CURSO DE PÓS-GRADUAÇÃO
“LATO SENSU” (ESPECIALIZAÇÃO) A DISTÂNCIA
ADMINISTRAÇÃO EM REDES LINUX

INTRODUÇÃO À LINGUAGEM C

João Carlos Giacomin

UFLA - Universidade Federal de Lavras
FAEPE - Fundação de Apoio ao Ensino, Pesquisa e Extensão
Lavras – MG

PARCERIA

UFLA – Universidade Federal de Lavras

FAEPE – Fundação de Apoio ao Ensino, Pesquisa e Extensão

REITOR

Fabiano Ribeiro do Vale

VICE-REITOR

Antônio Nazareno Guimarães Mendes

DIRETOR DA EDITORA

Marco Antônio Rezende Alvarenga

PRÓ-REITOR DE PÓS-GRADUAÇÃO

Luiz Edson Mota de Oliveira

COORDENADOR DE PÓS-GRADUAÇÃO "LATO SENSU"

Antônio Ricardo Evangelista

COORDENADOR DO CURSO

Joaquim Quinteiro Uchôa

PRESIDENTE DO CONSELHO DELIBERATIVO DA FAEPE

Antônio Eduardo Furtini Neto

EDITORAÇÃO

Grupo Ginux (`http://ginux.comp.ufla.br/`)

IMPRESSÃO

Gráfica Universitária/UFLA

Ficha Catalográfica preparada pela Divisão de Processos Técnicos da Biblioteca Central da UFLA

Lacerda, Wilian Soares

Arquitetura de Computadores / Wilian Soares Lacerda. - Lavras: UFLA/FAEPE, 2002.

190 p. : il. - Curso de Pós-Graduação "Lato Sensu" (Especialização) a Distância: Administração em Redes Linux

Bibliografia.

1. Processamento de Dados. 2. Computador Digital. 3. Arquitetura de Computadores. 4. Organização de Computadores. 5. Unidade Central de Processamento (UCP/CPU). I Universidade Federal de Lavras. II. Fundação de Apoio ao Ensino, Pesquisa e Extensão. III. Título.

CDD 004.22

SUMÁRIO

1 Introdução **11**
1.1 Linguagens de Máquina, Linguagens de Montagem, e Linguagens de Alto-Nível 11
1.2 A História do C 13
1.3 A Biblioteca padrão C 13
1.4 Características de C 14
1.5 C e C++ 15
1.6 O compilador GCC 15

2 Estrutura de um programa em C **17**
2.1 Visão geral de um programa C 18
2.2 Um programa utilizando uma função 19

3 Sintaxe **21**
3.1 Identificadores 21
3.2 Tipos de variáveis 22
3.2.1 Tipos Básicos 23
3.3 Declaração e Inicialização de Variáveis 23
3.4 Operadores 26
3.4.1 Operador de atribuição 26
3.4.2 Operadores Aritméticos 26
3.4.3 Operadores relacionais e lógicos 27
3.4.4 Precedência 29
3.4.5 Operador cast (modelador) 29
3.4.6 Operador sizeof 30

4 Funções Básicas da Biblioteca C **33**
4.1 Função printf( ) 33
4.2 Função scanf( ) 35
4.3 Função getchar() 36
4.4 Função putchar() 36

5 Estruturas de Controle de Fluxo **39**
5.1 If 39
5.2 if-else-if 40
5.3 Operador ternário 41
5.4 Switch 42
5.5 Loop for 42
5.6 While 47
5.7 Do-while 48
5.8 Break 49
5.9 Continue 49

6 Matrizes **53**
6.1 Matriz unidimensional . . . 53
6.2 Matriz Multidimensional . . . 54
6.3 Matrizes estáticas . . . 54
6.4 Limites das Matrizes . . . 54

7 Manipulação de Strings **57**
7.1 Função gets() . . . 57
7.2 Função puts() . . . 58
7.3 Função strcpy() . . . 58
7.4 Função strcat() . . . 59
7.5 Função strcmp() . . . 59

8 Ponteiros **61**
8.1 Declarando Ponteiros . . . 61
8.2 Manipulação de Ponteiros . . . 62
8.3 Expressões com Ponteiros . . . 63
8.4 Ponteiros para ponteiros . . . 64
8.5 Problemas com ponteiros . . . 65

9 Ponteiros e Matrizes **67**
9.1 Manipulando Matrizes através de Ponteiros . . . 68
9.2 String e Ponteiros . . . 69
9.3 Matrizes de Ponteiros . . . 70

10 Funções **71**
10.1 Função sem Retorno . . . 71
10.2 Função com Retorno . . . 72
10.3 Parâmetros Formais . . . 72
10.3.1 Chamada por Valor . . . 73
10.3.2 Chamada por Referência . . . 73
10.4 Classe de Variáveis . . . 74
10.4.1 Variáveis locais . . . 75
10.4.2 Variáveis Globais . . . 76
10.4.3 Variáveis Estáticas . . . 76
10.5 Funções com Matrizes . . . 78
10.5.1 Passando Parâmetros Formais . . . 78
10.5.2 Alterando os Valores da Matriz . . . 80

11 Argumentos da Linha de Comando **81**

12 Estruturas, Uniões e Enumerações **83**
12.1 Estruturas . . . 83
12.2 Uniões . . . 83
12.3 Enumerações . . . 85

13 Noções de Manipulação de Arquivos **89**
13.1 Abrindo e Fechando um Arquivo 89
13.1.1 A função fopen 89
13.1.2 A função exit 90
13.1.3 A função fclose 91
13.2 Lendo e Escrevendo Caracteres em Arquivos 92
13.2.1 putc 92
13.2.2 getc 93
13.2.3 feof 93

14 Noções de Alocação Dinâmica **95**

15 Exercícios **97**
15.1 Capítulo 1 97
15.2 Capítulo 2 97
15.3 Capítulo 3 98
15.4 Capítulo 4 98
15.5 Capítulo 5 98
15.6 Capítulo 6 101
15.7 Capítulo 7 101
15.8 Capítulos 8 e 9 102
15.9 Capítulo 10 103
15.10 Capítulo 11 104
15.11 Capítulo 12 104
15.12 Capítulo 13 104
15.13 Capítulo 14 104

Referências Bibliográficas **105**

LISTA DE FIGURAS

1.1 Trecho de programa em linguagem de máquina 11
1.2 Trecho de programa em linguagem de montagem 12
1.3 Trecho de programa em linguagem de alto nível 12

2.1 Estrutura de um programa em C 17
2.2 Um programa simples em C 18
2.3 Ambiente de programação do C 19
2.4 Um programa que utiliza uma função 20

3.1 Programa com instruções básicas 22
3.2 Inicialização de variáveis 25
3.3 Programa com diferentes tipos de variáveis. 25
3.4 Programa com operadores aritméticos 28
3.5 Programa para conversão de pés para metros 28
3.6 Programa com operadores relacionais e lógicos 29
3.7 Programa Maior_de_dois 29
3.8 Programa com exemplo de utilização de operador **cast** 30

4.1 Exemplo de utilização da função `printf` 34
4.2 Tamanho de campos na impressão 34
4.3 Arredondamento de números em ponto flutuante 34
4.4 Alinhamento de campos numéricos 34
4.5 Complementando com zeros à esquerda 35
4.6 Formas para imprimir caracteres utilizando a função `printf` 35
4.7 Exemplo de programa que utiliza a função scanf 36
4.8 Exemplo de programa que utiliza a função `getchar` 36
4.9 Exemplo de programa que utiliza a função `putchar` 37

5.1 Exemplo de programa que utiliza o comando de decisão if 39
5.2 O comando de decisão if com um bloco de instruções 40
5.3 Exemplo de um programa que utiliza o comando if-else-if 41
5.4 Exemplo de programa que utiliza o operador ternário 42
5.5 Exemplo de programa que utiliza o comando switch 43
5.6 Exemplo de for com apenas um comando 44
5.7 Exemplo de for com um bloco de instruções 44
5.8 Programa para calcular o valor de π 45
5.9 Laço de for com duas variáveis de controle 46
5.10 Loop infinito 46
5.11 Laços de for aninhados 46
5.12 Programa que exibe uma tabela das quatro primeiras potências dos números de 1 a 9 47
5.13 Exemplo simples de while 48
5.14 Geração de uma tabela de quadrados de inteiros utilizando while 48

5.15 Geração de uma tabela de quadrados de inteiros utilizando do-while 49
5.16 Exemplo de utilização do laço do-while 50
5.17 Exemplo simples de utilização do comando break 50
5.18 Exemplo de utilização do comando continue 51

6.1 Exemplo simples de utilização de matriz 54
6.2 Exemplo de armazenamento de dados em matriz unidimensional 55
6.3 Operações com matriz 55
6.4 Exemplo simples de utilização de matriz multidimensional 56
6.5 Exemplo de utilização de matriz estática 56
6.6 Erro na utilização de matriz 56

7.1 Exemplo simples de operação com strings 57
7.2 Programa simples que utiliza a função gets 58
7.3 Exemplo simples de utilização da função puts 58
7.4 Utilização da função strcpy da biblioteca string.h 59
7.5 Utilização da função strcat da biblioteca string.h 59
7.6 Utilização da função strcmp da biblioteca string.h 60

8.1 Utilização de ponteiros 63
8.2 Operações com Ponteiros 64
8.3 Ponteiros para ponteiros 65

9.1 Cópia de uma string (versão matriz) 68
9.2 Cópia de uma string (versão ponteiro) 69
9.3 Uso de um ponteiro no lugar de uma string 69
9.4 Utilização de matrizes de ponteiros para mensagens de erro 70

10.1 Inversão de uma string usando uma função que recebe um ponteiro 72
10.2 Um codificador simples 73
10.3 Função com retorno de inteiro 74
10.4 Parâmetros formais de uma função 74
10.5 Passagem de valor de uma variável para uma função 75
10.6 Função que utiliza passagem por referência 75
10.7 Variáveis locais em uma função 76
10.8 Variáveis globais em um programa 77
10.9 Variáveis locais em uma função 77
10.10 Função que utiliza uma matriz 78
10.11 Função que opera sobre uma matriz utilizando um ponteiro 79
10.12 Função que procura uma substring dentro de uma string 79
10.13 Alterando valores de uma matriz 80

11.1 Programa que recebe uma string na linha de comando 81
11.2 Programa que recebe argumentos na linha de comando 82

12.1 Exemplo simples de utilização de estrutura de dados 84
12.2 União de variáveis de mesmo tipo 85

12.3 União de variáveis de tipos diferentes 85
12.4 Programa que utiliza a union número 86
12.5 Exemplo de utilização de enumerações 87

13.1 Escrevendo e lendo em arquivo 90
13.2 Uso da função exit na abertura de arquivos 92

14.1 Primeiro exemplo de alocação dinâmica de memória 95
14.2 Segundo exemplo de alocação dinâmica de memória 96

15.1 Capítulo 2: Programa para o exercício 1 97
15.2 Capítulo 3: programa para o exercício 1 98
15.3 Capítulo 4: Programa para o exercício 1 99
15.4 Capítulo 5: Programa para o exercício 1 100
15.5 Capítulo 5: Algoritmo para o exercício 3 101
15.6 Capítulo 7: programa para o exercício 1 102
15.7 Capítulo 10: Programa para o exercício 2 103

LISTA DE TABELAS

3.1 Palavras reservadas do C 22
3.2 Tipos de dados do C 24
3.3 Operadores aritméticos 26
3.4 Comandos de atribuição reduzidos 27
3.5 Operadores relacionais e lógicos 28
3.6 Nível de precedência dos operadores 30

4.1 Caracteres de controle da função printf 33

13.1 Modos válidos de abertura de arquivos 91

1

INTRODUÇÃO

1.1 LINGUAGENS DE MÁQUINA, LINGUAGENS DE MONTAGEM, E LINGUAGENS DE ALTO-NÍVEL

Programadores escrevem instruções em várias linguagens de programação, algumas diretamente inteligíveis pelo computador e outras que requerem passos intermediários de *tradução*. Centenas de linguagens de computador são usadas atualmente. Estas podem ser divididas em três grandes grupos:

1. Linguagens de máquina,
2. Linguagens de montagem,
3. Linguagens de alto nível.

Qualquer computador pode entender diretamente apenas sua própria linguagem de máquina. A linguagem de máquina é a "linguagem natural" de um computador em particular. Ela é relacionada diretamente ao projeto do hardware daquele computador. Linguagens de máquina geralmente consistem de cadeias de números (que se reduzem a 1s e 0s) que instruem o computador a realizar suas operações mais elementares, uma de cada vez. Linguagens de máquina são dependentes da máquina (*machine-dependent*), isto é, uma linguagem de máquina particular pode ser usada apenas em um tipo de computador. Linguagens de máquina são desajeitadas para humanos, como poderá ser visto a seguir no trecho de um programa, na Figura 1.1, que soma pagamento de hora-extra ao pagamento básico e armazena o resultado no pagamento total.

```
+1300042774
+1400593419
+1200274027
```

Figura 1.1: Trecho de programa em linguagem de máquina

Quando os computadores se tornaram mais populares, tornou-se claro que a programação em linguagens de máquina era muito lenta e tediosa para a maioria dos programadores. Em vez de usar as cadeias de números que computadores podiam entender diretamente, os programadores começaram a usar abreviações derivadas da língua inglesa para representar as operações elementares do computador. Estas abreviações do Inglês formaram a base das linguagens de montagem (*assembly languages*). Tradutores de programas, chamados *assemblers*, foram desenvolvidos para converter programas em linguagem de montagem (*assembly*) para linguagem de máquina na velocidade dos computadores. A Figura 1.2 mostra um trecho de um programa em linguagem de montagem que também soma o pagamento de hora-extra ao pagamento básico e armazena o resultado no pagamento total, mas mais claramente que seu equivalente em linguagem de máquina:

```
LOAD    BASEPAY
ADD     OVERPAY
STORE   GROSSPAY
```

Figura 1.2: Trecho de programa em linguagem de montagem

A utilização do computador cresceu rapidamente com o advento das linguagens de montagem, mas estas ainda requeriam muitas instruções para realizar mesmo as tarefas mais simples. Para acelerar o processo de programação, linguagens de alto nível foram desenvolvidas nas quais declarações simples poderiam ser escritas para realizar tarefas substanciais. Os programas tradutores que convertem programas de linguagens de alto nível em programas de linguagem de máquina são chamados compiladores. Linguagens de alto nível permitem aos programadores escreverem instruções que se parecem proximamente com o Inglês diário e contêm notações matemáticas comumente usadas. Um programa de folha de pagamento escrito em linguagem de alto nível poderia conter uma declaração tal como mostrada na Figura 1.3

```
grossPay = basePay + overTimePay
```

Figura 1.3: Trecho de programa em linguagem de alto nível

Obviamente, linguagens de alto nível são muito mais desejáveis do ponto de vista dos programadores que linguagens de máquina ou linguagens de montagem. C e C++ estão entre as mais poderosas e mais largamente usadas linguagens de alto nível.

1.2 A HISTÓRIA DO C

C evoluiu das linguagens anteriores, BCPL e B. BCPL foi desenvolvida em 1967 por Martin Richards como a linguagem para escrever softwares de sistemas operacionais e compiladores. Ken Thompsom modelou muitas características de sua linguagem B, baseando-se BCPL desenvolvida por outros anteriormente, e usou B para criar as primeiras versões do sistema operacional UNIX no Bell Laboratories em 1970 em um computador DEC PDP-7. Tanto BCPL quanto B eram linguagens não “tipadas” — todos os itens de dados ocupavam uma palavra na memória e o fardo de tratar um dado como um número inteiro ou um número real, por exemplo, ficava a cargo do programador.

A linguagem C foi uma evolução da B feita por Dennis Ritchie no Bell Laboratories e foi originalmente implementada em um computador DEC PDP-11 em 1972. C inicialmente tornou-se largamente conhecida como a linguagem de desenvolvimento do sistema operacional UNIX. Atualmente, a maioria dos novos sistemas operacionais maiores são escritos em C e/ou C++. C está disponível para a maioria dos computadores atuais. C é independente de Hardware. Com um projeto cuidadoso, é possível escrever programas em C que são portáveis para a maioria dos computadores. C usa muitos dos importantes conceitos do BCPL e do B enquanto soma a tipagem de dados e outras características poderosas.

No final da década de 70, C havia evoluído para o que agora é conhecido como o tradicional “C”. A publicação, em 1978, do livro de Kernighan e Ritchie, *The C Programming Language*, trouxe uma grande atenção à linguagem. Esta publicação tornou-se um dos livros de ciência da computação de maior sucesso de todos os tempos.

A rápida expansão do C sobre vários tipos de computadores (algumas vezes chamados plataformas de Hardware) conduziu a muitas variações. Estas eram similares mas sempre incompatíveis. Isto era um problema sério para desenvolvedores de programas que precisavam desenvolver código que rodasse sobre várias plataformas. Tornou-se claro que uma versão padrão de C era necessária. Em 1983, o comitê técnico X3J11 foi criado sob a *American National Standards Committee on Computers and Information Processing* (X3), para “fornecer uma definição de linguagem não ambígua e independente de máquina”. Em 1989, o padrão foi aprovado. O documento é conhecido como ANSI/ISO 9899: 1990. Cópias deste documento podem ser requisitadas ao *Americam National Institute*. A Segunda edição de Kernighan e Ritchie, publicada em 1988, reflete esta versão chamada ANSI C, agora usada em todo o mundo.

1.3 A BIBLIOTECA PADRÃO C

Programas em C consistem de módulos ou pedaços, chamados *funções*. Você pode programar todas as funções que você precisa para formar um programa em C, mas a maio-

ria dos programadores em C aproveitam uma rica coleção de funções existentes, chamada *C Standard Library*. Portanto, há realmente duas etapas para aprender o "mundo" C. A primeira é aprender a própria linguagem C, e a segunda aprender como usar as funções da biblioteca padrão C.

Neste curso, você será levado a usar a abordagem de construção de blocos (*Building block approach*) para criar programas. Evite reinventar a roda. Use pedaços existentes — isto é chamado reutilização de software e é a chave para o campo de desenvolvimento de programas orientados ao objeto como veremos. Quando programando em C você usará tipicamente os seguintes blocos de construção:

- Funções da biblioteca padrão C,
- Funções que você mesmo criou,
- Funções que outras pessoas criaram e disponibilizaram para você.

A vantagem de criar suas próprias funções é que você vai saber exatamente como elas funcionam. Você poderá examinar o código C. A desvantagem é o tempo consumido em esforço que leva do projeto ao desenvolvimento de novas funções.

Usar funções existentes evita reinventar a roda. No caso de funções padrão ANSI, você sabe que elas são cuidadosamente escritas, e você sabe que por estar usando funções que são disponíveis em virtualmente todas as implementações ANSI C, seu programa terá uma chance muito maior de ser portável.

1.4 CARACTERÍSTICAS DE C

C é uma linguagem altamente portável e bastante eficiente em termos de desempenho. É uma linguagem de propósito geral, sendo utilizada para desenvolver os mais diversos tipos de software.

Entre as principais características da linguagem C, podem ser citadas:

- portabilidade
- modularidade
- recursos de baixo nível
- geração de código eficiente
- simplicidade
- facilidade de uso
- possibilidade de ser usada para os mais variados propósitos
- indicada para escrever compiladores, editores de textos, bancos de dados, etc.

Como resultado, tem-se que C e C++ podem ser consideradas “padrões de indústria” pelas empresas de desenvolvimento de software. Várias dessas empresas adotam estas linguagens na maioria de seus projetos. Para o programador, o conhecimento da linguagem C é de grande valor.

1.5 C E C++

Algumas vezes os iniciantes confundem o que é C++ e como difere de C. C++ é uma versão estendida e melhorada de C que foi projetada para suportar programação orientada a objetos (OOP). C++ contém e suporta toda a linguagem C e mais um conjunto de extensões orientadas a objetos.

Por muitos anos ainda os programadores escreverão, manterão e utilizarão programas escritos em C, sem se preocuparem em utilizar os benefícios da orientação a objetos proporcionados por C++, por isso o código escrito em C estará em uso por muitos anos ainda.

1.6 O COMPILADOR GCC

Todos os programas utilizados como exemplo neste curso utilizam apenas funções da linguagem C padrão ANSI (ANSI C), portanto recomendamos a utilização de um compilador de C que funcione bem para programas que seguem este padrão. O compilador GCC é o compilador de C do projeto GNU (`http//:www.gnu.org`), é gratuito e de código aberto.

O projeto GNU (Gnu’s Not Unix) foi proposto por Richard Stallman em 1983. Richard Stallman é pesquisador do MIT e inventor do editor EMACS. O projeto teve início em 1984, e a proposta era desenvolver um sistema operacional completo e livre, similar ao Unix: o sistema GNU (GNU é um acrônimo recursivo que se pronuncia “guh-NEW” ou “guniw”). Variações do sistema GNU, que utilizam o núcleo Linux, são hoje largamente utilizadas; apesar desses sistemas serem normalmente chamados de “Linux”, eles são mais precisamente chamados Sistemas GNU/Linux.

O desenvolvimento do GCC é uma parte do Projeto GNU, com o objetivo de melhorar o compilador usado no sistema GNU incluindo a variante GNU/Linux. O trabalho de desenvolvimento do GCC usa um ambiente aberto de desenvolvimento e suporta muitas outras plataformas com o propósito de promover um compilador de primeira linha (world-class) que seja otimizado continuamente, de reunir um grande time de desenvolvedores, de assegurar que o GCC e o sistema GNU trabalhem em múltiplas arquiteturas e diversos ambientes, e de atender a muitos outros testes e mais características de GCC.

Em Abril de 1999, o *egcs steering committee* foi indicado pelo FSF como o mantenedor oficial GNU para GCC. Naquele tempo GCC foi renomeado para “GNU C Compiler” e para “GNU Compiler Collection” e recebeu uma nova missão.

Atualmente GCC contém aplicações para C, C++, Objective C, Chill, Fortran, e Java bem como bibliotecas para estas linguagens (libstdc++, libgcj,...).

Uma versão do compilador GCC pode ser obtida no ftp do curso de Ciência da Computação da UFLA[1]. Obtenha uma cópia do compilador e instale em seu computador.

De posse do GCC instalado e configurado (independente de sua portagem ou ambiente), para compilar um programa denominado `nome.c`, execute a seguinte linha de comando:

```
gcc nome.c -o nome.exe
```

onde nome é o nome dado ao programa. Com isso, o arquivo `nome.c` será compilado gerado um executável `nome.exe` (em ambientes Linux, a extensão ‘`.exe`’ é desnecessária).

[1] `ftp://ftp.comp.ufla.br/pub/`

2

ESTRUTURA DE UM PROGRAMA EM C

Um programa em C é estruturado sobre funções. O programa possui uma ou várias funções, sendo que a principal, que dá início ao programa e chama todas as outras, é sempre chamada **main**. Além dessas, existem outras pré-programadas, que são incluídas nos programas como arquivos de cabeçalho, ou arquivos de inclusão. Todas as funções começam com **{** e terminam com **}** , como indicado na Figura 2.1.

```
/*  Estrutura de um programa em C  */
# include <arquivo_cabecalho.h>
int main ( )
{
   declaração de variáveis
   instrução_1;
   instrução_2;
   função_1(variáveis);
   instrução_3;
      -
      -
}
int função_1 (variáveis)
{
   declaração de variáveis
   instrução_1;
   instrução_2;
      -
      -
   return (INT);
}
```

Figura 2.1: Estrutura de um programa em C

A primeira linha compreende apenas comentários. Todos os comentários iniciam com `/*` e terminam com `*/`. A segunda linha indica a inclusão de um arquivo de cabeçalho. A terceira linha, inicia a função **main**. Após a função **main**, foi programada outra função chamada **função_1**. Excetuando a **main**, todas as outras funções podem ter qualquer nome que o programador desejar.

A declaração de funções segue sempre o padrão: **tipo nome (parâmetros)**. Nesse caso, **tipo** refere-se ao tipo de valor que a função pode retornar (inteiro, decimal, letra, etc), **nome** é o nome dado à função, e **parâmetros** são os valores passados pelo programa para a função. Quando não for especificado, o tipo da função será **int** (inteiro). Não é obrigatório o uso de parâmetros, como pode ser visto no exemplo, na função **main**.

A Figura 2.2 apresenta um programa simples escrito em linguagem C.

```
/*  programa exemplo 01  */
# include <stdio.h>
main ( )
{
   printf("Alo, Mundo!");
}
```

Figura 2.2: Um programa simples em C

Edite este programa com o nome **exemp_01.c** e compile-o utilizando o compilador GCC, ou outro de sua preferência. Escreva na linha de comando do computador:

```
gcc exemp_01.c -o exemp_01
```

Isto indica ao computador para utilizar o compilador GCC para compilar o programa **exemp_01.c** escrito em **C** e dar ao arquivo executável gerado o nome de **exemp_01**. Ao final, rode o programa.

A função **printf** já está programada dentro de **stdio.h**, e é utilizada para escrever mensagens na tela do computador. As funções pré-programadas devem ser escritas em letras minúsculas.

Modifique este programa para escrever outra mensagem além de “Alo, Mundo!”.

2.1 VISÃO GERAL DE UM PROGRAMA C

A geração do programa executável a partir do programa fonte obedece a uma seqüência de operações antes de tornar-se um executável, conforme descrito na Figura 2.3. Depois de escrever o módulo fonte em um editor de textos, o programador aciona o compilador que

no nosso exemplo é chamado pelo comando **gcc**. Essa ação desencadeia uma seqüência de etapas, cada qual traduzindo a codificação do usuário para uma forma de linguagem de nível inferior, que termina com o **executável** criado pelo **lincador**.

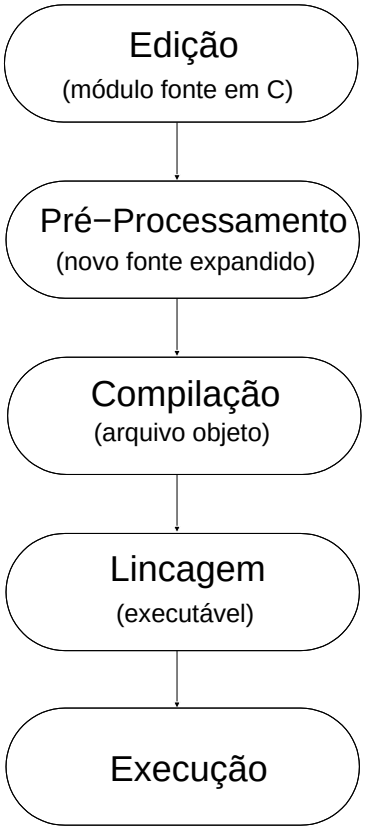

Figura 2.3: Ambiente de programação do C

2.2 UM PROGRAMA UTILIZANDO UMA FUNÇÃO

Para que fique entendida a estrutura de um programa em C, façamos um segundo exemplo (Figura 2.4), o qual utiliza uma função que retorna um valor do tipo inteiro.

A função **quadrado** é chamada pela **main** na sua quarta linha e retorna para a função **main** o quadrado do número passado como argumento. Na função **main**, o valor retornado é atribuído a variável **quad**. Ao final, o número lido pela função **scanf** e o seu quadrado são escritos na tela, utilizando-se a função **printf**.

A função **scanf** também pertence a **stdio.h** e é utilizada para ler valores enviados pelo teclado. Edite este programa com o nome **exemp_02.c** e compile-o utilizando o compilador GCC. Escreva na linha de comando do computador:

```
gcc exemp_02.c -o exemp_02
```

```c
/*  programa exemplo 02  */
# include <stdio.h>

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

main ( )
{
   int num, quad;
   printf("Envie um numero inteiro: ");
   scanf("%d",&num);
   quad = quadrado(num);
   printf("\n Quadrado de %d é : %d", num, quad);

}
```

Figura 2.4: Um programa que utiliza uma função

Modifique este programa para que ele calcule o cubo do número: crie uma função cubo.

3

SINTAXE

Sintaxe são regras detalhadas para cada construção válida na linguagem C. Estas regras estão relacionadas com os **tipos**, as **declarações**, as **funções** e as **expressões**.

Os **tipos** definem as propriedades dos dados manipulados em um programa.

As **declarações** expressam as partes do programa, podendo dar significado a um identificador, alocar memória, definir conteúdo inicial, definir funções.

As **funções** especificam as ações que um programa executa quando roda. A determinação e alteração de valores, e a chamada de funções de I/O são definidas nas **expressões**.

As **funções** são as entidades operacionais básicas dos programas em C, que por sua vez são a união de uma ou mais funções executando cada qual o seu trabalho. Há funções básicas que estão definidas na **biblioteca C**. As funções **printf( )** e **scanf( )** por exemplo, permitem respectivamente escrever na tela e ler os dados a partir do teclado. O programador também pode definir novas funções em seus programas, como rotinas para cálculos, impressão, etc.

Todo programa C inicia sua execução chamando a função **main( )**, sendo obrigatória a sua declaração no programa principal.

Comentários no programa são colocados entre `/*` e `*/` não sendo considerados na compilação. Os comentários podem também ser colocados em uma linha, após o sinal `//`.

Cada instrução encerra com **;** (ponto e vírgula) que faz parte do comando.

A Figura 3.1 mostra um exemplo de programa que utiliza estes conceitos básicos.

3.1 IDENTIFICADORES

São nomes usados para se fazer referência a variáveis, funções, rótulos e vários outros objetos definidos pelo usuário. O primeiro caracter deve ser uma letra ou um sublinhado. Os 32 primeiros caracteres de um identificador são significativos.

O C é “Case Sensitive”, isto é, os compiladores C entendem as letras maiúsculas e minúsculas como sendo diferentes. Portando uma variável declarada como **X1** será dife-

```
# include <stdio.h>
main( )  /* função obrigatória */
{
   char Nome[20];
   printf("\n Alô! Qual é o seu nome ? ");
   scanf("%s",Nome);
   printf("\n Bom dia, %s ! \n",Nome);
}
```

Figura 3.1: Programa com instruções básicas

rente de uma variável declarada com **x1**, uma variável chamada **Tempo** será diferente de **TEMPO**, **tempo** e **TeMPo**. Os comandos básicos do C devem ser escritos sempre com minúsculas (**if**, **for**, **switch**, etc.), caso contrário, o compilador não irá interpretá-los como sendo comandos, mas como variáveis. O C possui outros comandos e palavras reservada que são todos escritos em letras minúsculas. A Tabela 3.1 apresentada uma lista com as palavras reservadas do padrão ANSI C.

Tabela 3.1: Palavras reservadas do C

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

3.2 TIPOS DE VARIÁVEIS

Quando você declara um identificador, dá a ele um tipo. Os tipos principais podem ser colocados dentro da classe do tipo de objeto de dado. Um tipo de objeto de dados determina como valores de dados são representados, que valores pode expressar, e que tipo de operações você pode executar com estes valores. Os nomes dos identificadores As variáveis, identificadores, no C podem ter qualquer nome se duas condições forem satisfeitas: o nome deve começar com uma letra ou sublinhado (_) e os caracteres subsequentes devem

ser letras, números ou sublinhado (_). Há apenas mais duas restrições: o nome de uma variável não pode ser igual a uma palavra reservada, nem igual ao nome de uma função declarada pelo programador, ou pelas bibliotecas do C. Variáveis de até 32 caracteres são aceitas.

3.2.1 Tipos Básicos

Há cinco tipos básicos de dados em C: caractere (**char**), inteiro (**int**), ponto flutuante (**float**), ponto flutuante de precisão dupla (**double**) e vazio (**void**). Todos os outros tipos de dados em C são baseados nestes 5 tipos básicos. O tamanho e a faixa de representação de cada um destes tipos de dados depende do processador utilizado e da implementação do compilador C. O padrão ANSI determina apenas a faixa mínima de cada tipo, não o seu tamanho em bytes.

Exceto **void**, os tipos básicos de C podem ter vários modificadores precedendo-os. Um modificador é usado para alterar o significado de um tipo básico, modificando o seu tamanho ou sua forma de representação. Os modificadores são: **signed**, **unsigned**, **long** e **short**. Os tipos básicos do C juntamente com suas formas modificadas são apresentados na Tabela 3.2.

Uma implementação do compilador pode mostrar um faixa maior do que a mostrada na Tabela 3.2, mas não uma faixa menor. As potências de 2 usadas significam:

$$\begin{aligned} 2^{15} &= 32.768 \\ 2^{16} &= 65536 \\ 2^{31} &= 2.147.483.648 \\ 2^{32} &= 4.294.967.298 \end{aligned}$$

3.3 DECLARAÇÃO E INICIALIZAÇÃO DE VARIÁVEIS

As variáveis no C devem ser declaradas antes de serem usadas. A forma geral da declaração de variáveis é:

```
tipo_da_variável lista_de_variáveis;
```

As variáveis da lista de variáveis terão todas o mesmo tipo e deverão ser separadas por vírgula. Como o tipo **default** do C é o **int**, quando vamos declarar variáveis **int** com algum dos modificadores de tipo, basta colocar o nome do modificador de tipo. Assim um **long** basta para declarar um **long int**.

Há três lugares nos quais podemos declarar variáveis. O primeiro é fora de todas as funções do programa. Estas variáveis são chamadas variáveis globais e podem ser usadas

Tabela 3.2: Tipos de dados do C

Inteiros

char	$[0, 128)$	igual a signed char ou unsigned char
signed char	$[-128, 128)$	inteiro de pelo menos 8 bits
unsigned char	$[0, 256)$	mesmo que signed char sem negativos
short	$(2^{-15}, 2^{15})$	inteiro de pelo menos 16 bits; tamanho pelo menos igual a char
unsigned short	$[0, 2^{16})$	mesmo tamanho que short sem negativos
int	$(2^{-15}, 2^{15})$	inteiro de pelo menos 16 bits; tamanho pelo menos igual a short
unsigned int	$[0, 216)$	mesmo tamanho que int sem negativos
long	$(2^{-31}, 2^{31})$	inteiro com sinal de pelo menos 32 bits; tamanho pelo menos igual a int
unsigned log	$[0, 2^{32})$	mesmo tamanho que long sem valores negativos

Ponto Flutuante

float	$[3.4E-38, 3.4E+38]$	pelo menos 6 dígitos de precisão decimal
double	$(1.7E-308, 1.7E+308)$	pelo menos 10 dígitos decimais e precisão maior que do float
long double	$(1.7E-308, 1.7E+308)$	pelo menos 10 dígitos decimais e precisão maior que do double

a partir de qualquer lugar no programa. Pode-se dizer que, como elas estão fora de todas as funções, todas as funções as vêem. O segundo lugar no qual se pode declarar variáveis é no início de um bloco de código de uma função. Estas variáveis são chamadas locais e só têm validade dentro do bloco no qual são declaradas, isto é, só a função à qual ela pertence sabe da existência desta variável. O terceiro lugar onde se pode declarar variáveis é na lista de parâmetros de uma função. Mais uma vez, apesar de estas variáveis receberem valores externos, são conhecidas apenas pela função onde são declaradas.

As regras que regem onde uma variável é válida chamam-se regras de escopo da variável. Há mais dois detalhes que devem ser ressaltados. Duas variáveis globais não podem ter o mesmo nome. O mesmo vale para duas variáveis locais de uma mesma função. Já duas variáveis locais, de funções diferentes, podem ter o mesmo nome sem perigo algum de conflito.

Podemos inicializar variáveis no momento de sua declaração. Para fazer isto podemos usar a forma geral

```
tipo_da_variável nome_da_variável = constante;
```

Isto é importante pois quando o C cria uma variável ele não a inicializa. Isto significa que até que um primeiro valor seja atribuído à nova variável ela tem um valor indefinido e que não pode ser utilizado para nada. Nunca presuma que uma variável declarada vale zero ou qualquer outro valor. Exemplos de inicialização podem ser vistos na Figura 3.2.

```
char ch='D';
int count=0;
float pi=3.1416;
```

Figura 3.2: Inicialização de variáveis

A Figura 3.3 apresenta um programa que identifica o tamanho de variáveis de diferentes tipos.

```
# include <stdio.h>
main()
{
   char c;
   unsigned char uc;
   int i;
   unsigned int ui;
   float f;
   double d;
   printf("char %d",sizeof(c));
   printf("unsigned char %d",sizeof(uc));
   printf("int %d",sizeof(i));
   printf("unsigned int %d",sizeof(ui));
   printf("float %d",sizeof(f));
   printf("double %d",sizeof(d));
}
```

Figura 3.3: Programa com diferentes tipos de variáveis.

3.4 OPERADORES

3.4.1 Operador de atribuição

O operador de atribuição em C é o sinal de igual "=". Ao contrário de outras linguagens, o operador de atribuição pode ser utilizado em expressões que também envolvem outros operadores. O operador de atribuição "=" atribui à variável à sua esquerda o valor calculado à direita. O C permite que se façam atribuições encadeadas, como:

```
x = y = z = 2.45;
```

Nesta linha, o valor 2.45 será atribuído a **z**, depois a **y**, depois a **x**.

3.4.2 Operadores Aritméticos

Os operadores aritméticos são usados para desenvolver operações matemáticas. A Tabela 3.3 apresenta a lista dos operadores aritméticos do C.

Tabela 3.3: Operadores aritméticos

Operador	Ação
+	Soma (inteira e ponto flutuante)
-	Subtração ou Troca de sinal (inteira e ponto flutuante)
*	Multiplicação (inteira e ponto flutuante)
/	Divisão (inteira e ponto flutuante)
%	Resto de divisão (de inteiros)
++	Incremento (inteiro e ponto flutuante)
--	Decremento (inteiro e ponto flutuante)

O C possui operadores unários e binários. Os unários agem sobre uma variável apenas, modificando ou não o seu valor, e retornam o valor final da variável. Os binários usam duas variáveis e retornam um terceiro valor, sem alterar as variáveis originais. A soma é um operador binário pois pega duas variáveis, soma seus valores, sem alterar as variáveis, e retorna esta soma. Outros operadores binários são os operadores `-` (subtração), `*`, `/` e `%`. O operador `-` como troca de sinal é um operador unário que não altera a variável sobre a qual é aplicado, pois ele retorna o valor da variável multiplicado por **-1**.

Os operadores de incremento e decremento são unários que alteram a variável sobre a qual estão aplicados. O que eles fazem é incrementar ou decrementar de um a variável sobre a qual estão aplicados. Então

```
x++;
x--;
```

são operações equivalentes a

```
x = x+1;
x = x-1;
```

Estes operadores podem ser pré-fixados ou pós-fixados. A diferença é que quando são pré-fixados eles incrementam e retornam o valor da variável já incrementada. Quando são pós-fixados eles retornam o valor da variável sem o incremento e depois incrementam a variável. Então, em

```
x = 23;
y = x++;
```

teremos, no final, y = 23 e x = 24. Em

```
x = 23;
y = ++x;
```

teremos, no final, y = 24 e x = 24.

Uma curiosidade: a linguagem de programação C++ tem este nome pois ela seria um "incremento" da linguagem C padrão.

Os operadores `*`, `/`, `+` e `-` funcionam como na maioria das linguagens, o operador `%` indica o resto de uma divisão inteira. Quando efetuados sobre um mesmo operando, podem ser utilizados em forma reduzida, como ilustra a Tabela 3.4.

Tabela 3.4: Comandos de atribuição reduzidos

`i += 2;`	$\longrightarrow$	`i = i+2;`
`x *= y+1;`	$\longrightarrow$	`x = x*(y+1);`
`d -= 3;`	$\longrightarrow$	`d = d - 3;`

A Figura 3.4 apresenta um programa simples que utiliza operadores aritméticos sobre números inteiros, e a Figura 3.5 apresenta um programa para fazer a conversão de medidas em pés para metros, utilizando operações sobre números em ponto flutuante.

A função `scanf( )` é utilizada para ler um valor numérico do teclado do computador.

3.4.3 Operadores relacionais e lógicos

O termo **relacional** refere-se às relações que os valores podem ter um com o outro e o termo **lógico** se refere às maneiras como essas relações podem ser conectadas. Em C, entende-se como verdadeiro qualquer valor que não seja 0, enquanto que o valor 0 indica

```
# include <stdio.h>
main()
{
   int x, y;
   x =10; y = 3;
   printf("%d\n", x/y);
   printf("%d\n", x%y);
}
```

Figura 3.4: Programa com operadores aritméticos

```
#include <stdio.h>
/* Conversão de pés para metros. */
main()
{
   float pes, metros;
   printf("Informe o número de pés: ");
   scanf("%f", &pes);              /* lê um float */
   metros = pes * 0.3048;          /* conversão de pés para metros */
   printf("%f pes é %f metros\n", pes, metros);
}
```

Figura 3.5: Programa para conversão de pés para metros

condição falsa. As expressões que usam operadores de relação e lógicos retornarão **0** para **falso** e **1** para **verdadeiro**. Tanto os operadores de relação como os lógicos têm a precedência menor que os operadores aritméticos. As operações de avaliação produzem um resultado 0 ou 1.

Tabela 3.5: Operadores relacionais e lógicos

relacionais		**lógicos**	
>	maior que	&&	and (E)
>=	maior ou igual	||	or (OU)
<	menor	!	not (NÃO)
<=	menor ou igual		
==	igual		
!=	não igual		

As Figuras 3.6 e 3.7 apresentam programas com exemplos de utilização de operadores relacionais e lógicos.

```
# include <stdio.h>
main()
{
   int i,j;
   printf("digite dois números: ");
   scanf("%d%d", &i, &j);
   printf("%d == %d é %d\n", i, j, i==j);
   printf("%d != %d é %d\n", i, j, i!=j);
   printf("%d <= %d é %d\n", i, j, i<=j);
   printf("%d >= %d é %d\n", i, j, i>=j);
   printf("%d < %d é %d\n", i, j, i<j);
   printf("%d > %d é %d\n", i, j, i> j);
}
```

Figura 3.6: Programa com operadores relacionais e lógicos

```
/* Maior de Dois */
# include <stdio.h>
main( )
{
   int x=2, y=3, produto;
   if ((produto = x*y) > 0) printf("é maior");
}
```

Figura 3.7: Programa Maior_de_dois

3.4.4 Precedência

O nível de precedência dos operadores é avaliado da esquerda para a direita. Os parênteses podem ser utilizados para alterar a ordem da avaliação.

3.4.5 Operador cast (modelador)

Sintaxe: (tipo)expressão

Podemos forçar uma expressão a ser de um determinado tipo usando o operador **cast**. Um exemplo pode ser visto na Figura 3.8. Nesse exemplo, se não fosse utilizado o

Tabela 3.6: Nível de precedência dos operadores

Mais alta	! ++ -- -(unário)
	* / %
	+ -
	< >
	<= >=
	== !=
	&&
	||
Mais baixa	=

modelador (float) sobre a variável i, o programa imprimiria “1/3 = 0”, pois, ele faria primeiro a divisão de inteiros (1/3) resultando 0, e depois a conversão do resultado para float.

```
# include <stdio.h>
main( )
{
   int i=1;
   printf(" %d/3 é: %f ", i, (float) i/3);
}
```

Figura 3.8: Programa com exemplo de utilização de operador **cast**

3.4.6 Operador sizeof

O operador **sizeof** retorna o tamanho em **bytes** da variável, ou seja, do tipo que está em seu operando. É utilizado para assegurar a portabilidade do programa. O operador **sizeof** é usado para se saber o tamanho de variáveis ou de tipos. Ele retorna o tamanho do tipo ou variável em **bytes**. Mas porque usá-lo se sabemos, por exemplo, que um inteiro ocupa 2 bytes? Devemos usá-lo para garantir portabilidade. O tamanho de um inteiro pode depender do sistema para o qual se está compilando. O **sizeof** é chamado um operador porque ele é substituído pelo tamanho do tipo ou variável no momento da compilação. Ele não é uma função. O **sizeof** admite duas formas:

```
sizeof nome_da_variável
 sizeof (nome_do_tipo)
```

Se quisermos então saber o tamanho de um **float** fazemos **sizeof(float)**. Se declararmos a variável **f** como **float** e quisermos saber o seu tamanho faremos **sizeof(f)**. O operador **sizeof** também funciona com **estruturas**, **campos bit**, **uniões** e **enumerações**.

4

FUNÇÕES BÁSICAS DA BIBLIOTECA C

Existem algumas funções muito usadas em programas feitos em linguagem C. Estas são utilizadas principalmente para estabelecer a comunicação entre o usuário e o computador. Aqui serão citadas apenas algumas mais comuns.

4.1 FUNÇÃO PRINTF( )

Sintaxe: `printf("expressão de controle",argumentos);`

É uma função de **I/O**[1], que permite escrever no dispositivo padrão (tela). A expressão de controle pode conter caracteres que serão exibidos na tela e os códigos de formatação que indicam o formato em que os argumentos devem ser impressos. Cada argumento deve ser separado por vírgula.

Tabela 4.1: Caracteres de controle da função printf

`\n`	nova linha	`%c`	caractere simples
`\t`	tab	`%d`	decimal
`\b`	retrocesso	`%e`	notação científica
`\"`	aspas	`%f`	ponto flutuante
`\\`	Barra invertida	`%o`	octal
`\f`	salta formulário	`%s`	cadeia de caracteres
`\0`	nulo	`%u`	decimal sem sinal
		`%x`	hexadecimal

As Figuras 4.1, 4.2, 4.3, 4.4, 4.5 e 4.6 apresentam vários exemplos de utilização da função printf em programas simples.

[1] *Input/Output:* entrada/saída

```
#include <stdio.h>
main( )
{
   printf("Este é o numero dois: %d", 2);
   printf("%s está a %d milhões de milhas \n do sol", "Vênus", 67);
}
```

Figura 4.1: Exemplo de utilização da função `printf`

```
#include <stdio.h>
main( )
{
   printf("\n%2d",350);
   printf("\n%4d",350);
   printf("\n%6d",350);
}
```

Figura 4.2: Tamanho de campos na impressão

```
#include <stdio.h>
main( )
{
   printf("\n%4.2f", 3456.78);
   printf("\n%3.2f", 3456.78);
   printf("\n%3.1f", 3456.78);
   printf("\n%10.3f", 3456.78);
}
```

Figura 4.3: Arredondamento de números em ponto flutuante

```
#include <stdio.h>
main( )
{
   printf("\n%10.2f %10.2f %10.2f", 8.0, 15.3, 584.13);
   printf("\n%10.2f %10.2f %10.2f", 834.0, 1500.55, 4890.21);
}
```

Figura 4.4: Alinhamento de campos numéricos

```
#include <stdio.h>
main( )
{
   printf("\n%04d", 21);
   printf("\n%06d", 21);
   printf("\n%6.4d", 21);
   printf("\n%6.0d", 21);
}
```

Figura 4.5: Complementando com zeros à esquerda

```
#include <stdio.h>
main( )
{
   printf("%d %c %x %o\n", 'A', 'A', 'A', 'A');
   printf("%c %c %c %c\n", 'A', 65, 0x41, 0101);
}
```

Figura 4.6: Formas para imprimir caracteres utilizando a função `printf`

A tabela ASCII possui 256 códigos de 0 a 255, se imprimirmos em formato caractere um número maior que 255, será impresso o resto da divisão do número por 256; se o número for 3393 será impresso A, pois o resto de 3393 por 256 é 65.

4.2 FUNÇÃO SCANF( )

Também é uma função de **I/O** implementada em todos os compiladores C. Ela é o complemento de **printf()** e nos permite ler dados formatados da entrada padrão (teclado). Sua sintaxe é similar a **printf()**.

```
scanf("expressão de controle", argumentos);
```

A lista de argumentos deve consistir nos endereços das variáveis. C oferece um operador para tipos básicos chamado operador de endereço e referenciado pelo símbolo **&** que retorna o endereço do operando. Um exemplo de programa que utiliza a função scanf é visto na Figura 4.7.

Operador de endereço &:

A memória do computador é dividida em **bytes**, e são numerados de 0 até o limite da memória. Estas posições são chamadas de endereços. Toda variável ocupa uma certa localização na memória, e seu endereço é o primeiro **byte** ocupado por ela.

```
#include <stdio.h>
main( )
{
   int num;
   printf("Digite um número: ");
   scanf("%d",&num);
   printf("\no número é %d",num);
   printf("\no endereço e %u",&num);
}
```

Figura 4.7: Exemplo de programa que utiliza a função scanf

4.3 FUNÇÃO GETCHAR()

É a função original de entrada de caractere dos sistemas baseados em UNIX. A função **getchar()** armazena a entrada até que a tecla **ENTER** seja pressionada. Veja um exemplo na Figura 4.8.

```
#include <stdio.h>
main( )
{
   char ch;
   ch = getchar();
   printf("%c \n",ch);
}
```

Figura 4.8: Exemplo de programa que utiliza a função `getchar`

4.4 FUNÇÃO PUTCHAR()

Essa função escreve na tela o argumento de seu caractere na posição corrente. Veja um exemplo na Figura 4.9.

Há inúmeras outras funções de manipulação de char complementares às que foram vistas, como **isalpha()**, **isupper()**, **islower()**, **isdigit()**, **isspace()**, **toupper()**, **tolower()**.

```
# include <stdio.h>
main( )
{
   char ch;
   printf("digite uma letra minúscula : ");
   ch = getchar( );
   putchar(toupper(ch));
   putchar('\n');
}
```

Figura 4.9: Exemplo de programa que utiliza a função `putchar`

5

ESTRUTURAS DE CONTROLE DE FLUXO

Os comandos de controle de fluxo são a essência de qualquer linguagem, porque governam o fluxo da execução do programa. São poderosos e ajudam a explicar a popularidade da linguagem. Podemos dividir em três categorias. A primeira consiste em instruções condicionais **if** e **switch**. A segunda são os comandos de controle de **loop**, o **while**, o **for** e o **do-while**. A terceira contém instruções de desvio incondicional **goto**.

5.1 IF

Sintaxe:

```
if (condição) comando;
else comando;
```

Se a condição avaliar em verdadeiro (qualquer valor diferente de 0), o computador executará o comando ou o bloco, de outro modo, se a cláusula **else** existir, o computador executará o comando ou o bloco que é seu objetivo. Veja os exemplos das Figuras 5.1 e 5.2.

```
#include <stdio.h>
main( )
{
   int a,b;
   printf("digite dois números:");
   scanf("%d%d",&a,&b);
   if (b) printf("%d\n",a/b);
   else printf("divisão por zero\n");
}
```

Figura 5.1: Exemplo de programa que utiliza o comando de decisão if

No exemplo da Figura 5.1, são pedidos dois números inteiros, **a** e **b**, e depois será impresso o valor **a/b** somente se **b** for diferente de zero. Se **b** for igual a zero, será indicada condição falsa para o comando if, e será executado o comando alternativo (else), sendo impressa a mensagem de erro.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
main( )
{
   int num,segredo;
   srand(time(NULL));
   segredo = rand( )/100;
   printf("Qual e o numero: ");
   scanf("%d",&num);
   if (segredo == num)
   {
      printf("Acertou!");
      printf("\nO numero e %d\n", segredo);
   }
   else printf ("Errou, tente outra vez! \n");
}
```

Figura 5.2: O comando de decisão if com um bloco de instruções

Na Figura 5.1, havia apenas uma instrução a ser executada após a verificação do comando if, portanto, não era necessário utilizar-se chaves. Na Figura 5.2, havia um bloco de dois comandos a serem executados após o if, portanto, este conjunto de instruções deve ser escrito dentro de um bloco que inicia com abre-chaves e finaliza com fecha-chaves.

5.2 IF-ELSE-IF

Sintaxe:

```
if (condição) comando;
else if (condição) comando;
```

Uma variável é testada sucessivamente contra uma lista de variáveis inteiras ou de caracteres. Depois de encontrar uma coincidência, o comando ou o bloco de comandos é executado.

Em uma estrutura **if-else-if** são colocadas várias opções que serão testadas uma de cada vez, começando pela **1** e continuando a testar até que seja encontrada uma expressão cujo resultado seja diferente de **zero**. Neste caso o programa executa a declaração correspondente. Só uma declaração será executada, ou seja, só será executada a declaração equivalente à primeira condição que for diferente de zero. A última declaração (**default**) é a que será executada no caso de todas as condições serem falsas e é opcional.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
main( )
{
   int num,segredo;
   srand(time(NULL));
   segredo = rand( )/100;
   printf("Qual e o numero: ");
   scanf("%d", &num);
   if (segredo == num)
   {
      printf("Acertou!");
      printf("\nO numero e %d\n", segredo);
   }
   else if (segredo < num) printf ("Errado, muito alto!\n");
   else printf ("Errado, muito baixo!\n");
}
```

Figura 5.3: Exemplo de um programa que utiliza o comando if-else-if

5.3 OPERADOR TERNÁRIO

Sintaxe: `condição ? expressão1 : expressão2`

É uma maneira compacta de expressar **if-else**. A operação **?** funciona da seguinte forma: a condição é testada, se for verdadeira será calculada a **expressão1**, se for falsa, calcula-se a **expressão2**.

No exemplo apresentado na Figura 5.4, a variável **max** terá o valor de **x** se **x** for maior que **y**, caso contrário, **max** assumirá o valor de **y**.

```
#include <stdio.h>
main( )
{
   int x, y, max;
   printf("Entre com dois números: ");
   scanf("%d, %d", &x, &y);
   max = (x>y) ? x : y;
   printf("max = %d\n", max);
}
```

Figura 5.4: Exemplo de programa que utiliza o operador ternário

5.4 SWITCH

Sintaxe:

```
switch(variável){
   case constante1:
      seqüência de comandos;
      break;
   case constante2:
      seqüência de comandos;
      break;
   default:
      seqüência de comandos;
}
```

Uma variável é testada sucessivamente contra uma lista de variáveis inteiras ou de caracteres. Depois de encontrar uma coincidência, o comando ou o bloco de comandos é executado.

Se nenhuma coincidência for encontrada o comando **default** será executado. O **default** é opcional. A seqüência de comandos é executada até que o comando **break** seja encontrado. A Figura 5.5 apresenta um exemplo de uso do **switch**.

5.5 LOOP FOR

Sintaxe: `for (inicialização; condição; incremento) comando;`

O comando **for** é de alguma maneira encontrado em todas linguagens procedurais de programação. Este comando é normalmente utilizado para executar repetidamente um

```
#include <stdio.h>
main ( )
{
   char x;
   printf("1. Inclusão \n");
   printf("2. Alteração \n");
   printf("3. Exclusão \n");
   printf(" Digite sua opção: ");
   x = getchar( );
   switch(x) {
   case '1':
      printf("escolheu inclusão\n");
      break;
   case '2':
      printf("escolheu alteração\n");
      break;
   case '3':
      printf("escolheu exclusão\n");
      break;
   default:
      printf("opção inválida\n");
   }
}
```

Figura 5.5: Exemplo de programa que utiliza o comando switch

conjunto de comandos por um número pré determinado de vezes. As Figuras 5.6 e 5.7 apresentam dois exemplos simples de utilização do comando de repetição **for**.

O **for** é o primeiro de uma série de três comandos para se trabalhar com **loops** (laços de repetição). Os outros são o **while** e o **do-while**. Os três compõem a segunda família de comandos de **controle de fluxo**. Podemos pensar nesta família como sendo a dos comandos de repetição controlada.

Em sua forma mais simples, a incialização é um comando de atribuição que o compilador usa para estabelecer a variável de controle do **loop**. A condição é uma expressão de relação que testa a variável de controle do **loop** contra algum valor para determinar quando o **loop** terminará. O incremento define a maneira como a variável de controle do **loop** será alterada cada vez que o computador repetir a seqüência de comandos.

O comando **for** executa a inicialização incondicionalmente e testa a condição. Se a condição for falsa ele não faz mais nada. Se a condição for verdadeira ele executa a

```
#include <stdio.h>
main( )
{
   int x;
   for(x=1; x<100; x++) printf("%d\n",x);
}
```

Figura 5.6: Exemplo de for com apenas um comando

```
#include <stdio.h>
main()
{
   int x, xq;
   printf ("\n\t   Numero    Quadrado \n\n");
   for(x=1; x<100; x++)
   {
      xq = x*x;
      printf("\t  %d       %d \n", x, xq);
   }
}
```

Figura 5.7: Exemplo de for com um bloco de instruções

instrução (no primeiro exemplo **printf**) ou um bloco de instruções, faz o incremento e volta a testar a condição. Ele fica repetindo estas operações até que a condição seja falsa.

O primeiro exemplo mostra um **loop for** que executa apenas um comando, e o segundo exemplo executa um bloco com dois comandos, tendo, portanto, que ser escrito entre chaves.

Uma aplicação muito utilizada para laços de repetição é para o cálculo de séries numéricas. Por exemplo, o número π pode ser calculado utilizando-se a série abaixo.

$$S = 1 - \frac{1}{3^3} + \frac{1}{5^3} - \frac{1}{7^3} + \cdots$$

Esta série é calculada para os N primeiros inteiros, sendo N um número determinado pelo usuário. Quanto maior N, melhor a precisão obtida. Depois calcular o valor de S, calcula-se π:

$$\pi = \sqrt[3]{32 \times S}$$

O programa da Figura 5.8 apresenta uma forma de se calcular o valor de π para um número N de termos[1].

```
#include <stdio.h>
#include <math.h>
main( )
{
   int N, n, sinal;
   float S, x;
   printf(" \n Envie o numero de termos ");
   scanf(" %d ",&N);
   S = 1;
   x = 1;
   sinal = 1;
   for (n = 1; n < N; n++)
   {
      x +=2;
      sinal =  sinal;
      S = S + (float)sinal/(x*x*x);
   }
   S = exp((1./3)*log(32*S));
   printf("\n\n O valor de PI eh %f, para os %d", S, N);
   printf("primeiros termos \n");
}
```

Figura 5.8: Programa para calcular o valor de π

No exemplo apresentado na Figura 5.8, as funções **exp()** e **log()** fazem parte da biblioteca **math.h** e são utilizadas para se calcular uma exponencial b^e. Neste caso **b = 32*S** e **e = 1./3**. Observe que é necessário colocar-se o ponto depois de **1**, na fração, para que o resultado seja um número em ponto flutuante, caso contrário será calculado o quociente inteiro de **1/3** que resulta no valor zero.

A linguagem C permite que se utilize mais de uma variável no controle do laço de repetição **for**. É permitido também omitir-se qualquer um dos elementos do **for**, isto é, se não quisermos uma inicialização, por exemplo, poderemos omiti-la. Vejamos os exemplos das Figuras 5.9 e 5.10.

[1]Versões recentes do GCC requerem que o parâmetro `-lm` seja passado quando o programa utiliza a biblioteca `math.h`: `gcc -lm pi.c -o pi`

```
#include <stdio.h>
main( )
{
   int x,y;
   for (x = 0, y = 0; x+y<100; ++x,++y) printf("%d ", x+y);
}
```

Figura 5.9: Laço de for com duas variáveis de controle

Um uso interessante para o **for** é o **loop** infinito, como nenhuma das três definições são obrigatórias, podemos deixar a condição em aberto, como visto na Figura 5.10. Outra forma usual do **for** é o **for** aninhado, ou seja, um **for** dentro de outro, como visto nas Figuras 5.11 e 5.12.

```
#include <stdio.h>
main( )
{
   for ( ; ; ) printf ("loop infinito\n");
}
```

Figura 5.10: Loop infinito

```
#include <stdio.h>
main( )
{
   int linha,coluna;
   for(linha=1; linha<=24; linha++)
   {
      for(coluna=1; coluna<40; coluna++)  printf("-");
      putchar('\n');
   }
}
```

Figura 5.11: Laços de for aninhados

```
#include<stdio.h>
main( )
{
   int i, j, k, temp;
   printf("\t i      i^2      i^3      i^4 \n");
   for ( i=1;i<10; i++)  /* laço externo: define o numero base */
   {
      for ( j=1; j<5; j++)   /* primeiro nível do aninhamento */
      {
         temp = 1;
         for ( k=0; k<j; k++) /* laço mais interno: eleva a j */
         {
            temp = temp*i;
         }
         printf("%9d",temp);
      }
      printf("\n");
   }
}
```

Figura 5.12: Programa que exibe uma tabela das quatro primeiras potências dos números de 1 a 9

5.6 WHILE

Sintaxe: `while(condição) comando;`

Uma maneira possível de executar um laço é utilizando o comando **while**. Ele permite que o código fique sendo executado numa mesma parte do programa de acordo com uma determinada condição.

- o comando pode ser vazio, simples ou bloco
- ele é executado desde que a condição seja verdadeira
- testa a condição antes de executar o laço

As Figuras 5.13 e 5.14 apresentam exemplos de utilização do laço de repetição **while**.

```
# include <stdio.h>
main( )
{
   char ch;
   while (ch != 'a') ch = getchar( );
}
```

Figura 5.13: Exemplo simples de while

```
#include <stdio.h>
main( )
{
   int x, xq;
   printf ("\n\t   Numero    Quadrado \n\n");
   x = 1;
   while (x < 100)
   {
      xq = x*x;
      printf("\t  %d       %d \n", x, xq);
      x++;
   }
}
```

Figura 5.14: Geração de uma tabela de quadrados de inteiros utilizando while

5.7 DO-WHILE

Sintaxe:

```
do
{comando; }
while(condição);
```

Também executa comandos repetitivos, mas neste caso, a condição só será testada depois que o conjunto de instruções tiver sido executado pelo menos uma vez. As Figuras 5.15 e 5.16 apresentam exemplos de utilização do laço **do-while**.

A principal diferença entre os comandos while e do-while é que no segundo o conjunto de instruções do bloco deverá ser executado pelo menos uma vez, obrigatoriamente, enquanto no primeiro (while) pode acontecer do bloco de instruções não ser executado nenhuma vez.

```
#include <stdio.h>
main( )
{
   int x, xq;
   printf ("\n\t   Numero    Quadrado \n\n");
   x = 1;
   do {
      xq = x*x;
      printf("\t  %d       %d \n", x, xq);
      x++;
   }
   while (x <100);
}
```

Figura 5.15: Geração de uma tabela de quadrados de inteiros utilizando do-while

5.8 BREAK

Quando o comando **break** é encontrado em qualquer lugar do corpo do **for**, ele causa seu término imediato. O controle do programa passará então imediatamente para o código que segue o **loop**. Veja o exemplo da Figura 5.17.

5.9 CONTINUE

Algumas vezes torna-se necessário “saltar” uma parte do programa, para isso utilizamos o **continue**. O comando **continue** força a próxima iteração do **loop** — pula o código que estiver em seguida. Um exemplo é apresentado na Figura 5.18.

```
# include <stdio.h>
main( )
{
   char ch;
   printf("1. inclusão\n");
   printf("2. alteração\n");
   printf("3. exclusão\n");
   printf("4. sair\n");
   printf(" Digite sua opção:");
   do {
      ch=getchar();
      switch(ch)
      {
         case '1': printf("escolheu inclusao\n"); break;
         case '2': printf("escolheu alteracao\n"); break;
         case '3': printf("escolheu exclusao\n"); break;
         case '4': printf("sair\n");
      }
   }
   while(ch != '1' && ch != '2' && ch != '3' && ch != '4');
}
```

Figura 5.16: Exemplo de utilização do laço do-while

```
main( )
{
   char ch;
   for( ; ; )
   {
      ch = getchar( );
      if (ch == 'a') break;
   }
}
```

Figura 5.17: Exemplo simples de utilização do comando break

```
# include <stdio.h>
main( )
{
   int x;
   for(x = 0; x<100; x++)
   {
      if (x%2) continue;
      printf ("%d\n", x);
   }
}
```

Figura 5.18: Exemplo de utilização do comando continue

6

MATRIZES

A matriz é um tipo de dado usado para representar uma certa quantidade de variáveis que são referenciadas pelo mesmo nome. Consiste em locações contíguas de memória. O endereço mais baixo corresponde ao primeiro elemento. A matriz é um conjunto ordenado de dados que possuem o mesmo tipo.

6.1 MATRIZ UNIDIMENSIONAL

Sintaxe: `tipo nome[tamanho];`

As matrizes têm 0 como índice do primeiro elemento, portanto sendo declarada uma matriz de inteiros de 10 elementos, o índice varia de 0 a 9.

Quando o compilador C encontra uma declaração de matriz ele reserva um espaço na memória do computador suficientemente grande para armazenar o número de células especificadas em tamanho. Por exemplo, se declararmos:

```
float exemplo[20];
```

o C irá reservar $4 \times 20 = 80$ bytes. Estes bytes são reservados de maneira contígua. Neste exemplo, os dados serão indexados de 0 a 19. Para acessá-los escrevemos:

```
exemplo[0]
exemplo[1]
.
.
exemplo[19]
```

Mas ninguém o impede de escrever:

```
exemplo[30]
exemplo[103]
```

Por quê? Porque o C não verifica se o índice que você usou está dentro dos limites válidos. Este é um cuidado que o programador deve tomar. Se o programador não tiver atenção com os limites de validade para os índices ele corre o risco de ter dados sobrescritos ou de ver o computador travar. Vários erros sérios (*bugs*) terríveis podem surgir. As Figuras 6.1, 6.2 e 6.3 apresentam programas que utilizam matrizes unidimensionais.

```
#include <stdio.h>
main( )
{
   int x[10];
   int t;
   for(t=0;t<10;t++)
   {
      x[t] = t*2;
      printf ("%d\n", x[t]);
   }
}
```

Figura 6.1: Exemplo simples de utilização de matriz

6.2 MATRIZ MULTIDIMENSIONAL

Sintaxe: `tipo nome[tamanho][tamanho] ...;`

A matriz multidimensional funciona como a matriz de uma dimensão (vetor), mas tem mais de um índice. As dimensões são declaradas em seqüência entre colchetes. Veja um exemplo na Figura 6.4.

6.3 MATRIZES ESTÁTICAS

Os vetores de dados podem ser inicializados como os dados de tipos simples, mas somente como variáveis globais. Quando for inicializar uma matriz local sua classe deve ser **static**. Veja um exemplo na Figura 6.5.

6.4 LIMITES DAS MATRIZES

A verificação de limites não é feita pela linguagem, nem mensagem de erros são enviadas, o programa tem que testar os limites das matrizes. Na Figura 6.6 é apresentado

```
#include <stdio.h>
main ( )
{
   int num[100];
   int count=0;
   int totalnums;
   do
   {
      printf ("\nEntre com um numero (-999 p/ terminar): ");
      scanf ("%d",&num[count]);
      count++;
   }
   while (num[count-1] != -999);
       /* verifica quantos números foram digitados */
   totalnums = count -1;
       /* retira a contagem do número 999 */
   printf ("\n\n\n\t Os números que você digitou foram:\n\n");
   for (count = 0; count < totalnums; count++)
      printf (" %d", num[count]);
}
```

Figura 6.2: Exemplo de armazenamento de dados em matriz unidimensional

```
#include <stdio.h>
main( )
{
   int notas[5], i, soma;
   for (i = 0; i<5; i++)
   {
      printf("Digite a nota do aluno %d: ", i );
      scanf("%d", &notas[i] );
   }
   soma = 0;
   for( i = 0; i<5; i++) soma = soma + notas[i];
   printf("Media das notas: %d.", soma/5);
}
```

Figura 6.3: Operações com matriz

```
# include <stdio.h>
main ( )
{
   int x[10][10];
   int t, p=0;
   for( t = 0; t<10; t++,p++)
   {
      x[t][p] = t*p;
      printf("%d\n", x[t][p] );
   }
}
```

Figura 6.4: Exemplo simples de utilização de matriz multidimensional

```
#include <stdio.h>
main( )
{
   int i;
   static int x[10] ={0,1,2,3,4,5,6,7,8,9};
   for(i=0; i<10; i++) printf("%d\n", x[i] );
}
```

Figura 6.5: Exemplo de utilização de matriz estática

um programa que causará problemas ao computador durante sua execução porque o índice da matriz **erro**, dentro do laço de for, excederá o tamanho da matriz (que é 10).

```
# include <stdio.h>
main( )
{
   int erro[10], i;
   for( i = 0; i<100; i++)
   {
      erro[i]=1;
      printf (" %d\n ", erro[i] );
   }
}
```

Figura 6.6: Erro na utilização de matriz

7

MANIPULAÇÃO DE STRINGS

Em C não existe um tipo de dado **string**, no seu lugar é utilizado uma matriz de caracteres. Uma **string** é uma matriz tipo **char** que termina com **'\0'**. Por essa razão uma **string** deve conter uma posição a mais do que o número de caracteres que se deseja. O caractere **'\0'** tem o código numérico **00**, portanto pode-se verificar o final de uma **string** procurando o valor numérico **zero** armazenado na matriz de caracteres. Constantes **strings** são uma lista de caracteres que aparecem entre aspas, não sendo necessário colocar o **'\0'**, que é colocado pelo compilador. A Figura 7.1 mostra um programa simples para operação com strings.

```
#include <stdio.h>
#include <string.h>
main( )
{
   static char re[] = "lagarto";
      // um vetor de caracteres de 8 posições
   puts(re);
   puts(&re[0]);
      // uma variável do tipo matriz (ou vetor ) é referenciada
   putchar('\n');
      // como o endereço de sua primeira posição.
}
```

Figura 7.1: Exemplo simples de operação com strings

7.1 FUNÇÃO GETS()

Sintaxe: `gets(nome_matriz);`

É utilizada para leitura de uma **string** através do dispositivo padrão, até que a tecla <ENTER> seja pressionada. A função **gets()** não testa limites na matriz em que é chamada. A Figura 7.2 apresenta um programa simples que utiliza a função **gets**.

```
# include <stdio.h>
main()
{
   char str[80];
   gets(str);
   printf("%s",str);
}
```

Figura 7.2: Programa simples que utiliza a função gets

7.2 FUNÇÃO PUTS()

Sintaxe: `puts(nome_do_vetor_de_caracteres);`

Escreve o seu argumento no dispositivo padrão de saída (vídeo), coloca um **'\n'** no final. Reconhece os códigos de barra invertida. A Figura 7.3 apresenta um programa simples que utiliza a função **puts**.

```
# include <stdio.h>
# include <string.h>
main()
{
   puts("mensagem");
}
```

Figura 7.3: Exemplo simples de utilização da função puts

7.3 FUNÇÃO STRCPY()

Sintaxe: `strcpy(destino, origem);`

Esta função faz parte da biblioteca **string.h**. A função **strcpy()** copia o conteúdo de uma **string** para uma variável tipo **string** (um vetor de **char**). No programa da Figura 7.4, a string “alo” será copiada para a variável (matriz de caracteres) **str**.

```
# include <stdio.h>
# include <string.h>
main()
{
   char str[20];
   strcpy(str,"alo");
   puts(str);
}
```

Figura 7.4: Utilização da função strcpy da biblioteca string.h

7.4 FUNÇÃO STRCAT()

Sintaxe: `strcat(string1, string2);`

Concatena duas strings. Não verifica tamanho. Um exemplo é visto na Figura 7.5.

```
#include <string.h>
#include <stdio.h>
main()
{
   char um[20], dois[10];
   strcpy (um, "bom");
   strcpy (dois," dia");
   strcat (um, dois);      // une a string dois à string um
   printf ("%s\n", um);
}
```

Figura 7.5: Utilização da função strcat da biblioteca string.h

7.5 FUNÇÃO STRCMP()

Sintaxe: `strcmp(s1, s2);`

Essa função compara duas strings, se forem iguais devolve 0. A Figura 7.6 mostra um exemplo da função **strcmp()**.

```
main( )
{
   char s[80];
   printf("Digite a senha:");
   gets(s);
   if (strcmp(s,"laranja")) printf("senha inválida\n");
   else printf("senha ok!\n") ;
}
```

Figura 7.6: Utilização da função strcmp da biblioteca string.h

8

PONTEIROS

Sintaxe: `tipo *nomevar;`

Para ser um bom programador em linguagem C é fundamental que se tenha um bom domínio de **ponteiros**. Por isto, o leitor deve acompanhar com atenção esta parte do curso que trata deles. **Ponteiros** são tão importantes na linguagem C que você já os viu e nem percebeu, quando falamos da função **scanf**. Em **scanf("%d", &num);** o operador **&** indica o endereço de memória onde o compilador alocou a variável **num** ao ler a sua declaração no início do programa.

O **ponteiro** é um dos aspectos mais fortes e poderosos e perigosos da linguagem C. É comum, o programador utilizar o **ponteiro** incorretamente, ocasionando erros que são muito difíceis de encontrar.

Ponteiro é uma variável que contém o endereço de outra variável. Os **ponteiros** são utilizados para alocação dinâmica, podendo substituir matrizes com mais eficiência. Também fornecem a maneira pelas quais funções podem modificar os argumentos chamados, como veremos no capítulo de funções.

8.1 DECLARANDO PONTEIROS

Para declarar um ponteiro temos a seguinte forma geral:

```
tipo_do_ponteiro *nome_da_variável;
```

É o asterisco (*) que faz o compilador saber que aquela variável não vai guardar um valor mas sim um endereço para aquele tipo especificado. Vamos ver exemplos de declarações:

```
int *pt;
char *temp, *pt2;
```

O primeiro exemplo declara um ponteiro para um inteiro. O segundo declara dois ponteiros para caracteres. Eles ainda não foram inicializados (como toda variável do C

que é apenas declarada). Isto significa que eles apontam para um lugar indefinido. Este lugar pode estar, por exemplo, na porção da memória reservada ao sistema operacional do computador. Usar o ponteiro nestas circunstâncias pode levar a um travamento do micro, ou a algo pior. O ponteiro deve ser inicializado (apontado para algum lugar conhecido) antes de ser usado! Isto é de suma importância!

Para atribuir um valor a um ponteiro recém-criado poderíamos igualá-lo a um valor de memória. Mas, como saber a posição na memória de uma variável do nosso programa? Seria muito difícil saber o endereço de cada variável que usamos, mesmo porque estes endereços são determinados pelo compilador na hora da compilação e relocados na execução. Podemos então deixar que o compilador faça este trabalho por nós. Para saber o endereço de uma variável basta usar o operador **&**. Veja o exemplo:

```
int count;
int *pt;
pt = &count;
```

Declaramos a variável **count** como sendo do tipo **inteiro** e um declaramos um **ponteiro** para um inteiro, **pt**. A expressão **&count** nos dá o **endereço** de **count**, o qual armazenamos em **pt**. O valor de **count** não é alterado quando se inicializa o valor de **pt**.

Como nós colocamos um endereço em **pt**, ele está agora "liberado" para ser usado. Podemos, por exemplo, alterar o valor de **count** usando **pt**. Para tanto vamos usar o operador "inverso" do operador **&**. É o operador *. No exemplo acima, uma vez que fizemos **pt=&count** a expressão ***pt** é equivalente ao próprio **count**. Após a inicialização da variável **ponteiro** com o endereço da variável inteira, esta variável pode ser referenciada pelo seu nome ou pelo ponteiro que contém seu endereço. Isto significa que, se quisermos atribuir um valor para **count** podemos fazer de duas formas:

```
count = 5;
```

ou

```
*pt = 5;
```

As duas formas são equivalentes e têm o mesmo resultado. Qualquer modificação feita utilizando-se ***pt**, causará uma modificação na variável **count**.

8.2 MANIPULAÇÃO DE PONTEIROS

Desde que os ponteiros são variáveis, eles podem ser manipulados tal como as variáveis. Se **py** e **px** são **ponteiros** para **inteiros**, então podemos fazer a declaração:

```
py = px;
```

A Figura 8.1 apresenta um programa que exemplifica a utilização de ponteiros para acessar um variável **x**.

```
#include <stdio.h>
main( )
{
   int x,*px,*py;
   x = 9;
   px = &x;
   py = px;
   printf("x= %d\n",x);
      // imprime o valor de x
   printf("&x= %d\n",&x);
      // endereço da variável x, que é igual ao conteúdo de px
   printf("px= %d\n",px);
      // valor de px, que é o endereço de x
   printf("*px= %d\n",*px);
      // conteúdo da variável apontada por px, isto é, valor de x
   printf("*py= %d\n",*py);
      // imprime valor de x, pois py = px
}
```

Figura 8.1: Utilização de ponteiros

8.3 EXPRESSÕES COM PONTEIROS

Os ponteiros podem aparecer em expressões, se **px** aponta para um inteiro **x**, então ***px** pode ser utilizado em qualquer lugar que **x** o seria.

O operador ***** tem maior precedência que as operações aritméticas, assim a expressão abaixo pega o conteúdo do endereço que **px** aponta e soma **1**.

```
y = *px+1; // y é uma variável do tipo de x
```

No próximo caso somente o ponteiro será incrementado e o conteúdo da próxima posição da memória será atribuído a **y**.

```
y = *(px+1); // y é um ponteiro do tipo de px
```

Os incrementos e decrementos dos endereços podem ser realizados com os operadores ++ e --, que possuem precedência sobre o ***** e operações matemáticas e são avaliados da direita para a esquerda:

```
*px++;    /* sobe uma posição na memória*/
*(px--);  /* mesma coisa de *px-- */
```

No exemplo da Figura 8.2, os parênteses são necessários, pois sem eles **px** seria incrementado em vez do conteúdo que é apontado, porque os operadores ***** e **++** são avaliados da direita para esquerda.

```
(*px)++ /* equivale a x=x+1; ou *px+=1 */
```

```
# include <stdio.h>
main()
{
   int x, *px;
   x = 1;
   px = &x;
   printf("x= %d\n", x);
   printf("px= %u\n", px);
   printf("*px+1= %d\n", *px+1);
   printf("px=%u\n", px);
   printf("*px= %d\n", *px);
   printf("*px+=1= %d\n", *px+=1);
   printf("px= %u\n",px);
   printf("(*px)++=%d\n", (*px)++);
   printf("px= %u\n",px);
   printf("*(px++)= %d\n", *(px++));
   printf("px= %u\n",px);
   printf("*px++-=%d\n", *px++);
   printf("px= %u\n", px);
}
```

Figura 8.2: Operações com Ponteiros

8.4 PONTEIROS PARA PONTEIROS

Um ponteiro para um ponteiro é uma forma de indicação múltipla. Num ponteiro normal, seu valor é o endereço da variável desejada. Quando se trata de ponteiros para ponteiros, o primeiro ponteiro contém o endereço do segundo, que aponta para a variável desejada. A Figura 8.3 apresenta um programa exemplo.

```
float **balanço; // balanço é um ponteiro para um ponteiro float.
```

```
# include <stdio.h>
main( )
{
   int x,*p,**q;
   x=10;
   p=&x;
   q=&p;
   printf("%d",**q);
}
```

Figura 8.3: Ponteiros para ponteiros

8.5 PROBLEMAS COM PONTEIROS

O erro chamado de **ponteiro perdido** é um dos mais difíceis de se encontrar, pois a cada vez que a operação com o ponteiro é utilizada, poderá estar sendo lido ou gravado em posições desconhecidas da memória. Isso pode acarretar sobreposições em áreas de dados ou mesmo área do programa na memória.

```
int *p;
x = 10;
*p = x;
```

Neste trecho de programa, o ponteiro **p** não foi inicializado, não recebeu nenhum endereço, portanto o valor **10** foi colocado em uma posição aleatória e desconhecida de memória. A conseqüência desta atribuição é imprevisível.

9

PONTEIROS E MATRIZES

Em C existe um grande relacionamento entre ponteiros e matrizes, sendo que eles podem ser tratados da mesma maneira. As versões com ponteiros geralmente são mais rápidas.

Quando uma matriz é declarada da seguinte forma:

```
tipo_da_variável nome_da_variável [tam1][tam2] ...  [tamN];
```

o compilador C calcula o tamanho, em bytes, necessário para armazenar esta matriz. Este tamanho é:

```
tam1 x tam2 x tam3 x ...  x tamN x tamanho_do_tipo
```

O compilador então aloca este número de bytes em um espaço livre de memória. O nome da variável que você declarou é na verdade um ponteiro para o tipo da variável da matriz. Este conceito é fundamental. Eis porque: tendo alocado na memória o espaço para a matriz, ele toma o nome da variável (que é um ponteiro) e aponta para o primeiro elemento da matriz.

Mas aí surge a pergunta: então como é que podemos usar a seguinte notação?

```
nome_da_variável[índice]
```

Isto pode ser facilmente explicado desde que você entenda que a notação acima é absolutamente equivalente a se fazer:

```
*(nome_da_variável+índice)
```

É preciso lembrar que a memória de um computador não tem arranjos multidimensionais, isto é, ela pode ser descrita como uma extensa lista de posições de memória enfileiradas, cada uma com um endereço específico. Portanto, quando for declarada uma matriz de dimensão múltipla, o compilador colocará no programa executável uma rotina para o cálculo da posição de memória correspondente aos elementos da matriz referenciados pelos múltiplos índices.

Em C, a indexação de uma matriz começa com o valor zero. É porque, ao pegarmos o valor do primeiro elemento de uma matriz, queremos, de fato, ***nome_da_matriz** e então devemos ter um índice igual a zero, resultando **(*nome_da_matriz+0)**. Então sabemos que:

```
*nome_da_matriz é equivalente a nome_da_matriz[0]
```

9.1 MANIPULANDO MATRIZES ATRAVÉS DE PONTEIROS

Considerando a declaração da matriz **int a[10];** Sendo **pa** um ponteiro para inteiro então:

```
pa = &a[0];
  /*passa o endereço inicial do vetor a para o ponteiro pa */
pa = a;
  /* é a mesma coisa de pa=&a[0];*/
x = *pa;
  /*passa o conteúdo de a[0] para x */
```

Se **pa** aponta para um elemento particular de um vetor **a**, então por definição **pa+1** aponta para o próximo elemento, e em geral **pa-i** aponta para **i** elementos antes de **pa** e **pa+i** para **i** elementos depois.

Se **pa** aponta para **a[0]** então: ***(pa+1)** aponta para **a[1]; pa+i** é o endereço de **a[i]** e ***(pa+i)** é o conteúdo.

É possível fazer cópia de caracteres utilizando matrizes e ponteiros, conforme mostrado no programa da Figura 9.1.

```
# include <stdio.h>
main( )
{
   int i = 0;
   char t[10];
   static char s[ ] = "abobora";
   while (t[i] = s[i]) i++;
   printf("%s\n",t);
}
```

Figura 9.1: Cópia de uma string (versão matriz)

No laço **while**, é feita a cópia de cada elemento de s para cada posição de **t** e é incrementado o índice **i** até que o caractere **'\0'** finalizador de string, ou seja o valor **00** (NULL), seja encontrado em **s**.

O mesmo exemplo pode ser resolvido utilizando-se ponteiros, conforme mostrado na Figura 9.2.

```
# include <stdio.h>
main( )
{
   char *ps,*pt, t[10], s[10];
   strcpy (s,"abobora");
   ps = s;
   pt = &t[0];
   while(*ps) *pt++ = *ps++;
   printf("%s", t );
}
```

Figura 9.2: Cópia de uma string (versão ponteiro)

9.2 STRING E PONTEIROS

Sendo um ponteiro para caractere **char *texto**, podemos atribuir uma constante **string** para **texto**, que não é uma cópia de caracteres, somente ponteiros são envolvidos. Neste caso a **string** é armazenada como parte da função em que aparecem, ou seja, como constante. Um exemplo é visto na Figura 9.3.

```
char *texto = "composto";
  /* funciona como static char texto[ ] = "composto"; */
```

```
#include <stdio.h>
main( )
{
   char *al = "conjunto";
   char re[ ] = "simples";
   puts (al);
   puts (&re[0]);    /* ou puts(re); */
   for( ; al; al++) putchar(*al);
   putchar('\n');
}
```

Figura 9.3: Uso de um ponteiro no lugar de uma string

9.3 MATRIZES DE PONTEIROS

A declaração de matrizes de ponteiros é semelhante a qualquer outro tipo de matrizes:

```
int *x[10];
```

Para atribuir o endereço de uma variável inteira chamada **var** ao terceiro elemento da matriz de ponteiros pode-se fazer um comando de atribuição simples:

```
x[2] = &var;
```

Neste caso, a matriz **x** contém ponteiros para variáveis inteiras, e o endereço da variável **var** foi armazenado na posição 2 da matriz de ponteiros (**x**). Para verificar o conteúdo de **var** pode-se utilizar o ponteiro armazenado em **x**:

```
*x[2];
```

As matrizes de ponteiros são tradicionalmente utilizadas para mensagens de erro, que são constantes :

```
char *erro[ ] = {"arquivo não encontrado\n","erro de leitura\n"};
printf("%s", erro[0]);
printf("%s", erro[1]);
```

Um exemplo é mostrado na Figura 9.4.

```
#include <stdio.h>
main( )
{
   char *erro[2];
   erro[0] = "arquivo nao encontrado\n";
   erro[1] = "erro da leitura\n";
   for( ;*erro[0]; ) printf("%c", *erro[0]++);
}
```

Figura 9.4: Utilização de matrizes de ponteiros para mensagens de erro

10

FUNÇÕES

Uma função é uma unidade autônoma de código do programa e é projetada para cumprir uma tarefa particular. Geralmente os programas em C consistem em várias pequenas funções. A declaração do **tipo** da função é obrigatória no GCC. Os parâmetros de recepção de valores devem ser separados por vírgulas.

Sintaxe: `tipo nome(parâmetros){ comandos}`

Uma função pode retornar apenas um valor para o programa que a chamou, e este valor deve ser do **tipo** especificado no cabeçalho da função, é o **tipo da função**. O **nome** da função deverá sempre iniciar com uma letra, e normalmente é escrito em minúsculas. Às vezes, os programadores iniciantes usam nomes com letras maiúsculas para não correrem o risco de usar nome igual ao de alguma função já definida em alguma biblioteca existente no compilador que está sendo usado. Por exemplo, pode-se definir uma função com o nome **Printf**, escrita com P maiúsculo, para diferenciar da função **printf** da biblioteca **stdio.h**. Os **parâmetros** são valores de variáveis que serão passados pelo programa para a função. Além da variável de retorno nenhum outro valor pode ser retornado pela função para o programa que a chamou.

10.1 FUNÇÃO SEM RETORNO

Quando uma função não retorna um valor para a função que a chamou ela é declarada como **void**. No exemplo da Figura 10.1 a função **inverso** escreve a **string vet** em ordem reversa (do último para o primeiro caracter).

Quando o código da função é escrito após a linha de comando que a chamará pela primeira vez, deve-se escrever uma linha de anúncio da existência desta função, antes do início da função **main**. Este anúncio de função é chamado de **protótipo**. Na verdade o **protótipo** pode ser colocado em outro lugar, desde que seja antes da primeira chamada à função.

A Figura 10.2 apresenta um outro exemplo de função sem retorno.

```
#include <stdio.h>
void inverso(char *s );  /* protótipo de função */
main( )
{
   char *vet = "abcde";
   inverso(vet);
}

void inverso(char *s)   /* o parâmetro passado é um ponteiro */
{
   int t = 0;
   for( ; *s ; s++) t++;
       /* t conta quantos caracteres tem o vetor s */
   s--;
   for( ; t-- ; ) printf("%c",*s--);
   putchar('\n');
}
```

Figura 10.1: Inversão de uma string usando uma função que recebe um ponteiro

10.2 FUNÇÃO COM RETORNO

Quando o programador quiser que a função envie um valor para o programa que a chamou, ele deverá declarar um **tipo** diferente de **void** para a função. A Figura 10.3 apresenta um exemplo de programa que utiliza uma função que recebe dois valores inteiros (base, expoente) e retorna um inteiro (i) que é igual à base elevada ao expoente.

Observe que no **loop for** da função **elevado** não foi feita a inicialização da variável de controle, mas esta variável é o **expoente** que já foi passado como **parâmetro**. O **loop** vai parar de ser executado quando a variável **expoente** atingir o valor **zero**, que funciona como o valor lógico **falso**.

A função **elevado** da Figura 10.3 modifica o valor da variável **expoente** que lhe foi passada pela função **main**, mas esta modificação não afeta a variável **e** da função **main**, pois **expoente** é apenas uma cópia de **e**, e as variáveis de uma função só existem dentro desta, não sendo reconhecidas por outra função.

10.3 PARÂMETROS FORMAIS

Quando uma função utiliza argumentos, então ela deve declarar as variáveis que aceitaram os valores dos argumentos, sendo essas variáveis os parâmetros formais. Veja um

```
#include<stdio.h>
void code (char *s);
main( )
{
   char letras[26];
   printf("\n\n Insira uma frase a ser codificada ");
   printf("(máximo de 25 caracteres). \n");
   scanf("%s", letras);
   code( letras );
}
void code( char *s ) /* Codifica as letras */
{
   char ch;
   do
   {
      ch = *s++;
      printf("%c", ch+1);  /* desloca o alfabeto uma posição */
   } while(ch);
}
```

Figura 10.2: Um codificador simples

exemplo na Figura 10.4, onde são declarados os parâmetros formais **string**, um ponteiro que contém o endereço do início de uma string, e **caractere**, uma variável do tipo char.

A função **pertence** neste exemplo vai procurar um certo **caractere** dentro de uma **string**. Se for encontrado, retorna o valor 1, senão, retorna 0.

10.3.1 Chamada por Valor

O valor de um argumento é copiado para o parâmetro formal da função, portanto as alterações no processamento não alteram as variáveis. No exemplo da Figura 10.5, a variável **x** recebe uma cópia da variável **t**, e a modificação de seu valor dentro da função **Sqr** não altera o valor da variável **t** na função **main**.

10.3.2 Chamada por Referência

Quando se deseja que uma função modifique valores das variáveis do programa que a chamou, deve-se passar para a função não a variável, mas o seu endereço na memória do computador. Isto é feito utilizando-se ponteiros. No exemplo da Figura 10.6 a função

```
#include <stdio.h>
int elevado(int base, int expoente );
main( )
{
   int b,e;
   printf("Digite a base e expoente x, y : ");
   scanf("%d,%d", &b, &e);
   printf("valor = %d\n", elevado(b,e));
}
int elevado(int base, int expoente)
   // são passados dois inteiros como parâmetros
{
   int i;
   if ((expoente<0) || (!base)) return 0;
   i =1;
   for( ; expoente; expoente--) i = base*i;
   return i;   // retorna o inteiro i para a função main
}
```

Figura 10.3: Função com retorno de inteiro

```
int pertence(char *string, char caracter)
{
   while (*string)  if (*string == caracter) return 1;
                    else string++;
   return 0;
}
```

Figura 10.4: Parâmetros formais de uma função

troca troca os valores contidos nas variáveis **x** e **y** de **main**, utilizando os ponteiros **a** e **b** que recebem os endereços de **x** e **y** respectivamente.

10.4 CLASSE DE VARIÁVEIS

Uma função pode chamar outras funções, mas o código que compreende o corpo de uma função (bloco entre { }) está escondido do resto do programa, ele não pode afetar nem ser afetado por outras partes do programa, a não ser que o código use variáveis globais. Existem três **classes** básicas de variáveis: **locais**, **estáticas** e **globais**.

```
#include <stdio.h>
int Sqr( int x );
main( )
{
   int t =10;
   printf("%d eh o quadrado de %d", Sqr(t),t);
}
int Sqr(int x)
{
   x = x*x;  /* Modificação no valor de x não altera t */
   return(x);
}
```

Figura 10.5: Passagem de valor de uma variável para uma função

```
#include <stdio.h>
void troca (int *a, int *b );
main( )
{
   int x=10, y=20;
   troca(&x,&y);
   printf("x=%d y=%d\n", x, y);
}
void troca (int *a, int *b)
   // *a e *b são ponteiros que recebem os endereços de x e y
{
   int temp;
   temp = *a;
   *a = *b;
   *b = temp;
}
```

Figura 10.6: Função que utiliza passagem por referência

10.4.1 Variáveis locais

As variáveis que são declaradas **dentro** de uma função são chamadas de **locais**. Na realidade todas as variáveis declaradas dentro de um bloco { } podem ser referenciadas apenas dentro deste bloco. Elas existem apenas durante a execução do bloco de código no qual estão declaradas. O armazenamento de variáveis locais por default é feito na pilha,

assim sendo uma região dinâmica. A Figura 10.7 apresenta um exemplo de função que utiliza variáveis locais.

```
#include <stdio.h>
void linha (int x);
main( )
{
   int tamanho;
   printf ("Digite o tamanho: ");
   scanf ("%d", &tamanho);
   linha (tamanho);
}
void linha(int x)
{
   int i;
   for( i = 0; i <= x; i++) putchar(95);
   /* A variável i na função linha
      não é reconhecida pela função main.*/
}
```

Figura 10.7: Variáveis locais em uma função

10.4.2 Variáveis Globais

São conhecidas por todo programa e podem ser usadas em qualquer parte do código. Permanecem com seu valor durante toda execução do programa. Deve ser declarada **fora** de qualquer função e até mesmo antes da declaração da função **main**. Fica numa região fixa da memória própria para esse fim. Um exemplo é visto na Figura 10.8.

Sendo a variável **cont** declarada **fora** de qualquer função do programa, ela será uma variável **global**, sendo, portanto reconhecida pelos três blocos de programa (**main**, **func1**, **func2**). A modificação do valor de **cont** em **func2**, resulta na modificação do mesmo **cont** em **func1**. Logo, **func1** irá escrever o valor 109 (e não 100) para **cont**.

10.4.3 Variáveis Estáticas

Funcionam de forma parecida com as variáveis globais, conservando o valor durante a execução de diferentes funções do programa. No entanto só são reconhecidas na função onde estão declaradas. São muito utilizadas para inicializar vetores, conforme pode ser visto no programa da Figura 10.9.

```
# include <stdio.h>
void func1( ), func2( );
int cont;
main( )
{
   cont = 100;
   func1( );
}

void func1( )
{
   int temp;
   temp = cont;
   func2( );
   printf ("temp é = %d", temp);
   printf ("cont é = %d", cont);
}
void func2( )
{
   int cont;
   for(cont =1; cont<10; cont++) printf(" . ");
}
```

Figura 10.8: Variáveis globais em um programa

```
#include <stdio.h>
main( )
{
   int i;
   static int x[10] = {0,1,2,3,4,5,6,7,8,9};
   for(i=0; i<10; i++) printf (" %d \n ", x[i]);
}
```

Figura 10.9: Variáveis locais em uma função

10.5 FUNÇÕES COM MATRIZES

10.5.1 Passando Parâmetros Formais

Para passar uma **matriz** para uma função é necessário passar somente o **endereço** e não uma cópia da matriz. Quando vamos passar uma matriz como argumento de uma função, podemos declarar a função de três maneiras equivalentes. Digamos que temos a seguinte matriz:

```
int matrx [50];
```

e que queiramos passá-la como argumento de uma função **func()**. Podemos declarar **func()** das três maneiras seguintes:

```
void func (int matrx[50]);
void func (int matrx[]);
void func (int *matrx);
```

Veja que, nos três casos, teremos dentro de **func()** um **int*** chamado **matrx**. Note que, no caso de estarmos passando uma matriz para uma função, teremos de passá-la através de um **ponteiro**. Isto faz com que possamos alterar o valor desta matriz dentro da função. A Figura 10.10 apresenta um exemplo de função que trabalha com uma matriz.

```
#include <stdio.h>
void mostra (int num[ ] );
main( )
{
   int t[10], i;
   for ( i = 0; i < 10; i++) t[i] = i;
   mostra(t);
}
void mostra ( int num[ ] )
{
   int i;
   for( i = 0; i < 10; i++ ) printf ("%d", num[i]);
}
```

Figura 10.10: Função que utiliza uma matriz

A função **mostra**, da Figura 10.10, poderia ter sido escrita utilizando-se um **ponteiro** na passagem de parâmetros, conforme se vê na Figura 10.11.

```
void mostra ( int *num )
{
   int i;
   for( i = 0; i < 10; i++) printf ("%d", *(num+i));
}
```

Figura 10.11: Função que opera sobre uma matriz utilizando um ponteiro

A Figura 10.12 apresenta um exemplo de função que recebe dois ponteiros como argumentos e retorna um inteiro. Ela retorna o índice de início de uma **substring** dentro de uma **string** ou **-1**, se nenhuma correspondência for encontrada. Verifique que existem duas declarações **return**, o que é perfeitamente possível em C, visto que a função terminará quando passar pela primeira linha contendo **return**. Sem o uso da segunda declaração **return**, uma variável temporária e código extra precisariam ser usados.

```
encontra_substr(char *sub, char *str)
{
   register int t;
   char *p, *p2;

   for (t = 0; str[t] ; t++)
   {
      p = &str[t];   /* pega o ponto de início de str */
      p2 = sub;
      while(*p2 && (*p2==*p))
         /* enquanto nao for final de sub e for */
      {
         p++;  /* verificado igualdade entre os caracteres */
         p2++; /* de sub e str, avança. */
      }
      if (!*p2) return t;
         /* se está no final de sub, foi encontrada a substring */
   }
   return -1;
}
```

Figura 10.12: Função que procura uma substring dentro de uma string

10.5.2 Alterando os Valores da Matriz

Existem várias formas de se alterar os valores de uma matriz utilizando-se uma função. A Figura 10.13 mostra um exemplo de como fazê-lo utilizando um ponteiro que aponta para o primeiro elemento da matriz **s**.

```
#include <stdio.h>
void maiusc (char *string );
main( )
{
   char s[80];
   gets(s);
   maiusc(s);
}
void maiusc (char *string)
{
   register int t;
   for( t = 0; string[t]; t++)
   {
      string[t] = toupper ( string[t] );
      printf ("%c", string[t] );
   }
}
```

Figura 10.13: Alterando valores de uma matriz

11

ARGUMENTOS DA LINHA DE COMANDO

No ambiente C existe uma maneira de passar argumentos através da linha de comandos para um programa quando ele inicia. O primeiro argumento (**argc**) é a quantidade de argumentos que foram passados quando o programa foi chamado; o segundo argumento (**argv**) é um **ponteiro** de vetores de **caracteres** que contém os **argumentos**, um para cada **string**. Por convenção **argv[0]** é o nome do programa que foi chamado, portanto **argc** é pelo menos igual a **1**. Cada argumento da linha de comando deve ser separado por um espaço ou tabulação. A Figura 11.1 apresenta um programa simples que utiliza argumentos na linha de comando.

```
#include <stdio.h>
int main( int argc, char *argv[ ] )
{
   if (argc != 2)
   {
      printf("falta digitar o nome\n");
      exit(0);
   }
   printf("alo %s", argv[1]);
}
```

Figura 11.1: Programa que recebe uma string na linha de comando

Este programa, depois de compilado, deverá ser chamado na linha de comando como **NomePrograma NomeUsuário**. Então ele escreverá na tela do computador a mensagem: **“alo NomeUsuário”**.

O programa da Figura 11.2 deve ser chamado na linha de comando como **NomePrograma 3 display**, para que seja apresentada na tela a contagem 3, 2, 1, e seja emitido um sinal sonoro (o correspondente ASCII do número 7). Se não for digitado o número 3 e a palavra display, a contagem não será apresentada.

```
#include <stdio.h>
int main( int argc, char *argv[ ] )
{
   int disp, cont;
   if (argc<2)
   {
      printf("falta digitar o valor para contagem\n");
      exit(0);
   }
   if (argc==3 && !strcmp(argv[2],"display")) disp = 1;
   else disp = 0;
   for(cont = atoi( argv[1] ); cont; --cont)
   if(disp) printf("%d", cont);
   printf("%c", 7);
}
```

Figura 11.2: Programa que recebe argumentos na linha de comando

12

ESTRUTURAS, UNIÕES E ENUMERAÇÕES

12.1 ESTRUTURAS

Ao manusearmos dados muitas vezes deparamos com informações que não são fáceis de armazenar em variáveis escalares como são os tipos inteiros e pontos flutuantes, mas na verdade são conjuntos de coisas. Este tipo de dados são compostos com vários dos tipos básicos do C. As **estruturas** permitem uma organização dos dados dividida em **campos** e **registros**. Para se acessar cada elemento de uma estrutura, escreve-se o nome da variável que tem aquele tipo de estrutura, coloca-se um ponto e o nome do elemento que se deseja, como no exemplo da Figura 12.1, onde se referencia cada elemento da variável que tem a estrutura do tipo **lapis**.

Como ocorre com as variáveis, as estruturas também podem ser referenciadas por ponteiros. Assim, definindo-se por exemplo o **ponteiro *p** para a **estrutura** acima (**lapis**), pode-se usar a sintaxe **(*p).dureza**. Porém, para referenciar o **ponteiro** há ainda outra sintaxe, através do operador **->** , como por exemplo, **p -> dureza**.

12.2 UNIÕES

Uma declaração **union** determina uma única localização de memória onde podem estar armazenadas várias variáveis diferentes. A declaração de uma união é semelhante à declaração de uma estrutura:

```
union nome_da_union
{
   tipo_1 nome_1;
   tipo_2 nome_2;
    ...
   tipo_n nome_n;
} variáveis_union;
```

```
# include <stdio.h>
struct lapis { int dureza; char fabricante; int numero; };
main( )
{
   int i;
   struct lapis p[3], C;

   C.dureza = 1;
   C.fabricante = 'H';
   C.numero = 19;

   p[0].dureza = 2;
   p[0].fabricante = 'F';
   p[0].numero = 482;
   p[1].dureza = 0;
   p[1].fabricante = 'G';
   p[1].numero = 33;
   p[2].dureza = 3;
   p[2].fabricante = 'E';
   p[2].numero = 107;

   printf("Dureza Fabricante Numero \n\n");
   printf("Lapis de Cor \n");
   printf("%d \t%c \t%d \n", C.dureza, C.fabricante, C.numero);
   printf("Lapis de Escrever \n");
   for( i = 0; i<3; i++)
   {
      printf("%d \t%c", p[i].dureza, p[i].fabricante);
      printf("\t%d \n", p[i].numero);
   }
}
```

Figura 12.1: Exemplo simples de utilização de estrutura de dados

As **variáveis_union** não precisam ser declaradas imediatamente após a declaração da **union**, elas podem ser declaradas posteriormente com se declaram as variáveis inteiras e de ponto flutuante.

Como exemplo, vamos considerar a união da Figura 12.2.

```
union angulo
{
   float graus;
   float radianos;
};
```

Figura 12.2: União de variáveis de mesmo tipo

Nela, temos duas variáveis (**graus** e **radianos**) que, apesar de terem nomes diferentes, ocupam o mesmo local da memória. Isto quer dizer que só gastamos o espaço equivalente a um único **float**. Uniões podem ser feitas também com variáveis de diferentes tipos. Neste caso, a memória alocada corresponde ao tamanho da maior variável no **union**. Veja o exemplo da Figura 12.3.

```
union numero
{
   char Ch;
   int I;
   float F;
};
```

Figura 12.3: União de variáveis de tipos diferentes

Após a declaração da **union**, podemos declarar as variáveis deste tipo dentro do corpo do programa. Numa declaração como esta, em que variáveis de tipos diferentes ocupam o mesmo espaço na memória, deve-se ter muito cuidado na sua utilização. O exemplo da Figura 12.4 mostra como utilizar a **union numero** em um programa.

No programa da Figura 12.4 deve-se tomar cuidado para, quando se escrever um valor **inteiro** na variável **N**, não tentar acessá-lo como **ponto flutuante**. Deve-se ter cuidado para não se misturar os tipos de dados escritos e acessados na variável do tipo **union**.

12.3 ENUMERAÇÕES

Uma **enumeração** é uma extensão da linguagem C acrescentada pelo padrão ANSI. Uma **enumeração** é um conjunto de constantes inteiras que especifica todos os valores legais que uma variável desse tipo pode ter. Enumerações são comuns na vida cotidiana. Por exemplo, uma enumeração dos dias da semana é

```
Dom, Seg, Ter, Qua, Qui, Sex, Sab.
```

```
#include <stdio.h>
union numero
{
   char Ch;
   int I;
   float F;
};

main (void)
{
   union numero N;   // declara a variável N que é uma union numero
   N.I=123;
   printf ("%d",N.I);
   N.F=123;
   printf ("%f",N.F);
   return 0;
}
```

Figura 12.4: Programa que utiliza a union número

Enumerações são definidas de forma semelhante a estruturas; a palavra-chave **enum** assinala o início de um tipo de enumeração. A forma geral para enumeração é

```
enum nome_da_enumeração {lista_de_valores} lista_de_variáveis;
```

A lista de variáveis é opcional na declaração da enumeração.

Vamos considerar o seguinte exemplo:

```
enum dias_da_semana { Dom, Seg, Ter, Qua, Qui, Sex, Sab };
```

O programador diz ao compilador que qualquer variável do tipo **dias_da_semana** só pode ter os valores enumerados. Isto quer dizer que poderíamos fazer o programa da Figura 12.5, onde duas variáveis **d1** e **d2** serão do tipo **enumeração dias_da_semana**.

Você deve estar se perguntando como é que a enumeração funciona. Simples. O compilador pega a lista que você fez de valores e associa, a cada um, um número inteiro. Então, ao primeiro da lista, é associado o número zero, ao segundo o número 1 e assim por diante. As variáveis declaradas são então variáveis **int**.

```
#include <stdio.h>
enum dias_da_semana { Dom, Seg, Ter, Qua, Qui, Sex, Sab };
int main ( )
{
   enum dias_da_semana d1,d2;
   d1 = Seg;
   d2 = Sex;
   if (d1 == d2)
      // são comparados os inteiros correspondentes a Seg e Sex
   {
      printf ("O dia eh o mesmo.");
   }
   else
   {
      printf ("São dias diferentes.");
   }
   return 0;
}
```

Figura 12.5: Exemplo de utilização de enumerações

13

NOÇÕES DE MANIPULAÇÃO DE ARQUIVOS

Para tratar de arquivos a linguagem C fornece um nível de abstração entre o programador e o dispositivo que estiver sendo usado. Esta abstração é chamada **fila de bytes** e o dispositivo normalmente é o **arquivo**. Existe um sistema bufferizado de acesso ao arquivo, onde um **ponteiro de arquivo** define vários aspectos do arquivo, como **nome**, **status** e **posição corrente**, além de ter a **fila** associada a ele. A Figura 13.1 mostra um programa exemplo que escreve uma seqüência de números em um arquivo (teste.dat) e depois lê os valores contidos no arquivo.

13.1 ABRINDO E FECHANDO UM ARQUIVO

No sistema de entrada e saída ANSI é definido o tipo "ponteiro de arquivo". Este não é um tipo propriamente dito, mas uma definição usando o comando **typedef**. Esta definição está no arquivo cabeçalho **stdio.h** ou **stdlib.h** dependendo do seu compilador. Podemos declarar um ponteiro de arquivo da seguinte maneira:

```
FILE *p;
```

p será então um ponteiro para um arquivo. É usando este tipo de ponteiro que vamos poder manipular arquivos no C.

13.1.1 A função fopen

Esta é a função de abertura de arquivos. Seu protótipo é:

```
FILE *fopen (char *nome_do_arquivo, char *modo);
```

O **nome_do_arquivo** determina qual arquivo deverá ser aberto. Este nome deve ser válido no sistema operacional que estiver sendo utilizado. O **modo** de abertura diz à função **fopen()** que tipo de uso você vai fazer do arquivo. A Tabela 13.1 mostra os valores de modo válidos.

Poderíamos então, para abrir um arquivo binário, escrever:

```
#include <stdio.h>
main ( )
{
   FILE *fp;
   char ch;
   int nu, *pn;
   pn = &nu;
   fp = fopen("teste.dat", "w");
   printf("Entre com os numeros para gravar e 0 para sair: ");
   scanf("%d", &nu);
   while(nu)
   {
      fprintf (fp,"%d ", nu);
      scanf("%d", &nu);
   }
   fclose(fp);
   fp = fopen("teste.dat", "r");
   while(!feof(fp))
   {
      fscanf( fp,"%d ", &nu);
      printf( "%d", nu);
   }
}
```

Figura 13.1: Escrevendo e lendo em arquivo

```
FILE *fp;
fp = fopen ("exemplo.bin","wb");
if (!fp)
   printf ("Erro na abertura do arquivo.");
```

A condição **!fp** testa se o arquivo foi aberto com sucesso porque no caso de um erro a função **fopen()** retorna um ponteiro nulo (**NULL**).

13.1.2 A função exit

Aqui abrimos um parêntesis para explicar a função **exit()** cujo protótipo é:

```
void exit (int codigo_de_retorno);
```

Esta função aborta a execução do programa. Pode ser chamada de qualquer ponto no programa e faz com que o programa termine e retorne, para o sistema operacional, o

Tabela 13.1: Modos válidos de abertura de arquivos

Modo	Significado
`"r"`	Abre um arquivo para leitura
`"w"`	Cria um arquivo para escrita
`"a"`	Acrescenta dados no fim do arquivo (“append”)
`"rb"`	Abre um arquivo binário para leitura
`"wb"`	Cria um arquivo binário para escrita
`"ab"`	Acrescenta dados binários no fim do arquivo
`"r+"`	Abre um arquivo para leitura e escrita
`"w+"`	Cria um arquivo para leitura e escrita
`"a+"`	Acrescenta dados ou cria uma arquivo para leitura e escrita
`"r+b"`	Abre um arquivo binário para leitura e escrita
`"w+b"`	Cria um arquivo binário para leitura e escrita
`"a+b"`	Acrescenta dados ou cria uma arquivo binário para leitura e escrita
`"rt"`	Abre um arquivo texto para leitura
`"wt"`	Cria um arquivo texto para escrita
`"at"`	Acrescenta dados no fim do arquivo texto
`"r+t"`	Abre um arquivo texto para leitura e escrita
`"w+t"`	Cria um arquivo texto para leitura e escrita
`"a+t"`	Acrescenta dados ou cria uma arquivo texto para leitura e escrita

código_de_retorno. A convenção mais usada é que um programa retorne zero no caso de um término normal e retorne um número não nulo no caso de ter ocorrido um problema. A função **exit()** se torna importante em casos como alocação dinâmica e abertura de arquivos pois pode ser essencial que uma determinada memória seja alocada ou que um arquivo seja aberto. Poderíamos reescrever o exemplo da seção anterior usando agora o **exit()** para garantir que o programa não deixará de abrir o arquivo, como é mostrado na Figura 13.2.

13.1.3 A função fclose

Quando abrimos um arquivo devemos fechá-lo. Para tanto devemos usar a função **fclose()**:

```
int fclose (FILE *fp);
```

É importante que se perceba que se deve tomar o maior cuidado para não se “perder” o ponteiro do arquivo. “Perder” neste caso seria se atribuir um valor de um outro ponteiro

```
#include <stdio.h>
main (void)
{
   FILE *fp;

   /* Comandos... */

   fp = fopen ("exemplo.bin", "wb");
   if (!fp)
   {
      printf ("Erro na abertura do arquivo. Fim de programa.");
      exit (1);
   }

   /* Comandos ... */

   return 0;
}
```

Figura 13.2: Uso da função exit na abertura de arquivos

qualquer ao ponteiro de arquivo (perdendo assim o valor original). É utilizando este ponteiro que vamos poder trabalhar com o arquivo. Se perdermos o ponteiro de um determinado arquivo não poderemos nem fechá-lo. O ponteiro **fp** passado à função **fclose()** determina o arquivo a ser fechado. A função retorna zero no caso de sucesso.

13.2 LENDO E ESCREVENDO CARACTERES EM ARQUIVOS

Além das funções **fprintf** e **fscanf** vista no exemplo inicial, outras funções existem no padrão ANSI para se escrever e se ler em arquivos. Aqui serão citadas brevemente as funções **putc**, **getc**, e a função **feof** que verifica o se o arquivo chegou ao final.

13.2.1 putc

Toda vez que estamos trabalhando com arquivos, há uma espécie de posição atual no arquivo. Esta posição, gerenciada pelo compilador, é a posição de onde será lido ou escrito o próximo caracter. Normalmente, num acesso seqüencial a um arquivo, não temos que mexer nesta posição pois quando lemos um caractere a posição no arquivo é automatica-

mente atualizada. Num acesso randômico teremos que mexer nesta posição (ver **fseek()**). Protótipo:

```
int putc (int ch,FILE *fp);
```

Escreve um caracter no arquivo.

13.2.2 getc

Retorna um caracter lido do arquivo. Protótipo:

```
int getc (FILE *fp);
```

13.2.3 feof

EOF (“*End of file*”) indica o fim de um arquivo. Às vezes, é necessário verificar se um arquivo chegou ao fim. Para isto podemos usar a função **feof()**. Ela retorna não-zero se o arquivo chegou ao **EOF**, caso contrário retorna zero. Seu protótipo é:

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

14

NOÇÕES DE ALOCAÇÃO DINÂMICA

Há duas maneiras de armazenar variáveis na memória do computador. Primeiro por variáveis **globais** e **static** locais, segundo através de **alocação dinâmica**, quando o C armazena a informação em uma área de memória livre, de acordo com a necessidade. No caso do C standard, a alocação dinâmica fica disponível com a inclusão de **stdio.h**. A alocação dinâmica de memória é muito útil quando se tem que trabalhar com várias matrizes de grandes dimensões. Para cada matriz a ser utilizada pelo programa destina-se uma área de memória suficiente, e após o seu uso, esta área é liberada para que possa ser utilizada por outras matrizes. As Figuras 14.1 e 14.2 mostram exemplos de alocação dinâmica de memória.

```
#include <stdio.h>
main( )
{
   int *p, t;
   p=(int *) malloc(40*sizeof(int));
   if (!p) printf("memoria insuficiente\n");
   else
   {
      for(t=0; t<40; ++t) *(p+t) = t;
      for(t=0; t<40; ++t) printf("%d ", *(p+t));
      free(p);
   }
}
```

Figura 14.1: Primeiro exemplo de alocação dinâmica de memória

```
#include <stdio.h>
main( )
{
   int i,quant;
   float max,min,*p;
   printf ("quantidade de numeros:\n");
   scanf("%d", &quant);
   if ( !( p = (float*) malloc((quant+1)*sizeof(float))) )
   {
      printf ("sem memoria\n");
      exit(1);
   }
   printf ("digite %d numeros:\n", quant);
   for ( i = 1; i <= quant; i++) scanf ("%f", (p+i));
   max = *(p+1);
   for ( i = 2; i <= quant; i++)
   {
      if (*(p+i) >= max) max = *(p+i);
   }
   printf("O maior e :%f \n", max);
   free(p);
}
```

Figura 14.2: Segundo exemplo de alocação dinâmica de memória

15

EXERCÍCIOS

15.1 CAPÍTULO 1

1. Procurar em livros e/ou guias de referência da Linguagem C quais são as funções das seguintes bibliotecas: string.h e math.h. Anotar o nome de todas as funções com seus respectivos tipos de retorno e seus argumentos.

2. Dizer o que fazem as seguintes funções da biblioteca stdio.h: getchar, putc, puts.

15.2 CAPÍTULO 2

1. Editar o programa da Figura 15.1 e fazer sua compilação. Executar o programa e dizer o que ele faz.

```
/* programa do Exercicio_1  capitulo_2 */
# include <stdio.h>
int main ( )
{
   int   x, y;
   float    X, Y;
   printf (" \n\t Envie dois numeros inteiros \n\t ");
   scanf (" %d %d ", &x, &y );
   X = x;
   Y = y;
   printf ( "\n\t Divisão X/Y \n");
   printf ( "\n\t %f \n\t %d \n\t %d \n", X/Y, x/y, x%y);
}
```

Figura 15.1: Capítulo 2: Programa para o exercício 1

Para escrever os dois números pedidos pelo programa, pode-se digitar o primeiro, um espaço, digitar o segundo, e pressionar <ENTER>. Pode-se também digitar <ENTER> após cada número.

15.3 CAPÍTULO 3

1. Editar, compilar e executar o programa da Figura 15.2.

```
/* programa do Exercicio_1  capitulo_3 */
# include <stdio.h>
int main ( )
{
   int Dias;
   float Anos;
   printf ("\n\t Entre com o número de dias: ");
   scanf ("%d", &Dias);
   Anos = Dias/365.25;
   printf ("\n\n\t %d dias equivalem a %f anos.\n",Dias,Anos);
}
```

Figura 15.2: Capítulo 3: programa para o exercício 1

Dizer o que o programa faz. Explicar (comentar) o que faz cada linha. Explique porque a variável **Dias** é inteira e **Anos** é float.

2. Modifique o programa anterior para que sejam requisitados o dia, o mês e o ano atuais, o dia, o mês e o ano de nascimento do usuário, e então seja calculado e anunciado o número de dias vivido pelo usuário.

15.4 CAPÍTULO 4

1. Editar, compilar e executar o programa da Figura 15.2.

2. Modifique o programa para que todos os números sejam impressos com quatro algarismos antes do ponto decimal e três algarismos após.

15.5 CAPÍTULO 5

1. Dizer o que faz o programa da Figura 15.4.

```c
/* programa do Exercicio_1  capitulo_3 */
# include <stdio.h>
int main ( )
{
   char Ch;
   float x, y;
   printf ("\n Envie dois números : ");
   scanf ("%f %f ", &x, &y );
   printf ( "\n O produto de %f e %f eh %f ", x, y, x*y );
   Ch = getchar( );
   printf ( "\n O quociente de %f e %f eh %f ", x, y, x/y );
   printf ( "\n A tecla pressionada foi %c \n\n", Ch );
   printf ( "Fim de Programa\n" );
}
```

Figura 15.3: Capítulo 4: Programa para o exercício 1

2. Reescrever o programa do item anterior utilizando o comando switch. Editar, compilar e executar o programa.

3. Fazer um programa em C que identifique triângulos, conforme o algoritmo da Figura 15.5

4. Modificar o programa de cálculo de PI (Figura 5.8), do Capítulo 5, de forma que ele calcule a série até que encontre um termo cujo valor absoluto seja menor que 0,00001. Para isto, deverá ser usado um Loop while ou do-while.

5. O seno de um ângulo qualquer (dado em radianos) pode ser calculado pela série abaixo.

   $senA = A - \frac{A^3}{6} + \frac{A^5}{120} - \frac{A^7}{5040} + \cdots$

   Os números que aparecem no denominador são os fatoriais dos expoentes de cada termo.

   Fazer um programa em linguagem C que calcule o seno de um ângulo dado utilizando os N primeiros termos da série. O ângulo A(radianos) e o valor de N deverão ser requisitados ao usuário do programa.

6. Reescrever o programa anterior para que o seno do ângulo seja calculado até que o módulo do último termo da série seja menor que 0,00001.

```
#include<stdio.h>
main( )
{
   int opcao;
   int valor;
   printf("Converter:\n");
   printf("        1: decimal para hexadecimal\n");
   printf("        2: hexadecimal para decimal\n");
   printf("        3: decimal para octal\n");
   printf("        4: octal para decimal\n");
   printf("informe a sua opção:");
   scanf("%d", &opcao);
   if(opcao==1)  {
      printf("informe um valor em decimal:");
      scanf("%d", &valor);
      printf("%d em hexadecimal é : %x", valor, valor);
   }
   if(opcao==2) {
      printf("informe um valor em hexadecimal:");
      scanf("%x", &valor);
      printf("%x em decimal é: %d", valor, valor);
   }
   if(opcao==3){
      printf("informe um valor em decimal:");
      scanf("%d", &valor);
      printf("%d em octal é: %o", valor, valor);
   }
   if(opcao==4){
      printf("informe um valor em octal:");
      scanf("%o", &valor);
      printf("%o em decimal é: %d", valor, valor);
   }
}
```

Figura 15.4: Capítulo 5: Programa para o exercício 1

```
/* Identificação de triângulos */

Início
   Ler   A, B, C   (reais, lados do triângulos)
   Se   A < (B+C) e B < (A+C) e C < (A+B)
      /* verifica se é triângulo */
   Então
      Se    A = B e  B = C
      Então Imprima (Triângulo Equilátero)
      Senão
         Se    A = B ou A = C ou B = C
            Então Imprima (Triângulo Isóceles)
            Senão Imprima (Triângulo Escaleno)
         FimSe
      FimSe
   FimSe
Fim
```

Figura 15.5: Capítulo 5: Algoritmo para o exercício 3

15.6 CAPÍTULO 6

1. Fazer um programa em Linguagem C que leia uma matriz de dimensões NxM, e calcule a sua transposta. Ao final o programa apresentará a matriz e sua transposta na tela do computador.

2. Fazer um programa em Linguagem C que leia os nomes de 10 alunos de uma turma, e para cada aluno, 4 notas de 0 a 100%. Os nomes dos alunos serão armazenados em um vetor de strings, as notas serão armazenadas em uma matriz bidimensional. Serão calculadas as médias dos alunos, (Nota1+Nota2+Nota3+Nota4)/4, e estas serão armazenadas em um vetor. Ao final, será apresentado um relatório contendo 3 colunas: Nome do aluno, Média, Aprovação. Na coluna aprovação será colocada a letra A (aprovado) para alunos com média igual ou superior a 60, e R (reprovado) para alunos com média inferior a 60.

15.7 CAPÍTULO 7

1. Editar, compilar, executar e dizer o que faz o programa da Figura 15.6.

```
#include <stdio.h>
#include <string.h>
main( )
{
   char st1[11], st2[11], st3[21], ch ;
   int i, j;
   printf ("\n Envie uma string de ate 10 caracteres  ");
   gets(st1);
   printf ("\n Envie outra string de ate 10 caracteres  ");
   gets(st2);
   for ( i = 0; st1[i]; i++) st3[i] = st1[i];
   for ( j = 0; st2[j]; j++) st3[i+j] = st2[j];
   j = j+i;
   puts (st3);
   for ( i = j; i ; i-- )
   {
      ch = st3[i-1];
      putchar (ch);
      putchar ('\n');
   }
   puts ("\n");
}
```

Figura 15.6: Capítulo 7: programa para o exercício 1

Explique porque as strings st1 e st2 devem ter no máximo 10 caracteres se os respectivos vetores foram declarados com 11 posições.

15.8 CAPÍTULOS 8 E 9

1. Fazer um programa em C que leia uma string de até 30 caracteres e armazene-a em uma variável Stfonte. Utilizando ponteiros, copie o conteúdo de Stfonte para uma variável Stdestino, de forma que a string fique escrita de forma inversa. Utilizando ponteiros, apresentar as strings Stfonte e Stdestino em forma de duas colunas.

2. Refazer o programa do cálculo das médias dos alunos, Capítulo 6 (Figura 6.3, utilizando ponteiros para o cálculo das médias e para a apresentação dos resultados.

15.9 CAPÍTULO 10

1. Fazer uma função em C que concatena duas strings. Utilize o nome StrCat para esta função, para não confundir com como strcat() da biblioteca string.h.

2. Editar, compilar e executar o programa da Figura 15.7

```
/* cálculo da área de um círculo */

#include <stdio.h>
#include <math.h>

float area(float raio);     /* protótipo*/

main( )
{
   float r,res;
   printf("Informe o raio: ");
   scanf("%f", &r);
   res=area(r);
   printf("A área é: %f\n", res);
}

float area(float raio)
{
   return 3.1415926 * pow(raio,2);
}
```

Figura 15.7: Capítulo 10: Programa para o exercício 2

3. Modificar o programa anterior para calcular a área e o perímetro do círculo, e calcular o volume e a área de uma esfera de mesmo raio. Deverão ser feitas 4 funções.

4. Modifique o programa anterior para que ele repita os cálculo para diferentes raios enviados pelo usuário. Deverá ser utilizado um loop do-while onde, ao final será perguntado ao usuário se deseja novo cálculo. O loop será repetido até que o usuário responda N (não).

5. Fazer uma função em C que calcule o fatorial de um número passado como parâmetro.

6. Fazer uma função em C que calcule o produto de duas matrizes A e B e retorne o resultado em outra matriz P.

15.10 CAPÍTULO 11

1. Fazer um programa em C que receba em sua linha de comando o nome do usuário e escreva a mensagem: “Bom dia <usuário>”. Onde se lê <usuário> deverá estar escrito o nome completo do usuário. O programa deverá identificar quantos nomes foram escritos.

2. Fazer um programa em C que calcule as raízes de uma equação de segundo grau. Os valores A, B, C da equação serão passados na linha de comando. $Ax^2 + Bx + C = 0$.

15.11 CAPÍTULO 12

1. Fazer um programa em C para ler e armazenar em uma estrutura, os dados de uma pessoa. Os campos da estrutura DADOS serão: nome, idade, telefone, sexo (M/F).

15.12 CAPÍTULO 13

1. Usar a estrutura do exercício 1 do Capítulo 12 para armazenar em um vetor os dados de um conjunto de 20 pessoas, que poderiam ser por exemplo os candidatos a vagas em uma empresa.

15.13 CAPÍTULO 14

1. Modificar o programa do exercício 1 do Capítulo 13 para armazenar os dados em arquivo. O programa deverá ter um menu de opções para trabalhar com os dados (usar o comando switch). No menu será previsto inclusão de dados, exclusão de dados, e consulta. Para cada opção deverá ser feita uma função específica. Na opção consulta, deverá ter um menu de opções: por nome, idade, sexo. Quando for pedida uma consulta por nome, por exemplo, deverão ser apresentados todos os dados das pessoas que possuem aquele nome.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] GALIC-UNICAMP, **Introdução à Linguagem C**, UNICAMP, Campinas, 2001.

[2] CPDEE-UFMG, **Curso de C**, UFMG, Belo Horizonte, 2000.

[3] KERNIGHAN, Brian W. e RITCH, Dennis M., **C: A Linguagem de Programação**, Rio de Janeiro, Campus, 1986.

[4] PLAUGER, P.J. e BRODIE J. **Standart C: guia de referência básica**, São Paulo, Mcgraw-Hill, 1991. 306p.

[5] HANCOCK, Les e KRIEGER, Morris. **Manual de Linguagem C**, Rio de Janeiro, Campus,1985. 182p.

[6] MIZRAHI, Viviane V. **Treinamento em Linguagem C — módulo 1 e 2**, São Paulo, McGraw-Hill, 1990, 241p.

[7] CHILDT, Herbert. **Turbo C: Guia do Usuário**, São Paulo, McGraw-Hill, 1988, 414p.