários (i.e., salários dos meses de janeiro a junho de cada funcionário). Primeiramente, é calculado o salário médio de cada funcionários nos 6 meses. Em seguida, é realizado um aumento salarial de 5% no salário do mês de junho para todos os funcionários. O programa mostra, além do salário médio de cada funcionário, cada um dos salários dos 4 funcionários.

### Programa 3

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define LINHAS 4 // número de linhas da matriz salFunc
5  #define COLUNAS 6 // número de colunas da matriz salFunc
6
```

```
7  int main(int argc, char *argv[])
8  {
9      float salFunc[LINHAS][COLUNAS]; // salário dos funcionários
10         // nos meses de janeiro a junho
11     int i, j; // índices para salFunc
12
13     float somaSalario; // soma dos salários de cada funcionário
14     float salarioMedio; // salário médio de um funcionário
15
16     // leitura dos dados dos funcionários
17
18     for (i = 0; i < LINHAS; i++)
19     {
20         printf("Entre com os salarios do funcionario %d.\n", i+1);
21         for (j = 0; j < COLUNAS; j++)
22         {
23             printf("Digite o salario (R$) do mes %d: ", j+1);
24             scanf("%f", &salFunc[i][j]);
25         }
26         printf("\n");
27     }
28
29     // cálculo do salário médio por funcionário
30
31     for (i = 0; i < LINHAS; i++)
32     {
33         printf("O salario medio do funcionario %d eh: ", i+1);
34         somaSalario = 0;
35         for (j = 0; j < COLUNAS; j++)
36             somaSalario += salFunc[i][j];
37         salarioMedio = somaSalario / COLUNAS;
38         printf("%.2f\n", salarioMedio);
39     }
40     printf("\n");
41
42     // aumento salarial para o mês de junho
43     // aumento de 5% para todos os funcionários
44
45     for (i = 0; i < LINHAS; i++)
46         salFunc[i][5] *= 1.05;
47
48     // escrita dos dados dos funcionários
49
50     for (i = 0; i < LINHAS; i++)
51     {
```



```
52     printf("Salarios do funcionario %d.\n", i+1);
53     for (j = 0; j < COLUNAS; j++)
54         printf("Salario (R$) do mes %d: %.2f\n", j+1, salFunc[i][j]);
55     printf("\n");
56 }
57
58 // finalização do programa principal
59
60 system("PAUSE");
61 return 0;
62 }
```

No Programa 3, as linhas 4 e 5 definem constantes que indicam a quantidade de linhas e colunas da matriz de salários. Esta matriz de salários, chamada de `salFunc`, é declarada na linha 9, como uma matriz cujos elementos são números do tipo `float`. A linha 11 declara os índices que percorrerão as linhas e colunas de `salFunc`.

Nas linhas 18 a 27 é realizada a leitura dos dados dos funcionários. Desde que uma matriz possui duas dimensões, é necessário o uso de dois comandos de repetição “for” aninhados. O primeiro comando for percorre a matriz em termos de suas linhas usando o índice `i`, enquanto o segundo comando for (i.e., comando for mais interno) percorre a matriz em termos de suas colunas usando o índice `j`. O acesso a cada elemento de `salFunc` é realizado na linha 24 usando `&salFunc[i][j]`. O uso de dois comandos for aninhados para percorrer todos os elementos da matriz `salFunc` também pode ser encontrado nas linhas 31 a 39 e nas linhas 50 a 56.

Nem sempre é necessário o uso de dois comandos “for” aninhados para manipular os elementos de uma matriz. Nas linhas 45 e 46, um único comando for é usado para percorrer todas as linhas de `salFunc`. Isto é possível porque o aumento salarial é aplicado apenas para o mês de junho, ou seja, para a coluna 5. As Figuras Figura 7 e Figura 8 ilustram um exemplo de execução do Programa 3.

```
F:\Usuários\Ricardo\Programação\Dev-Cpp\LabProg\Capítulo 05\Progra...
Entre com os salarios do funcionario 1.
Digite o salario (R$) do mes 1: 1000
Digite o salario (R$) do mes 2: 1000
Digite o salario (R$) do mes 3: 1500.47
Digite o salario (R$) do mes 4: 1500.47
Digite o salario (R$) do mes 5: 1500.47
Digite o salario (R$) do mes 6: 1600

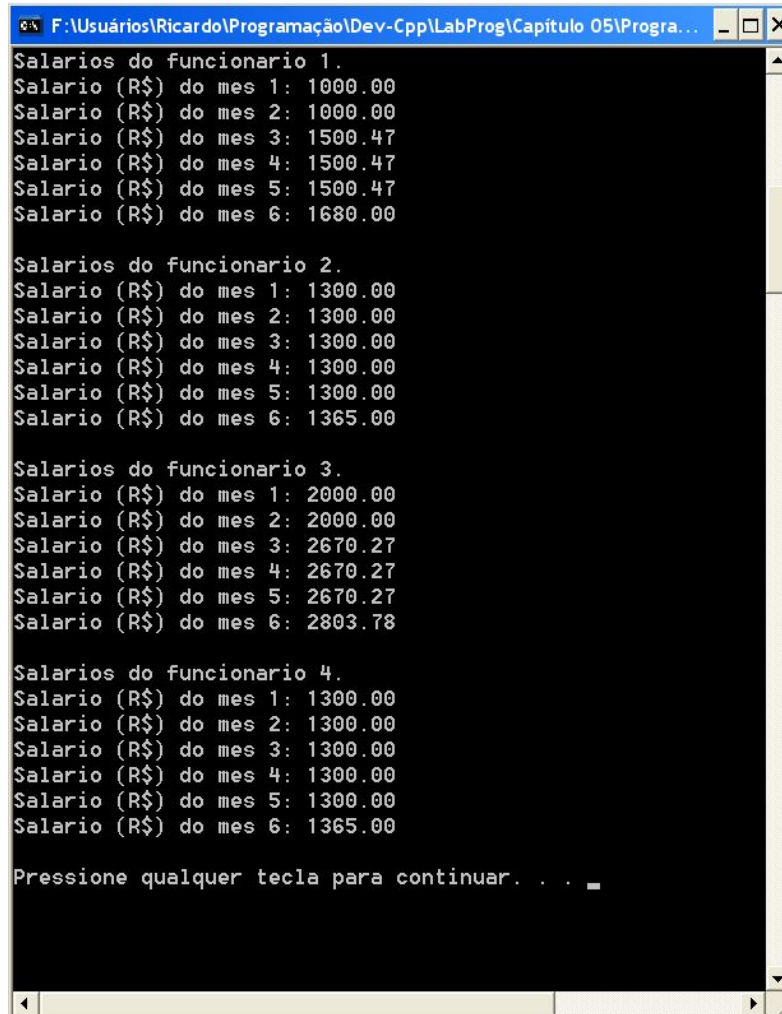
Entre com os salarios do funcionario 2.
Digite o salario (R$) do mes 1: 1300
Digite o salario (R$) do mes 2: 1300
Digite o salario (R$) do mes 3: 1300
Digite o salario (R$) do mes 4: 1300
Digite o salario (R$) do mes 5: 1300
Digite o salario (R$) do mes 6: 1300

Entre com os salarios do funcionario 3.
Digite o salario (R$) do mes 1: 2000
Digite o salario (R$) do mes 2: 2000
Digite o salario (R$) do mes 3: 2670.27
Digite o salario (R$) do mes 4: 2670.27
Digite o salario (R$) do mes 5: 2670.27
Digite o salario (R$) do mes 6: 2670.27

Entre com os salarios do funcionario 4.
Digite o salario (R$) do mes 1: 1300
Digite o salario (R$) do mes 2: 1300
Digite o salario (R$) do mes 3: 1300
Digite o salario (R$) do mes 4: 1300
Digite o salario (R$) do mes 5: 1300
Digite o salario (R$) do mes 6: 1300

O salario medio do funcionario 1 eh: 1350.23
O salario medio do funcionario 2 eh: 1300.00
O salario medio do funcionario 3 eh: 2446.85
O salario medio do funcionario 4 eh: 1300.00
```

Figura 7. Exemplo de execução do Programa 3 (parte 1).



```
F:\Usuários\Ricardo\Programação\Dev-Cpp\LabProg\Capítulo 05\Progra...
Salarios do funcionario 1.
Salario (R$) do mes 1: 1000.00
Salario (R$) do mes 2: 1000.00
Salario (R$) do mes 3: 1500.47
Salario (R$) do mes 4: 1500.47
Salario (R$) do mes 5: 1500.47
Salario (R$) do mes 6: 1680.00

Salarios do funcionario 2.
Salario (R$) do mes 1: 1300.00
Salario (R$) do mes 2: 1300.00
Salario (R$) do mes 3: 1300.00
Salario (R$) do mes 4: 1300.00
Salario (R$) do mes 5: 1300.00
Salario (R$) do mes 6: 1365.00

Salarios do funcionario 3.
Salario (R$) do mes 1: 2000.00
Salario (R$) do mes 2: 2000.00
Salario (R$) do mes 3: 2670.27
Salario (R$) do mes 4: 2670.27
Salario (R$) do mes 5: 2670.27
Salario (R$) do mes 6: 2803.78

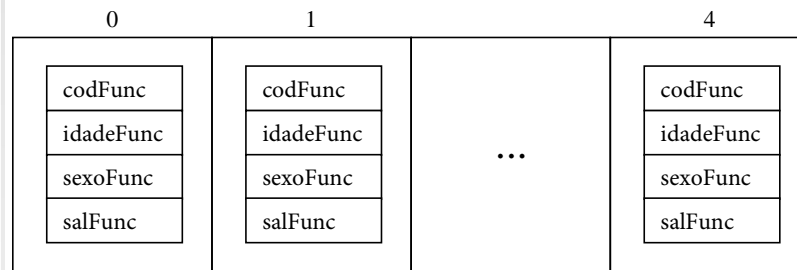
Salarios do funcionario 4.
Salario (R$) do mes 1: 1300.00
Salario (R$) do mes 2: 1300.00
Salario (R$) do mes 3: 1300.00
Salario (R$) do mes 4: 1300.00
Salario (R$) do mes 5: 1300.00
Salario (R$) do mes 6: 1365.00

Pressione qualquer tecla para continuar. . . _
```

Figura 8. Exemplo de execução do Programa 3 (parte 2).

## 5.3 Arranjos de Estruturas

Arranjos combinados com estruturas constituem um recurso poderoso utilizado para organizar melhor os dados manipulados em um programa. Um arranjo de estruturas consiste em um conjunto de elementos de mesmo tipo, sendo que cada elemento é uma estrutura. A Figura 9 representa uma variável, chamada `vetorFunc`, que é um arranjo unidimensional cujos elementos são uma estrutura funcionário.



```
typedef
struct
{
    int codFunc;
    int idadeFunc;
    char sexoFunc;
    float salFunc;
} funcionario;

funcionario vetorFunc[5];
```

Figura 9. Vetor de estruturas vetorFunc.

O Programa 4 a seguir exemplifica o uso de arranjos de estruturas. O programa realiza a leitura dos dados de 5 funcionários (i.e., código, idade, sexo e salário) em um primeiro vetor de estruturas. Os dados destes funcionários são então atribuídos a um segundo vetor de estruturas, em ordem inversa. Ao final da atribuição, o primeiro elemento do primeiro vetor será o último elemento do segundo vetor, o segundo elemento do primeiro vetor será o quarto elemento do segundo vetor, e assim sucessivamente. Finalmente, todos os elementos do segundo vetor são impressos na tela.

#### Programa 4

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  # define TAMANHO 5
5
6  // definição do tipo funcionario
7
8  typedef
```

```
9      struct {
10          int codFunc;
11          int idadeFunc;
12          char sexoFunc;
13          float salFunc;
14      } funcionario;
15
16      int main(int argc, char *argv[])
17      {
18          funcionario vetorFunc[TAMANHO];
19          funcionario vetorInversoFunc[TAMANHO];
20
21          int i;          // índice do vetor
22
23          // leitura dos dados dos funcionários
24
25          for (i = 0; i < TAMANHO; i++)
26          {
27              printf("A seguir, entre com todos os dados do funcionario %d.\n", i+1);
28              printf("Digite o codigo: ");
29              scanf("%d", &vetorFunc[i].codFunc);
30              printf("Digite a idade: ");
31              scanf("%d", &vetorFunc[i].idadeFunc);
32              fflush(stdin);
33              printf("Digite o sexo: ");
34              scanf("%c", &vetorFunc[i].sexoFunc);
35              printf("Digite o salario (R$): ");
36              scanf("%f", &vetorFunc[i].salFunc);
37              printf("\n");
38              fflush(stdin);
39          }
40
41          // copiando os funcionários de vetorFunc na ordem inversa
42          // para vetorInversoFunc
43
44          for (i = 0; i < TAMANHO; i++)
45              vetorInversoFunc[i] = vetorFunc[TAMANHO-1-i];
46
47          // escrita dos dados de vetorInversoFunc
48
49          printf("Vetor invertido\n\n");
50          for (i = 0; i < TAMANHO; i++)
51          {
52              printf("Dados do funcionario %d.\n", i+1);
53              printf("Codigo: %d\n", vetorInversoFunc[i].codFunc);
```

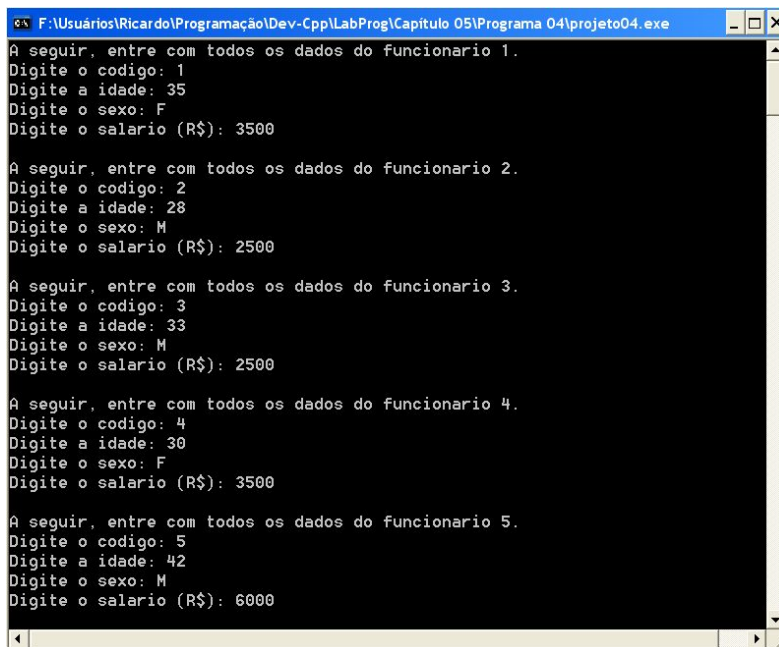
```
54     printf("Idade: %d\n", vetorInversoFunc[i].idadeFunc);
55     printf("Sexo: %c\n", vetorInversoFunc[i].sexoFunc);
56     printf("Salario: %.2f\n", vetorInversoFunc[i].salFunc);
57     printf("\n");
58 }
59
60 // finalização do programa principal
61
62 system("PAUSE");
63 return 0;
64
65 }
```

O Programa 4 define na linha 4 a constante TAMANHO, a qual especifica o número de elementos dos vetores de estruturas. Para facilitar a declaração de um vetor de estruturas, é fortemente recomendada a definição de um novo tipo de dados para a estrutura. Nas linhas 8 a 14 é definido um novo tipo de dados chamado funcionario, que representa uma estrutura com os campos codFunc, idadeFunc, sexoFunc e salFunc. A declaração dos vetores de estruturas é realizada nas linhas 18 e 19. Note que esta declaração é similar à declaração de qualquer vetor. A única diferença é que o tipo do vetor é uma estrutura, e não um tipo de dados simples tal como int, float ou char.

Nas linhas 25 a 39, os dados de todos os funcionários são lidos campo a campo e armazenados na variável vetorFunc. O acesso a campos individuais de elementos de um vetor de estruturas é feito usando o formato: **nome\_variável[índice].nome\_campo**. Este formato também é similar ao utilizado em variáveis do tipo estrutura. A única diferença é a necessidade de especificar o elemento do vetor por meio de seu índice. Por exemplo, a linha 36 realiza a leitura de um número e armazena-o no campo salFunc do elemento na posição i da variável vetorFunc (i.e., vetorFunc[i].salFunc).

Os comandos nas linhas 29, 31, 34 e 36 acessam os elementos da variável vetorFunc campo a campo. Em contrapartida, na linha 45 os elementos das variáveis vetorInversoFunc e vetorFunc são acessados por inteiro. Em outras palavras, a estrutura completa

destes elementos é acessada, ou seja, todos os campos destes elementos são manipulados de uma única vez. Note que o acesso à estrutura completa dos elementos não pode ser usado com as funções `printf( )` e `scanf( )`, desde que estas funções não possuem um especificador de formato para o tipo de dados estrutura. As Figuras Figura 10 e Figura 11 ilustram um exemplo de execução do Programa 4.



```
F:\Usuários\Ricardo\Programação\Dev-Cpp\LabProg\Capitulo 05\Programa 04\projeto04.exe
A seguir, entre com todos os dados do funcionario 1.
Digite o codigo: 1
Digite a idade: 35
Digite o sexo: F
Digite o salario (R$): 3500

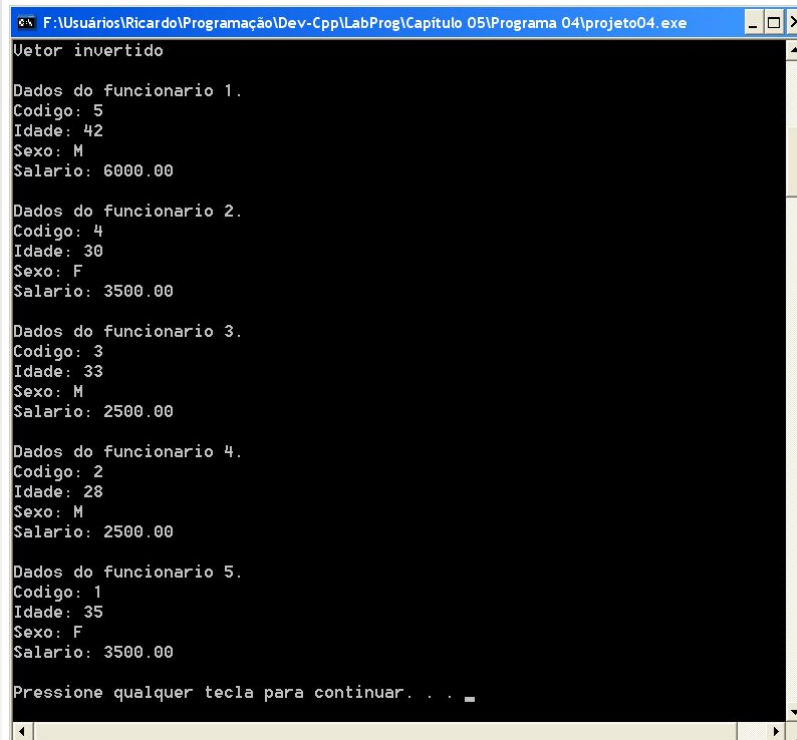
A seguir, entre com todos os dados do funcionario 2.
Digite o codigo: 2
Digite a idade: 28
Digite o sexo: M
Digite o salario (R$): 2500

A seguir, entre com todos os dados do funcionario 3.
Digite o codigo: 3
Digite a idade: 33
Digite o sexo: M
Digite o salario (R$): 2500

A seguir, entre com todos os dados do funcionario 4.
Digite o codigo: 4
Digite a idade: 30
Digite o sexo: F
Digite o salario (R$): 3500

A seguir, entre com todos os dados do funcionario 5.
Digite o codigo: 5
Digite a idade: 42
Digite o sexo: M
Digite o salario (R$): 6000
```

Figura 10. Exemplo de execução do Programa 4 (parte 1).



```
Vetor invertido

Dados do funcionario 1.
Codigo: 5
Idade: 42
Sexo: M
Salario: 6000.00

Dados do funcionario 2.
Codigo: 4
Idade: 30
Sexo: F
Salario: 3500.00

Dados do funcionario 3.
Codigo: 3
Idade: 33
Sexo: M
Salario: 2500.00

Dados do funcionario 4.
Codigo: 2
Idade: 28
Sexo: M
Salario: 2500.00

Dados do funcionario 5.
Codigo: 1
Idade: 35
Sexo: F
Salario: 3500.00

Pressione qualquer tecla para continuar. . . .
```

Figura 11. Exemplo de execução do Programa 4 (parte 2).

## 5.4 Explicações Adicionais

Esta seção reforça a explicação de alguns dos conceitos estudados. Para tanto, serão usados exemplos de trechos de programas.

### Inicialização de arranjos na declaração

Os valores dos elementos de vetores e matrizes podem ser inicializados na sua declaração. O trecho de programa a seguir ilustra a declaração e inicialização do vetor `idadeFunc` (Figura 1) e da matriz `salFunc` (Figura 4).



```
...
int idadeFunc[5] = {54, 23, 32, 25, 41};
float salFunc[4][6] = {1000.00, 1000.00, 1500.47, 1500.47, 1500.47, 1600.00,
                      1300.00, 1300.00, 1300.00, 1300.00, 1300.00, 1300.00,
                      2000.00, 2000.00, 2670.27, 2670.27, 2670.27, 2670.27,
                      1300.00, 1300.00, 1300.00, 1300.00, 1300.00, 1300.00};
...
```

A dimensão de um vetor pode ser omitida em uma declaração com inicialização. A linguagem C calcula automaticamente a quantidade de elementos em função dos elementos contidos em sua declaração. Para matrizes, somente o índice da dimensão 1 pode ser omitido, desde que a linguagem C precisa saber o número de colunas para calcular o endereçamento dos elementos. O trecho de programa a seguir ilustra a declaração de **arranjos não dimensionados**, ou seja, que não especifica alguns de seus índices. Esta declaração é equivalente à realizada no trecho de programa anterior.

```
...
int idadeFunc[] = {54, 23, 32, 25, 41};
float salFunc[][6] = {1000.00, 1000.00, 1500.47, 1500.47, 1500.47, 1600.00,
                     1300.00, 1300.00, 1300.00, 1300.00, 1300.00, 1300.00,
                     2000.00, 2000.00, 2670.27, 2670.27, 2670.27, 2670.27,
                     1300.00, 1300.00, 1300.00, 1300.00, 1300.00, 1300.00};
...
```

### Arranjos Multidimensionais

A linguagem C permite a declaração de arranjos com 3 ou mais dimensões. O número máximo de dimensões que pode ser definido para um arranjo é determinado pelo compilador, mas, em geral pode-se usar uma quantidade grande de dimensões. O trecho de programa a seguir ilustra a declaração de um arranjo tridimensional, chamado de poluicao. Cada elemento deste arranjo armazena um nível de poluição. A dimensão 1 representa o local, a dimensão 2 representa a data de coleta e a dimensão 3 representa o horário da coleta.

...

```
float poluicao[10][31][3];    // coleta do nível de poluição para
                             // 10 cidades
                             // 31 dias
                             // 3 vezes por dia
...
printf("%.2f", poluicao[5][3][1]; // nível de poluição da
                                // sexta cidade
                                // no quarto dia da coleta
                                // no segundo horário da coleta
...
```

### Ponteiros e matrizes

Um aspecto muito importante na linguagem C é o **relacionamento entre os conceitos de arranjos bidimensionais e ponteiros**. O nome de uma matriz sem o seu índice representa o endereço do primeiro elemento da matriz (i.e., &nome\_matriz[0][0]). Este endereço pode ser atribuído a um ponteiro, o qual pode ser usado para acessar os elementos da matriz. Uma vez ocorrida esta atribuição, a aritmética de ponteiros permite o acesso a qualquer um dos elementos. No uso da aritmética de ponteiros, a ordem dos elementos de uma matriz é representada de acordo com a Figura 6.

```
...
float salFunc[4][6] = {1000.00, 1000.00, 1500.47, 1500.47, 1500.47, 1600.00,
                      1300.00, 1300.00, 1300.00, 1300.00, 1300.00, 1300.00,
                      2000.00, 2000.00, 2670.27, 2670.27, 2670.27, 2670.27,
                      1300.00, 1300.00, 1300.00, 1300.00, 1300.00, 1300.00};
float *pont_salFunc;
...
pont_salFunc = salFunc; // &salFunc[0][0]
printf("%.2f", *pont_salFunc); // imprime o valor de salFunc[0][0]: 1000.00
pont_salFunc++;
printf("%.2f", *pont_salFunc); // imprime o valor de salFunc[0][1]: 1000.00
pont_salFunc += 5;
printf("%.2f", *pont_salFunc); // imprime o valor de salFunc[1][0]: 1300.00
...
```

### Vetores e matrizes como argumentos de funções

A linguagem C não permite que arranjos sejam passados por valor para uma função. Isto ocorre porque um arranjo ocupa uma quantidade grande de memória, e a passagem de um arranjo por valor é muito demorada. Ao invés disso, arranjos são sempre passados por referência usando ponteiros. O nome do arranjo representa o endereço de seu primeiro elemento e deve ser o argumento na chamada da função. Esta característica é válida para vetores, matrizes e arranjos de qualquer dimensão.

A definição do parâmetro formal da função, relativo a um arranjo, pode ser realizada de formas distintas. O parâmetro formal pode ser declarado como um ponteiro, como um arranjo dimensionado, ou como um arranjo não dimensionado. Estas três formas de passagem de parâmetro são equivalentes. De fato, independente da forma usada, a linguagem C converte-a para um ponteiro. O trecho de programa a seguir ilustra a chamada de uma função com um argumento arranjo e a declaração desta função usando as três formas de parâmetros formais.

```
...
int idadeFunc[5] = {54, 23, 32, 25, 41};
float salFunc[4][6] = {1000.00, 1000.00, 1500.47, 1500.47, 1500.47, 1600.00,
                      1300.00, 1300.00, 1300.00, 1300.00, 1300.00, 1300.00,
                      2000.00, 2000.00, 2670.27, 2670.27, 2670.27, 2670.27,
                      1300.00, 1300.00, 1300.00, 1300.00, 1300.00, 1300.00};
...
f1(idadeFunc); // &idadeFunc[0]
f2(idadeFunc);
f3(idadeFunc);
...
f4(salFunc); // &salFunc[0][0]
f5(salFunc);
f6(salFunc);
...
void f1(int *p) // primeira forma de declaração do parâmetro formal
{
    // manipulação do vetor idadeFunc
}
```

```
void f2(int idade[5]) // segunda forma de declaração do parâmetro formal
{
    // manipulação do vetor idadeFunc
}

void f3(int idade[]) // terceira forma de declaração do parâmetro formal
{
    // manipulação do vetor idadeFunc
}

void f4(int *p) // primeira forma de declaração do parâmetro formal
{
    // manipulação da matriz salFunc
}

void f5(int salario[4][6]) // segunda forma de declaração do parâmetro formal
{
    // manipulação da matriz salFunc
}

void f6(int salario[][6]) // terceira forma de declaração do parâmetro formal
{
    // manipulação da matriz salFunc
}
...
```

### **Indexação de ponteiros referentes à passagem de arranjos para funções**

Um arranjo sempre é passado para uma função por referência usando ponteiros. Dentro da função, o ponteiro pode ser usado para acessar o conteúdo dos elementos do arranjo. O ponteiro também pode assumir a forma de arranjo indexado (por exemplo, nome\_ponteiro[índice]). O trecho de programa a seguir ilustra as diversas formas de acesso aos elementos de um arranjo usando ponteiros dentro de uma função.

```

...
int idadeFunc[] = {54, 23, 32, 25, 41};
...
f1(idadeFunc); // &idadeFunc[0]
...
void f1(int *p)
{
    *p = 58; // idadeFunc[0] = 58
    p[3] = 32; // idadeFunc[3] = 32
    *(p+2) = 36; // idadeFunc[2] = 36
}
...

```

### Definição alternativa de matrizes

Uma matriz pode ser definida como um vetor de vetores. Ou seja, cada elemento da matriz aponta para um vetor. Este conceito é equivalente ao conceito de matriz linha-coluna. Isto significa que a declaração e o uso de matrizes são os mesmos que os estudados até o momento. A única diferença refere-se à representação gráfica da matriz. Esta representação é mais apropriada quando a matriz é manipulada usando ponteiros. A Figura 12 ilustra a representação da matriz `salFunc[3][5]` da Figura 4.

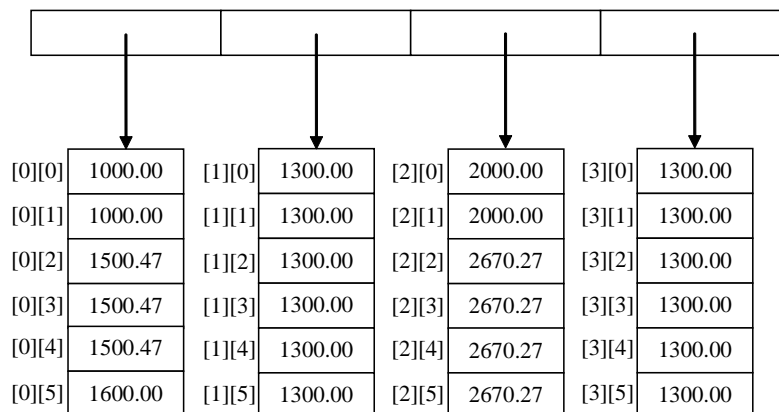


Figura 12. Representação da matriz `salFunc[3][5]` da Figura 4.

## 5.5 Mapeamento de Algoritmo para Programa C

A seguir são apresentados os mapeamentos de 2 algoritmos para programas C, com ênfase em arranjos unidimensionais e bidimensionais.

### Algoritmo 10-1 (apostila de “Construção de Algoritmos”)

```
1  { calcular, para um conjunto de 20 valores inteiros, quantos destes
2    são maiores ou iguais à média do conjunto }
3
4  algoritmo
5    { declaração de variáveis }
6    declare
7      i, valor[20]: inteiro { i: um inteiro; valor: 20 inteiros }
8    contador, soma: inteiro
9    média: real
10
11   { obtenção dos dados e cálculo da média }
12   soma ← 0
13   para i ← 0 até 19 faça
14     leia(valor[i])
15     soma ← soma + valor[i]
16   fim-para
17   média ← soma / 20.0
18
19   { contagem de quantos valores são maiores ou iguais à média }
20   contador ← 0
21   para i ← 0 até 19 faça
22     se valor[i] >= média então
23       contador ← contador + 1
24     fim-se
25   fim-para
26
27   { resultado }
28   escreva("Há", contador, "item(s) maior(es) que a média", média)
29 fim-algoritmo
```

**Programa 5 (equivalente ao algoritmo 10-1)**

```
1  /* calcular, para um conjunto de 20 valores inteiros, quantos destes
2     são maiores ou iguais à média do conjunto */
3
4  #define TAMANHO 20
5  #define TAMANHO_REAL 20.0
6
7  #include <stdio.h>
8  #include <stdlib.h>
9
10 int main(int argc, char *argv[])
11 {
12     // declaração de variáveis
13
14     int i, valor[TAMANHO]; // i: um inteiro; valor: 20 inteiros
15     int soma, contador;
16     float media;
17
18     soma = 0;
19     for (i = 0; i < TAMANHO; i++)
20     {
21         printf("Digite o numero %d: ", i+1);
22         scanf("%d", &valor[i]);
23         soma += valor[i];
24     }
25     media = soma / TAMANHO_REAL;
26
27     // contagem de quantos valores são maiores ou iguais à média
28
29     contador = 0;
30     for (i = 0; i < TAMANHO; i++)
31         if (valor[i] >= media)
32             contador++;
33
34     // resultado
35     printf("\n");
36     printf("Ha %d iten(s) maior(s) que a media %.2f\n", contador, media);
37     printf("\n");
38
39     system("PAUSE");
40     return 0;
41 }
```

### Algoritmo 10-2 (apostila de “Construção de Algoritmos”)

```

1  { listar o(s) melhor(es) aluno(s) de uma turma com 40 alunos, dadas
2  as notas de três provas e sabendo-se que todas possuem o mesmo peso
3  no cálculo da média }
4
5  algoritmo
6  { declarações }
7  constante númeroAlunos: inteiro = 40
8  declare
9      i: inteiro
10     notaProva1, notaProva2, notaProva3, melhorMédia: real
11     nomeAluno[númeroAlunos]: literal
12     média[númeroAlunos]: real
13
14     { obtenção dos dados, cálculo das médias e seleção da melhor }
15     melhorMédia ← -1 { força substituição logo para o primeiro aluno }
16     para i ← 0 até númeroAlunos - 1 faça
17         leia(nomeAluno[i], notaProva1, notaProva2, notaProva3)
18         média[i] ← (notaProva1 + notaProva2 + notaProva3)/3
19
20         se média[i] > melhorMédia então
21             melhorMédia ← média[i]
22         fim-se
23     fim-para
24
25     { apresentação dos resultados }
26     escreva("Melhor nota final:", melhorMédia)
27     para i ← 0 até númeroAlunos - 1 faça
28         se média[i] = melhorMédia então
29             escreva("Nome do aluno:", nomeAluno[i])
30         fim-se
31     fim-para
32 fim-algoritmo

```

### Programa 6 (equivalente ao algoritmo 10-2)

```

1  /* listar o(s) melhor(es) aluno(s) de uma turma com 40 alunos, dadas
2  as notas de três provas e sabendo-se que todas possuem o mesmo peso
3  no cálculo da média */
4
5  #include <stdio.h>
6  #include <stdlib.h>

```



```
7
8     #define TAMANHO_STRINGS 81 // tamanho máximo de cada
string 80 + \0
9
10    int main(int argc, char *argv[])
11    {
12        // declarações
13
14        const int numeroAlunos = 40;
15        int i;
16        float notaProva1, notaProva2, notaProva3, melhorMedia;
17        float media[numeroAlunos];
18
19        // declaração de um vetor de strings
20        // equivalente a uma matriz
21        char nomeAluno[numeroAlunos][TAMANHO_STRINGS];
22
23        // obtenção dos dados, cálculo das médias e seleção da melhor
24
25        melhorMedia = -1; // força substituição logo para o primeiro aluno
26        for (i = 0; i < numeroAlunos; i++)
27        {
28            printf("Digite o nome do aluno: ");
29            gets(nomeAluno[i]);
30            printf("Digite a nota da prova 1: ");
31            scanf("%f", &notaProva1);
32            printf("Digite a nota da prova 2: ");
33            scanf("%f", &notaProva2);
34            printf("Digite a nota da prova 3: ");
35            scanf("%f", &notaProva3);
36            fflush(stdin);
37            printf("\n");
38            media[i] = (notaProva1 + notaProva2 + notaProva3) / 3;
39            if (media[i] > melhorMedia)
40                melhorMedia = media[i];
41        }
42
43        // apresentação dos resultados
44
45        printf("Melhor nota final: %.2f\n\n", melhorMedia);
46
47        for (i = 0; i < numeroAlunos; i++)
48            if (media[i] == melhorMedia)
49                printf("Nome do aluno: %s\n", nomeAluno[i]);
```

```
50     printf("\n");  
51  
52     system("PAUSE");  
53     return 0;  
54 }
```



Departamento de Produção gráfica-UFSCar  
Universidade Federal de São Carlos  
Rodovia Washington Luís, km 235  
13.565-905 - São Carlos - São Paulo - Brasil  
(16) 3351-8351