E BALAGURUSAMY

PROGRAMMING IN ANSI C

Fifth Edition

Copyrighted material

Tata McGraw-Hill

Published by Tata McGraw Hill Education Private Limited,
7 West Patel Nagar, New Delhi 110 008

Programming in ANSI C, 5e

Copyright © 2011, 2007, by Tata McGraw Hill Education Private Limited

First reprint 2010
DQLARRYZRABCC

No part of this publication may be reproduced or distributed in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise or stored in a database or retrieval system without the prior written permission of the publishers. The program listings (if any) may be entered, stored and executed in a computer system, but they may not be reproduced for publication.

This edition can be exported from India only by the publishers,
Tata McGraw Hill Education Private Limited.

ISBN (13 digits): 978-0-07-068182-8
ISBN (10 digits): 0-07-068182-1

Vice President and Managing Director—McGraw-Hill Education: Asia Pacific Region: *Ajay Shukla*

Head—Higher Education Publishing and Marketing: *Vibha Mahajan*
Manager: Sponsoring—SEM & Tech Ed: *Shalini Jha*
Asst Sponsoring Editor: *Surabhi Shukla*
Development Editor: *Surbhi Suman*
Executive—Editorial Services: *Sohini Mukherjee*
Jr Production Manager: *Anjali Razdan*

Dy Marketing Manager—SEM & Tech Ed: *Biju Ganesan*

General Manager—Production: *Rajender P Ghansela*
Asst General Manager—Production: *B L Dogra*

Information contained in this work has been obtained by Tata McGraw-Hill, from sources believed to be reliable. However, neither Tata McGraw-Hill nor its authors guarantee the accuracy or completeness of any information published herein, and neither Tata McGraw-Hill nor its authors shall be responsible for any errors, omissions, or damages arising out of use of this information. This work is published with the understanding that Tata McGraw-Hill and its authors are supplying information but are not attempting to render engineering or other professional services. If such services are required, the assistance of an appropriate professional should be sought.

Typeset at Text-o-Graphics, B1/56 Arawali Apartment, Sector 34, Noida 201301 and
printed at Gopsons, A – 2 & 3, Sector – 64, Noida, U.P. – 201 301

Cover Printer: Gopsons

The McGraw·Hill Companies

Copyrighted material

Contents

Preface to the Fifth Edition *xi*

1 Overview of C **1**

1.1 History of C *1*
1.2 Importance of C *3*
1.3 Sample Program 1: Printing a Message *3*
1.4 Sample Program 2: Adding Two Numbers *6*
1.5 Sample Program 3: Interest Calculation *8*
1.6 Sample Program 4: Use of Subroutines *10*
1.7 Sample Program 5: Use of Math Functions *11*
1.8 Basic Structure of C Programs *12*
1.9 Programming Style *14*
1.10 Executing a 'C' Program *14*
1.11 Unix System *16*
1.12 Ms-Dos System *18*

Review Questions 19
Programming Exercises 20

2 Constants, Variables, and Data Types **23**

2.1 Introduction *23*
2.2 Character Set *23*
2.3 C Tokens *25*
2.4 Keywords and Identifiers *25*
2.5 Constants *26*
2.6 Variables *30*
2.7 Data Types *31*
2.8 Declaration of Variables *34*
2.9 Declaration of Storage Class *37*
2.10 Assigning Values to Variables *38*
2.11 Defining Symbolic Constants *44*
2.12 Declaring a Variable as Constant *45*
2.13 Declaring a Variable as Volatile *45*

2.14 Overflow and Underflow of Data *46*

Review Questions 49
Programming Exercises 51

3 Operators and Expressions **52**

3.1 Introduction *52*
3.2 Arithmetic Operators *52*
3.3 Relational Operators *55*
3.4 Logical Operators *57*
3.5 Assignment Operators *57*
3.6 Increment and Decrement Operators *59*
3.7 Conditional Operator *61*
3.8 Bitwise Operators *61*
3.9 Special Operators *61*
3.10 Arithmetic Expressions *63*
3.11 Evaluation of Expressions *64*
3.12 Precedence of Arithmetic Operators *65*
3.13 Some Computational Problems *67*
3.14 Type Conversions in Expressions *68*
3.15 Operator Precedence and Associativity *72*
3.16 Mathematical Functions *74*

Review Questions 78
Programming Exercises 81

4 Managing Input and Output Operations **84**

4.1 Introduction *84*
4.2 Reading a Character *85*
4.3 Writing a Character *88*
4.4 Formatted Input *89*
4.5 Formatted Output *98*

Review Questions 110
Programming Exercises 112

5 Decision Making and Branching **114**

5.1 Introduction *114*
5.2 Decision Making with IF Statement *114*
5.3 Simple IF Statement *115*
5.4 The IF.....ELSE Statement *119*
5.5 Nesting of IF....ELSE Statements *122*
5.6 The ELSE IF Ladder *126*
5.7 The Switch Statement *129*
5.8 The ? : Operator *133*
5.9 The GOTO Statement *136*

Review Questions 144
Programming Exercises 148

6 Decision Making and Looping 152

6.1 Introduction *152*
6.2 The WHILE Statement *154*
6.3 The DO Statement *157*
6.4 The FOR Statement *159*
6.5 Jumps in LOOPS *166*
6.6 Concise Test Expressions *174*

Review Questions 182
Programming Exercises 186

7 Arrays 190

7.1 Introduction *190*
7.2 One-dimensional Arrays *192*
7.3 Declaration of One-dimensional Arrays *193*
7.4 Initialization of One-dimensional Arrays *195*
7.5 Two-dimensional Arrays *199*
7.6 Initializing Two-dimensional Arrays *204*
7.7 Multi-dimensional Arrays *208*
7.8 Dynamic Arrays *209*
7.9 More about Arrays *209*

Review Questions 223
Programming Exercises 225

8 Character Arrays and Strings 229

8.1 Introduction *229*
8.2 Declaring and Initializing String Variables *230*
8.3 Reading Strings from Terminal *231*
8.4 Writing Strings to Screen *236*
8.5 Arithmetic Operations on Characters *241*
8.6 Putting Strings Together *242*
8.7 Comparison of Two Strings *244*
8.8 String-handling Functions *244*
8.9 Table of Strings *250*
8.10 Other Features of Strings *252*

Review Questions 257
Programming Exercises 259

9 User-defined Functions 262

9.1 Introduction *262*
9.2 Need for User-defined Functions *262*
9.3 A Multi-function Program *263*
9.4 Elements of User-defined Functions *266*
9.5 Definition of Functions *267*
9.6 Return Values and their Types *269*
9.7 Function Calls *270*
9.8 Function Declaration *272*

9.9 Category of Functions 274
9.10 No Arguments and no Return Values 274
9.11 Arguments but no Return Values 277
9.12 Arguments with Return Values 280
9.13 No Arguments but Returns a Value 284
9.14 Functions that Return Multiple Values 285
9.15 Nesting of Functions 286
9.16 Recursion 288
9.17 Passing Arrays to Functions 289
9.18 Passing Strings to Functions 294
9.19 The Scope, Visibility and Lifetime of Variables 295
9.20 Multifile Programs 305

Review Questions 311
Programming Exercises 315

10 Structures and Unions **317**

10.1 Introduction 317
10.2 Defining a Structure 317
10.3 Declaring Structure Variables 319
10.4 Accessing Structure Members 321
10.5 Structure Initialization 322
10.6 Copying and Comparing Structure Variables 324
10.7 Operations on Individual Members 326
10.8 Arrays of Structures 327
10.9 Arrays within Structures 329
10.10 Structures within Structures 331
10.11 Structures and Functions 333
10.12 Unions 335
10.13 Size of Structures 337
10.14 Bit Fields 337

Review Questions 344
Programming Exercises 348

11 Pointers **351**

11.1 Introduction 351
11.2 Understanding Pointers 351
11.3 Accessing the Address of a Variable 354
11.4 Declaring Pointer Variables 355
11.5 Initialization of Pointer Variables 356
11.6 Accessing a Variable through its Pointer 358
11.7 Chain of Pointers 360
11.8 Pointer Expressions 361
11.9 Pointer Increments and Scale Factor 362
11.10 Pointers and Arrays 364
11.11 Pointers and Character Strings 367
11.12 Array of Pointers 369

11.13 Pointers as Function Arguments *370*
11.14 Functions Returning Pointers *373*
11.15 Pointers to Functions *373*
11.16 Pointers and Structures *376*
11.17 Troubles with Pointers *379*

Review Questions 385
Programming Exercises 388

12 File Management in C 389

12.1 Introduction *389*
12.2 Defining and Opening a File *390*
12.3 Closing a File *391*
12.4 Input/Output Operations on Files *392*
12.5 Error Handling During I/O Operations *398*
12.6 Random Access to Files *400*
12.7 Command Line Arguments *405*

Review Questions 408
Programming Exercises 409

13 Dynamic Memory Allocation and Linked Lists 411

13.1 Introduction *411*
13.2 Dynamic Memory Allocation *411*
13.3 Allocating a Block of Memory: MALLOC *413*
13.4 Allocating Multiple Blocks of Memory: CALLOC *415*
13.5 Releasing the Used Space: Free *415*
13.6 Altering the Size of a Block: REALLOC *416*
13.7 Concepts of Linked Lists *417*
13.8 Advantages of Linked Lists *420*
13.9 Types of Linked Lists *421*
13.10 Pointers Revisited *422*
13.11 Creating a Linked List *424*
13.12 Inserting an Item *428*
13.13 Deleting an Item *431*
13.14 Application of Linked Lists *433*

Review Questions 440
Programming Exercises 442

14 The Preprocessor 444

14.1 Introduction *444*
14.2 Macro Substitution *445*
14.3 File Inclusion *449*
14.4 Compiler Control Directives *450*
14.5 ANSI Additions *453*

Review Questions 456
Programming Exercises 457

15 Developing a C Program: Some Guidelines 458

15.1 Introduction *458*
15.2 Program Design *458*
15.3 Program Coding *460*
15.4 Common Programming Errors *462*
15.5 Program Testing and Debugging *469*
15.6 Program Efficiency *471*

Review Questions 472

Appendix I: Bit-level Programming *474*
Appendix II: ASCII Values of Characters *480*
Appendix III: ANSI C Library Functions *482*
Appendix IV: Projects *486*
Appendix V: C99 Features *537*

Bibliography 545
Index 547

Preface to the Fifth Edition

C is a powerful, flexible, portable and elegantly structured programming language. Since C combines the features of high-level language with the elements of the assembler, it is suitable for both systems and applications programming. It is undoubtedly the most widely used general-purpose language today.

Since its standardization in 1989, C has undergone a series of changes and improvements in order to enhance the usefulness of the language. The version that incorporates the new features is now referred to as C99.

The fifth edition comes with a free CD. The CD contains the programs of the book along with two major projects in ready-to-compile and execute format.

Organization of the Book

The book starts with an overview of C, which talks about the history of C, basic structure of C programs and their execution. The second chapter discusses how to declare the constants, variables and data types. The third chapter describes the built-in operators and how to build expressions using them. The fourth chapter details the input and output operations. Decision making and branching is discussed in the fifth chapter, which talks about the if-else, switch and goto statements. Further, decision making and looping is discussed in Chapter six, which covers while, do and for loops. Arrays and ordered arrangement of data elements are important to any programming language and have been covered in chapters seven and eight. Strings are also covered in Chapter eight. Chapters nine and ten are on functions, structures and unions. Pointers, perhaps the most difficult part of C to understand, is covered in Chapter eleven in the most user-friendly manner. Chapters twelve and thirteen are on file management and dynamic memory allocation respectively. Chapter fourteen deals with the preprocessor, and finally Chapter 15 is on developing a C program, which provides an insight on how to proceed with development of a program. The above organization would help the students in understanding C better if followed appropriately.

The content has been revised keeping the updates which have taken placed in the field of C programming and the present day syllabus needs. As always, the concept of learning by example. has been stressed throughout the book. Each major feature of the language is treated in depth followed by a complete program example to illustrate its use. The sample programs are meant to be both simple and educational.

Each chapter includes a section at the beginning to introduce the topic in a proper perspective. It also provides a quick look into the features that are discussed in the chapter. Wherever necessary, pictorial descriptions of concepts are included to improve clarity and to facilitate better understanding. Language

Copyrighted material

tips and other special considerations are highlighted as notes wherever essential. Following are some of the key features of the book.

- Free CD with the book containing—
  - Executable codes to the programs (given inside the book) in chapterwise fashion.
  - Two programming projects: Inventory and Record Entry.
- Codes with comments are provided throughout the book to illustrate how the various features of the language are put together to accomplish specified tasks.
- Supplementary information and notes that complement but stand apart from the general text have been included in boxes.
- Guidelines for developing efficient C programs are given in the last chapter, together with a list of some common mistakes that a less experienced C programmer could make.
- Case studies at the end of the chapters illustrate common ways C features are put together and also show real-life applications.
- The Just Remember section at the end of the chapters lists out helpful hints and possible problem areas.
- Numerous chapter-end questions and exercises provide ample opportunities to the readers to review the concepts learned and to practice their applications.
- Programming projects discussed in the appendix give insight on how to integrate the various features of C when handling large programs.

Supplementory Material

The book is also accompanied with a website (*http://www.mhhe.com/balagurusamy/ansic5e*) which includes the following:

For the Instructor

- Chapterwise PowerPoint Slides

For the Student

- Case Studies (chapterwise)
- Two mini projects with step-by-step description and user manual.
- Reading material on C

This book is designed for all those who wish to be C programmers, regardless of their past knowledge and experience in programming. It explains in a simple and easy-to-understand style the what, why and how of programming with ANSI C.

E Balagurusamy

Feedback

Suggestions and constructive criticism always go a long way in enhancing any endeavour. We request all readers to email us their valuable comments/views/feedback for the betterment of the book at tmh.csefeedback@gmail.com mentioning the title and author name in the subject line. Please report any piracy spotted by you as well!

Copyrighted material

1

Overview of C

1.1 HISTORY OF C

'C' seems a strange name for a programming language. But this strange sounding language is one of the most popular computer languages today because it is a structured, high-level, machine independent language. It allows software developers to develop programs without worrying about the hardware platforms where they will be implemented.

The root of all modern languages is ALGOL, introduced in the early 1960s. ALGOL was the first computer language to use a block structure. Although it never became popular in USA, it was widely used in Europe. ALGOL gave the concept of structured programming to the computer science community. Computer scientists like Corrado Bohm, Guiseppe Jacopini and Edsger Dijkstra popularized this concept during 1960s. Subsequently, several languages were announced.

In 1967, Martin Richards developed a language called BCPL (Basic Combined Programming Language) primarily for writing system software. In 1970, Ken Thompson created a language using many features of BCPL and called it simply B. B was used to create early versions of UNIX operating system at Bell Laboratories. Both BCPL and B were "typeless" system programming languages.

C was evolved from ALGOL, BCPL and B by Dennis Ritchie at the Bell Laboratories in 1972. C uses many concepts from these languages and added the concept of data types and other powerful features. Since it was developed along with the UNIX operating system, it is strongly associated with UNIX. This operating system, which was also developed at Bell Laboratories, was coded almost entirely in C. UNIX is one of the most popular network operating systems in use today and the heart of the Internet data superhighway.

For many years, C was used mainly in academic environments, but eventually with the release of many C compilers for commercial use and the increasing popularity of UNIX, it began to gain widespread support among computer professionals. Today, C is running under a variety of operating system and hardware platforms.

During 1970s, C had evolved into what is now known as *"traditional C"*. The language became more popular after publication of the book *'The C Programming Language'* by Brian Kerningham and Dennis Ritchie in 1978. The book was so popular that the language came to be known as "K&R C" among the programming community. The rapid growth of C led to the development of different versions of the language that were similar but often incompatible. This posed a serious problem for system developers.

Copyrighted material

To assure that the C language remains standard, in 1983, American National Standards Institute (ANSI) appointed a technical committee to define a standard for C. The committee approved a version of C in December 1989 which is now known as ANSI C. It was then approved by the International Standards Organization (ISO) in 1990. This version of C is also referred to as C89.

During 1990's, C++, a language entirely based on C, underwent a number of improvements and changes and became an ANSI/ISO approved language in November 1977. C++ added several new features to C to make it not only a true object-oriented language but also a more versatile language. During the same period, Sun Microsystems of USA created a new language **Java** modelled on C and C++.

All popular computer languages are dynamic in nature. They continue to improve their power and scope by incorporating new features and C is no exception. Although C++ and Java were evolved out of C, the standardization committee of C felt that a few features of C++/Java, if added to C, would enhance the usefulness of the language. The result was the 1999 standard for C. This version is usually referred to as C99. The history and development of C is illustrated in Fig. 1.1.

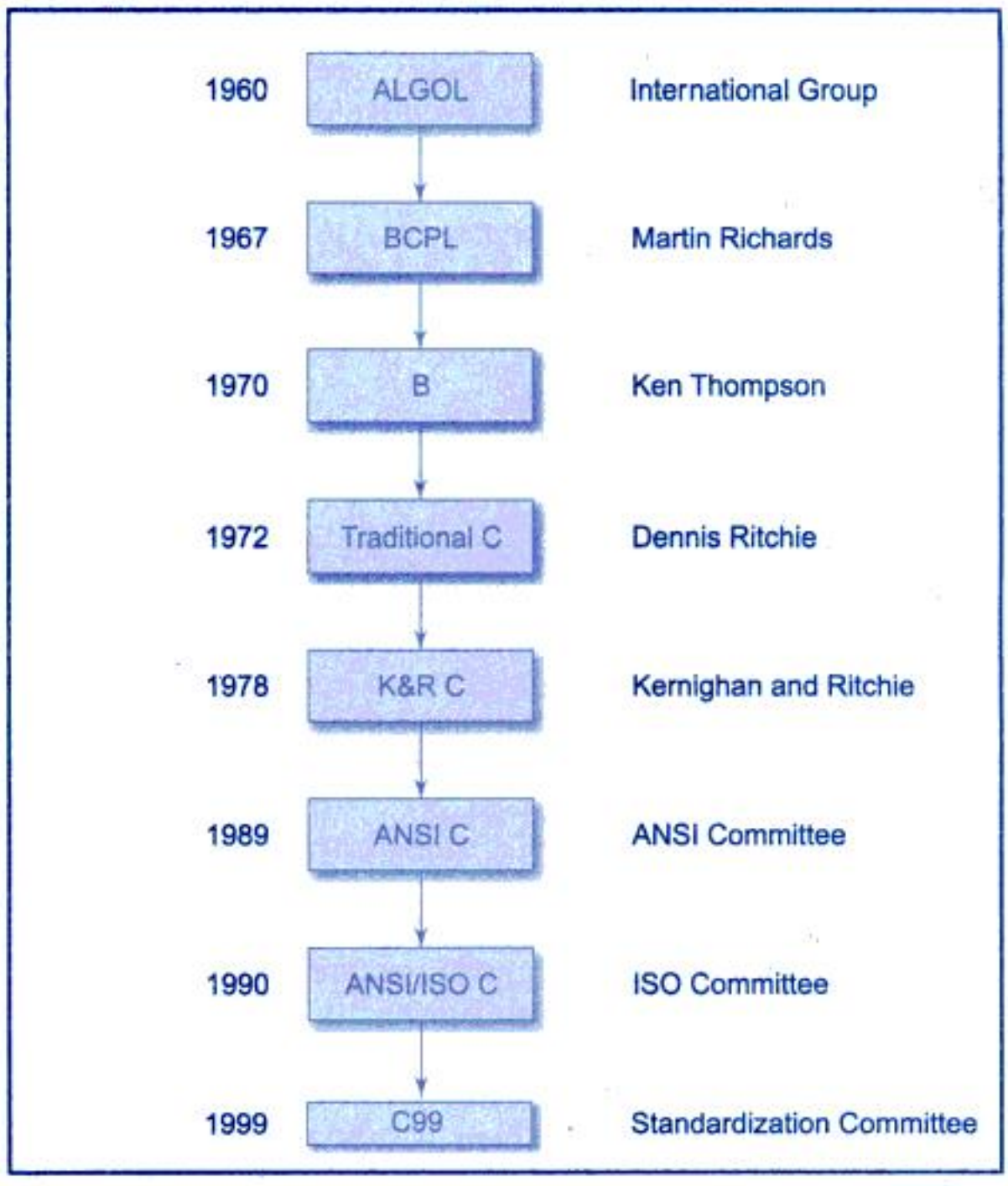


Fig. 1.1 *History of ANSI C*

Copyrighted material

Although C99 is an improved version, still many commonly available compilers do not support all of the new features incorporated in C99. We, therefore, discuss all the new features added by C99 in an appendix separately so that the readers who are interested can quickly refer to the new material and use them wherever possible.

1.2 IMPORTANCE OF C

The increasing popularity of C is probably due to its many desirable qualities. It is a robust language whose rich set of built-in functions and operators can be used to write any complex program. The C compiler combines the capabilities of an assembly language with the features of a high-level language and therefore it is well suited for writing both system software and business packages. In fact, many of the C compilers available in the market are written in C.

Programs written in C are efficient and fast. This is due to its variety of data types and powerful operators. It is many times faster than BASIC. For example, a program to increment a variable from 0 to 15000 takes about one second in C while it takes more than 50 seconds in an interpreter BASIC.

There are only 32 keywords in ANSI C and its strength lies in its built-in functions. Several standard functions are available which can be used for developing programs.

C is highly portable. This means that C programs written for one computer can be run on another with little or no modification. Portability is important if we plan to use a new computer with a different operating system.

C language is well suited for structured programming, thus requiring the user to think of a problem in terms of function modules or blocks. A proper collection of these modules would make a complete program. This modular structure makes program debugging, testing and maintenance easier.

Another important feature of C is its ability to extend itself. A C program is basically a collection of functions that are supported by the C library. We can continuously add our own functions to C library. With the availability of a large number of functions, the programming task becomes simple.

Before discussing specific features of C, we shall look at some sample C programs, and analyze and understand how they work.

1.3 SAMPLE PROGRAM I: PRINTING A MESSAGE

Consider a very simple program given in Fig. 1.2.

```
main( )
{
/*.........printing begins.............*/
    printf("I see, I remember");
/*.............printing ends.............*/
}
```

Fig. 1.2 *A program to print one line of text*

Copyrighted material

This program when executed will produce the following output:

```
I see, I remember
```

Let us have a close look at the program. The first line informs the system that the name of the program is **main** and the execution begins at this line. The **main( )** is a special function used by the C system to tell the computer where the program starts. Every program must have *exactly one main* function. If we use more than one **main** function, the compiler cannot understand which one marks the beginning of the program.

The empty pair of parentheses immediately following **main** indicates that the function **main** has no *arguments* (or parameters). The concept of arguments will be discussed in detail later when we discuss functions (in Chapter 9).

The opening brace "{ " in the second line marks the beginning of the function **main** and the closing brace "}" in the last line indicates the end of the function. In this case, the closing brace also marks the end of the program. All the statements between these two braces form the *function body*. The function body contains a set of instructions to perform the given task.

In this case, the function body contains three statements out of which only the **printf** line is an executable statement. The lines beginning with **/*** and ending with ***/** are known as *comment* lines. These are used in a program to enhance its readability and understanding. Comment lines are not executable statements and therefore anything between **/*** and ***/** is ignored by the compiler. In general, a comment can be inserted wherever blank spaces can occur—at the beginning, middle or end of a line—"but never in the middle of a word ".

Although comments can appear anywhere, they cannot be nested in C. That means, we cannot have comments inside comments. Once the compiler finds an opening token, it ignores everything until it finds a closing token. The comment line

```
/* = = = =/* = = = = */ = = = = */
```

is not valid and therefore results in an error.

Since comments do not affect the execution speed and the size of a compiled program, we should use them liberally in our programs. They help the programmers and other users in understanding the various functions and operations of a program and serve as an aid to debugging and testing. We shall see the use of comment lines more in the examples that follow.

Let us now look at the **printf( )** function, the only executable statement of the program.

```
printf("I see, I remember");
```

printf is a predefined standard C function for printing output. *Predefined* means that it is a function that has already been written and compiled, and linked together with our program at the time of linking. The concepts of compilation and linking are explained later in this chapter. The **printf** function causes everything between the starting and the ending quotation marks to be printed out. In this case, the output will be:

```
I see, I remember
```

Note that the print line ends with a semicolon. *Every statement in C should end with a semicolon (;) mark.*

Suppose we want to print the above quotation in two lines as

```
I see,
I remember!
```

This can be achieved by adding another **printf** function as shown below:

Copyrighted material

```
printf("I see, \n");
printf("I remember !");
```

The information contained between the parentheses is called the *argument* of the function. This argument of the first **printf** function is " I see, \n" and the second is "I remember !". These arguments are simply strings of characters to be printed out.

Notice that the argument of the first **printf** contains a combination of two characters \ and **n** at the end of the string. This combination is collectively called the *newline* character. A newline character instructs the computer to go to the next (new) line. It is similar in concept to the carriage return key on a typewriter. After printing the character comma (,) the presence of the newline character **\n** causes the string "I remember !" to be printed on the next line. No space is allowed between \ and n.

If we omit the newline character from the first **printf** statement, then the output will again be a single line as shown below.

```
I see, I remember !
```

This is similar to the output of the program in Fig. 1.2. However, note that there is no space between , and I.

It is also possible to produce two or more lines of output by one **printf** statement with the use of newline character at appropriate places. For example, the statement

```
printf("I see,\n I remember !");
```

will output

```
I see,
 I remember !
```

while the statement

```
printf( "I\n.. see,\n… … … I\n… … … remember !");
```

will print out

```
I
.. see,
… … … I
… … … remember !
```

NOTE: Some authors recommend the inclusion of the statement

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

at the beginning of all programs that use any input/output library functions. However, this is not necessary for the functions ***printf*** and ***scanf*** which have been defined as a part of the C language. See Chapter 4 for more on input and output functions.

Before we proceed to discuss further examples, we must note one important point. C does make a distinction between *uppercase* and *lowercase* letters. For example, **printf** and **PRINTF** are not the same. In C, everything is written in lowercase letters. However, uppercase letters are used for symbolic names representing constants. We may also use uppercase letters in output strings like "I SEE" and "I REMEMBER"

The above example that printed **I see, I remember** is one of the simplest programs. Figure 1.3 highlights the general format of such simple programs. All C programs need a **main** function.

Copyrighted material

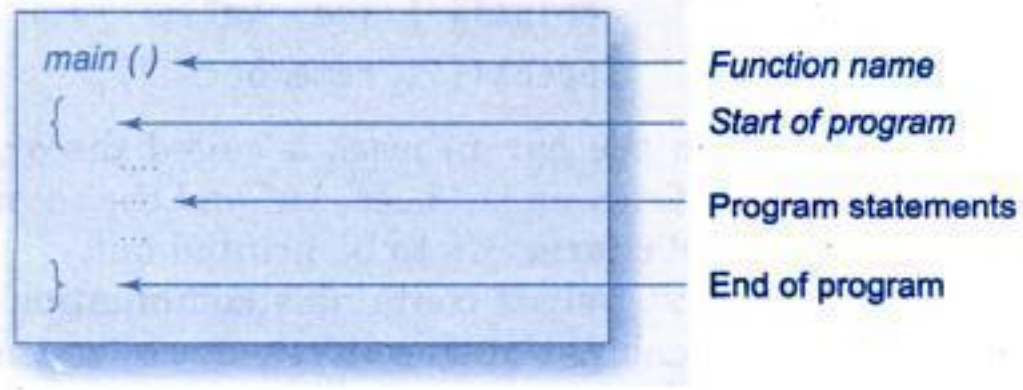


Fig. 1.3 *Format of simple C programs*

The main Function

The main is a part of every C program. C permits different forms of main state ment. Following forms are allowed.

- main()
- int main()
- void main()
- main(void)
- void main(void)
- int main(void)

The empty pair of parentheses indicates that the function has no arguments. This may be explicitly indicated by using the keyword **void** inside the parentheses. We may also specify the keyword **int** or **void** before the word **main**. The keyword **void** means that the function does not return any information to the operating system and **int** means that the function returns an integer value to the operating system. When **int** is specified, the last statement in the program must be "return 0". For the sake of simplicity, we use the first form in our programs.

1.4 SAMPLE PROGRAM 2: ADDING TWO NUMBERS

Consider another program, which performs addition on two numbers and displays the result. The complete program is shown in Fig. 1.4.

```
/* Programm ADDITION                    line-1 */
/* Written by EBG                       line-2 */
main()                               /*  line-3 */
{                                    /*  line-4 */
```

Copyrighted material

```
        int number;                                  /*  line-5 */
        float amount;                                /*  line-6 */
                                                     /*  line-7 */
        number = 100;                                /*  line-8 */
                                                     /*  line-9 */
        amount = 30.75 + 75.35;                      /* line-10 */
        printf("%d\n",number);                       /* line-11 */
        printf("%5.2f",amount);                      /* line-12 */
}                                                    /* line-13 */
```

Fig. 1.4 *Program to add two numbers*

This program when executed will produce the following output:

```
100
106.10
```

The first two lines of the program are comment lines. It is a good practice to use comment lines in the beginning to give information such as name of the program, author, date, etc. Comment characters are also used in other lines to indicate line numbers.

The words **number** and **amount** are *variable names* that are used to store numeric data. The numeric data may be either in *integer* form or in *real* form. In C, *all variables should be declared* to tell the compiler what the ***variable names*** are and what ***type of data*** they hold. The variables must be declared before they are used. In lines 5 and 6, the declarations

```
int number;
float amount;
```

tell the compiler that **number** is an integer (**int**) and **amount** is a floating (**float**) point number. Declaration statements must appear at the beginning of the functions as shown in Fig.1.4. All declaration statements end with a semicolon; C supports many other data types and they are discussed in detail in Chapter 2.

The words such as **int** and **float** are called the *keywords* and cannot be used as *variable* names. A list of keywords is given in Chapter 2.

Data is stored in a variable by *assigning* a data value to it. This is done in lines 8 and 10. In line-8, an integer value 100 is assigned to the integer variable **number** and in line-10, the result of addition of two real numbers 30.75 and 75.35 is assigned to the floating point variable **amount.** The statements

```
number = 100;
amount = 30.75 + 75.35;
```

are called the *assignment* statements. Every assignment statement must have a semicolon at the end.

The next statement is an output statement that prints the value of **number**. The print statement

```
printf("%d\n", number);
```

contains two arguments. The first argument "%d" tells the compiler that the value of the second argument **number** should be printed as a *decimal integer*. Note that these arguments are separated by a comma. The newline character \n causes the next output to appear on a new line.

Copyrighted material

The last statement of the program

```
printf("%5.2f", amount);
```

prints out the value of **amount** in floating point format. The format specification %5.2*f* tells the compiler that the output must be in *floating point,* with five places in all and two places to the right of the decimal point.

1.5 SAMPLE PROGRAM 3: INTEREST CALCULATION

The program in Fig. 1.5 calculates the value of money at the end of each year of investment, assuming an interest rate of 11 percent and prints the year, and the corresponding amount, in two columns. The output is shown in Fig. 1.6 for a period of 10 years with an initial investment of 5000.00. The program uses the following formula:

Value at the end of year = Value at start of year (1 + interest rate)

In the program, the variable **value** represents the value of money at the end of the year while **amount** represents the value of money at the start of the year. The statement

```
amount = value ;
```

makes the value at the end of the *current* year as the value at start of the *next* year.

```
/*———————— INVESTMENT PROBLEM ————————*/
#define PERIOD        10
#define PRINCIPAL   5000.00
/*————————— MAIN PROGRAM BEGINS —————————*/
main()
{ /*———————— DECLARATION STATEMENTS ————————*/
      int year;
      float amount, value, inrate;
/*————————— ASSIGNMENT STATEMENTS —————————*/
      amount = PRINCIPAL;
      inrate = 0.11;
      year = 0;
/*————————— COMPUTATION STATEMENTS —————————*/
/*———————— COMPUTATION USING While LOOP ————————*/
      while(year <= PERIOD)
      { printf("%2d   %8.2f\n",year, amount);
          value = amount + inrate * amount;
           year = year + 1;
         amount = value;
      }
/*————————— while LOOP ENDS —————————*/
}
/*————————— PROGRAM ENDS —————————*/
```

Fig. 1.5 *Program for investment problem*

Copyrighted material

Let us consider the new features introduced in this program. The second and third lines begin with **#define** instructions. A **#define** instruction defines value to a *symbolic constant* for use in the program. Whenever a symbolic name is encountered, the compiler substitutes the value associated with the name automatically. To change the value, we have to simply change the definition. In this example, we have defined two symbolic constants **PERIOD** and **PRINCIPAL** and assigned values 10 and 5000.00 respectively. These values remain constant throughout the execution of the program.

```
         0        5000.00
         1        5550.00
         2        6160.50
         3        6838.15
         4        7590.35
         5        8425.29
         6        9352.07
         7       10380.00
         8       11522.69
         9       12790.00
        10       14197.11
```

Fig. 1.6 *Output of the investment program*

The #define Directive

A **#define** is a preprocessor compiler directive and not a statement. Therefore **#define** lines should not end with a semicolon. Symbolic constants are generally written in uppercase so that they are easily distinguished from lowercase variable names. **#define** instructions are usually placed at the beginning before the **main()** function. Symbolic constants are not declared in declaration section. Preprocessor directives are discussed in Chapter 14.

We must note that the defined constants are not variables. We may not change their values within the program by using an assignment statement. For example, the statement

```
PRINCIPAL = 10000.00;
```

is illegal.

The declaration section declares **year** as integer and **amount, value** and **inrate** as floating point numbers. Note all the floating-point variables are declared in one statement. They can also be declared as

Copyrighted material

```
float amount;
float value;
float inrate;
```

When two or more variables are declared in one statement, they are separated by a comma.

All computations and printing are accomplished in a **while** loop. **while** is a mechanism for evaluating repeatedly a statement or a group of statements. In this case as long as the value of **year** is less than or equal to the value of **PERIOD**, the four statements that follow **while** are executed. Note that these four statements are grouped by braces. We exit the loop when **year** becomes greater than **PERIOD**. The concept and types of loops are discussed in Chapter 6.

C supports the basic four arithmetic operators (–, +, *, /) along with several others. They are discussed in Chapter 3.

1.6 SAMPLE PROGRAM 4: USE OF SUBROUTINES

So far, we have used only **printf** function that has been provided for us by the C system. The program shown in Fig. 1.7 uses a user-defined function. A function defined by the user is equivalent to a subroutine in FORTRAN or subprogram in BASIC.

Figure 1.7 presents a very simple program that uses a **mul ( )** function. The program will print the following output.

Multiplication of 5 and 10 is 50

```
/*———————— PROGRAM USING FUNCTION ————————*/
int mul (int a, int b); /*—— DECLARATION ———*/
/*————————— MAIN PROGRAM BEGINS —————————*/
    main ()
    {
        int a, b, c;
        a = 5;
        b = 10;
        c = mul (a,b);

        printf ("multiplication of %d and %d is %d",a,b,c);
    }
/* ————————      MAIN PROGRAM ENDS
                  MUL() FUNCTION STARTS ——————————*/
      int mul (int x, int y)
      int p;
      {
            p = x*y;
            return(p);
      }
/* ——————————— MUL () FUNCTION ENDS ———————————*/
```

Fig. 1.7 *A program using a user-defined function*

Copyrighted material

The **mul ( )** function multiplies the values of **x** and **y** and the result is returned to the **main ( )** function when it is called in the statement

```
c = mul (a, b);
```

The **mul ( )** has two *arguments* **x** and **y** that are declared as integers. The values of **a** and **b** are passed on to **x** and **y** respectively when the function **mul ( )** is called. User-defined functions are considered in detail in Chapter 9.

1.7 SAMPLE PROGRAM 5: USE OF MATH FUNCTIONS

We often use standard mathematical functions such as cos, sin, exp, etc. We shall see now the use of a mathematical function in a program. The standard mathematical functions are defined and kept as a part of C **math library**. If we want to use any of these mathematical functions, we must add an **#include** instruction in the program. Like **#define,** it is also a compiler directive that instructs the compiler to link the specified mathematical functions from the library. The instruction is of the form

```
#include <math.h>
```

math.h is the filename containing the required function. Figure 1.8 illustrates the use of cosine function. The program calculates cosine values for angles 0, 10, 20.....180 and prints out the results with headings.

```
/*------- PROGRAM USING COSINE FUNCTION ------- */
#include <math.h>
#define  PI 3.1416
#define  MAX  180

main ( )
{
     int angle;
     float x,y;

     angle = 0;
     printf(" Angle    Cos(angle)\n\n");

     while(angle <= MAX)
     {
          x = (PI/MAX)*angle;
          y = cos(x);
          printf("%15d %13.4f\n", angle, y);
          angle = angle + 10;
     }
}
```

Output

```
      Angle              Cos(angle)
          0                 1.0000
         10                 0.9848
         20                 0.9397
         30                 0.8660
```

Copyrighted material

```
 40        0.7660
 50        0.6428
 60        0.5000
 70        0.3420
 80        0.1736
 90       -0.0000
100       -0.1737
110       -0.3420
120       -0.5000
130       -0.6428
140       -0.7660
150       -0.8660
160       -0.9397
170       -0.9848
180       -1.0000
```

Fig. 1.8 *Program using a math function*

Another **#include** instruction that is often required is

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

stdio.h refers to the *standard* I/O header file containing standard input and output functions

The #include Directive

As mentioned earlier, C programs are divided into modules or functions. Some functions are written by users, like us, and many others are stored in the C library. Library functions are grouped category-wise and stored in different files known as *header files*. If we want to access the functions stored in the library, it is necessary to tell the compiler about the files to be accessed.

This is achieved by using the preprocessor directive **#include** as follows:

#include<filename>

filename is the name of the library file that contains the required function definition. Preprocessor directives are placed at the beginning of a program.

A list of library functions and header files containing them are given in Appendix III.

1.8 BASIC STRUCTURE OF C PROGRAMS

The examples discussed so far illustrate that a C program can be viewed as a group of building blocks called *functions*. A function is a subroutine that may include one or more *state-*

Copyrighted material

ments designed to perform a *specific task*. To write a C program, we first create functions and then put them together. A C program may contain one or more sections as shown in Fig. 1.9.

The documentation section consists of a set of comment lines giving the name of the program, the author and other details, which the programmer would like to use later. The link section provides instructions to the compiler to link functions from the system library. The definition section defines all symbolic constants.

There are some variables that are used in more than one function. Such variables are called *global* variables and are declared in the *global* declaration section that is outside of all the functions. This section also declares all the user-defined functions.

Every C program must have one **main()** function section. This section contains two parts, declaration part and executable part. The declaration part declares all the variables used in the executable part. There is at least one statement in the executable part. These two parts must appear between the opening and the closing braces. The program execution begins at the opening brace and ends at the closing brace. The closing brace of the main function section is the logical end of the program. All statements in the declaration and executable parts end with a semicolon(;).

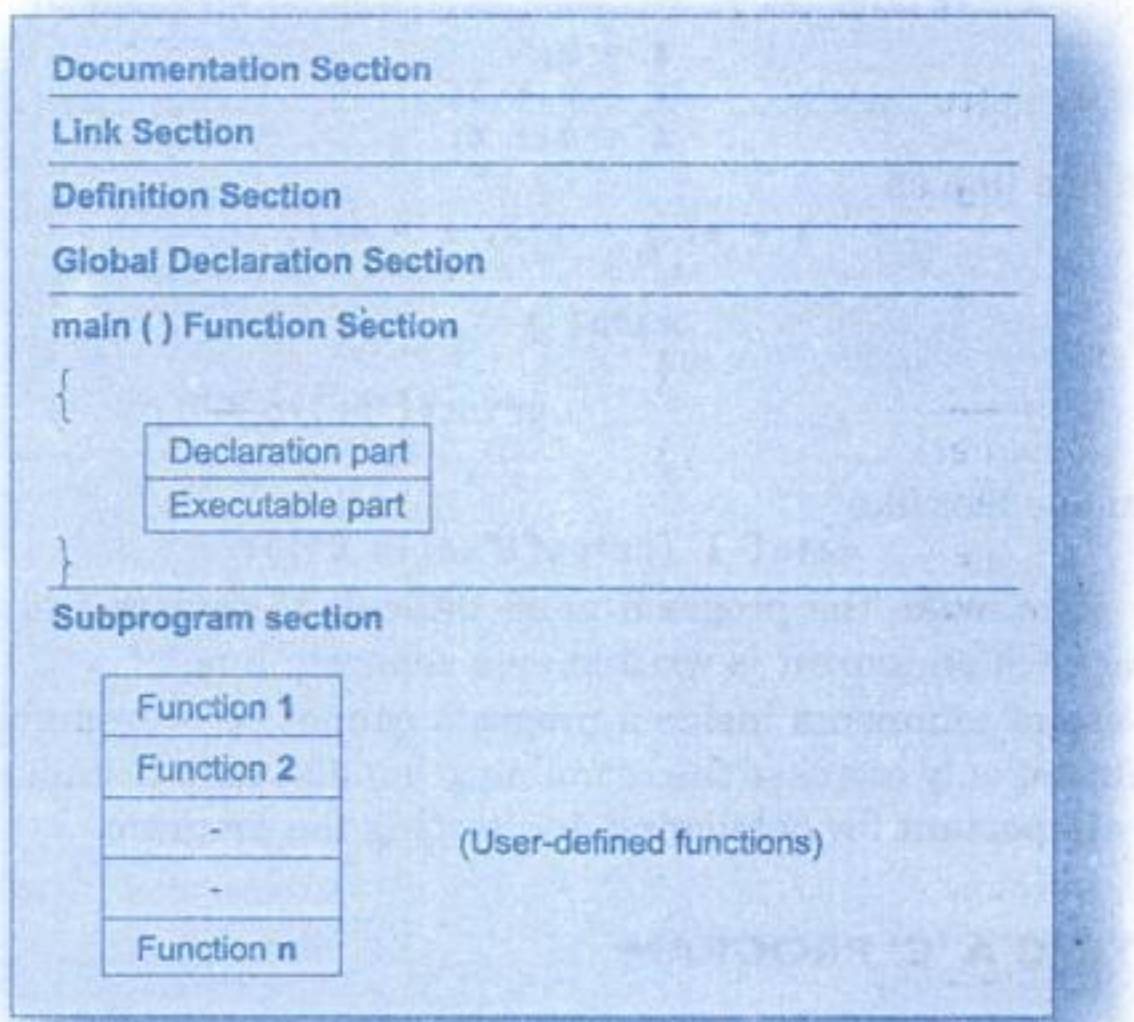


Fig. 1.9 *An overview of a C program*

The subprogram section contains all the user-defined functions that are called in the **main** function. User-defined functions are generally placed immediately after the **main** function, although they may appear in any order.

Copyrighted material

All sections, except the **main** function section may be absent when they are not required.

1.9 PROGRAMMING STYLE

Unlike some other programming languages (COBOL, FORTRAN, etc.,) C is a *free-form* language. That is, the C compiler does not care, where on the line we begin typing. While this may be a licence for bad programming, we should try to use this fact to our advantage in developing readable programs. Although several alternative styles are possible, we should select one style and use it with total consistency.

First of all, we must develop the habit of writing programs in lowercase letters. C program statements are written in lowercase letters. Uppercase letters are used only for symbolic constants.

Braces, group program statements together and mark the beginning and the end of functions. A proper indentation of braces and statements would make a program easier to read and debug. Note how the braces are aligned and the statements are indented in the program of Fig. 1.5.

Since C is a free-form language, we can group statements together on one line. The statements

```
a = b;
x = y + 1;
z = a + x;
```

can be written on one line as

```
a = b; x = y+1; z = a+x;
```

The program

```
main( )
{
        printf("hello C");
}
```

may be written in one line like

```
main( ) {printf("Hello C")};
```

However, this style make the program more difficult to understand and should not be used. In this book, each statement is written on a separate line.

The generous use of comments inside a program cannot be overemphasized. Judiciously inserted comments not only increase the readability but also help to understand the program logic. This is very important for debugging and testing the program.

1.10 EXECUTING A 'C' PROGRAM

Executing a program written in C involves a series of steps. These are:

1. Creating the program;
2. Compiling the program;
3. Linking the program with functions that are needed from the C library; and
4. Executing the program.

Figure 1.10 illustrates the process of creating, compiling and executing a C program. Although these steps remain the same irrespective of the *operating system*, system

Copyrighted material

commands for implementing the steps and conventions for naming *files* may differ on different systems.

An operating system is a program that controls the entire operation of a computer system. All input/output operations are channeled through the operating system. The operating system, which is an interface between the hardware and the user, handles the execution of user programs.

The two most popular operating systems today are UNIX (for minicomputers) and MS-DOS (for microcomputers). We shall discuss briefly the procedure to be followed in executing C programs under both these operating systems in the following sections.

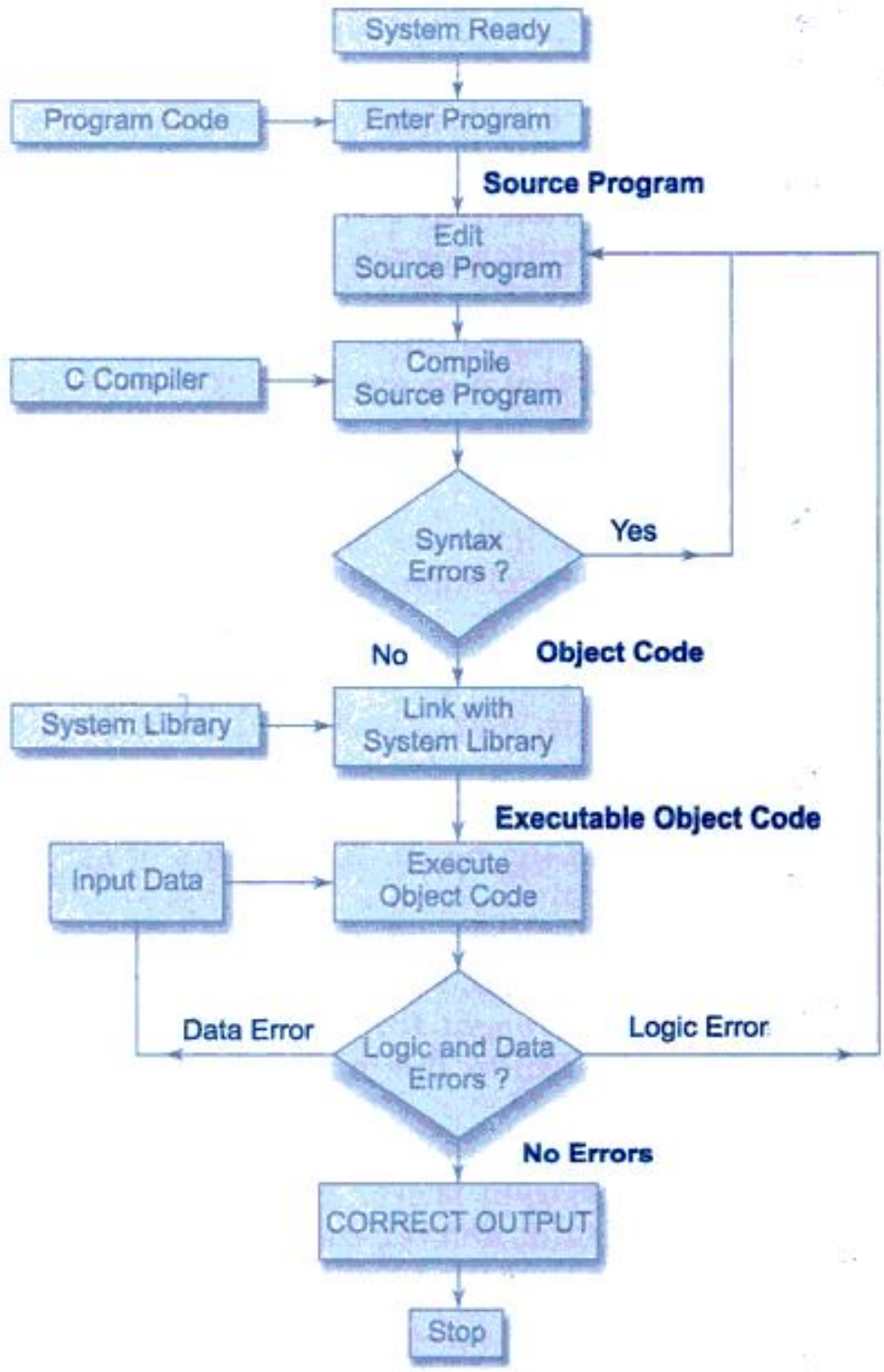


Fig. 1.10 *Process of compiling and running a C program*

Copyrighted material

1.11 UNIX SYSTEM

Creating the program

Once we load the UNIX operating system into the memory, the computer is ready to receive program. The program must be entered into a file. The file name can consist of letters, digits and special characters, followed by a dot and a letter **c**. Examples of valid file names are:

```
hello.c
program.c
ebg1.c
```

The file is created with the help of a *text editor*, either **ed** or **vi.** The command for calling the editor and creating the file is

```
ed filename
```

If the file existed before, it is loaded. If it does not yet exist, the file has to be created so that it is ready to receive the new program. Any corrections in the program are done under the editor. (The name of your system's editor may be different. Check your system manual.)

When the editing is over, the file is saved on disk. It can then be referenced any time later by its file name. The program that is entered into the file is known as the *source program*, since it represents the original form of the program.

Compiling and Linking

Let us assume that the source program has been created in a file named *ebg1.c*. Now the program is ready for compilation. The compilation command to achieve this task under UNIX is

```
cc ebg1.c
```

The source program instructions are now translated into a form that is suitable for execution by the computer. The translation is done after examining each instruction for its correctness. If everything is alright, the compilation proceeds silently and the translated program is stored on another file with the name *ebg1.o*. This program is known as *object code*.

Linking is the process of putting together other program files and functions that are required by the program. For example, if the program is using **exp()** function, then the object code of this function should be brought from the **math library** of the system and linked to the main program. Under UNIX, the linking is automatically done (if no errors are detected) when the **cc** command is used.

If any mistakes in the *syntax* and *semantics* of the language are discovered, they are listed out and the compilation process ends right there. The errors should be corrected in the source program with the help of the editor and the compilation is done again.

The compiled and linked program is called the *executable object code* and is stored automatically in another file named **a.out.**

Note that some systems use different compilation command for linking mathematical functions.

```
cc filename - lm
```

is the command under UNIPLUS SYSTEM V operating system.

Copyrighted material

Executing the Program

Execution is a simple task. The command

```
a.out
```

would load the executable object code into the computer memory and execute the instructions. During execution, the program may request for some data to be entered through the keyboard. Sometimes the program does not produce the desired results. Perhaps, something is wrong with the program *logic* or *data*. Then it would be necessary to correct the source program or the data. In case the source program is modified, the entire process of compiling, linking and executing the program should be repeated.

Creating Your Own Executable File

Note that the linker always assigns the same name **a.out**. When we compile another program, this file will be overwritten by the executable object code of the new program. If we want to prevent from happening, we should rename the file immediately by using the command.

```
mv a.out name
```

We may also achieve this by specifying an option in the cc command as follows:

```
cc -o name source-file
```

This will store the executable object code in the file name and prevent the old file **a.out** from being destroyed.

Multiple Source Files

To compile and link multiple source program files, we must append all the files names to the **cc** command.

```
cc filename-1.c .... filename-n.c
```

These files will be separately compiled into object files called

```
filename-i.o
```

and then linked to produce an executable program file **a.out** as shown in Fig. 1.11.

It is also possible to compile each file separately and link them later. For example, the commands

```
cc -c mod1.c
cc -c mod2.c
```

will compile the source files *mod1.c* and *mod2.c* into objects files *mod1.o* and *mod2.o*. They can be linked together by the command

```
cc mod1.o mod2.o
```

we may also combine the source files and object files as follows:

```
cc mod1.c mod2.o
```

Only *mod1.c* is compiled and then linked with the object file mod2.o. This approach is useful when one of the multiple source files need to be changed and recompiled or an already existing object files is to be used along with the program to be compiled.

Copyrighted material

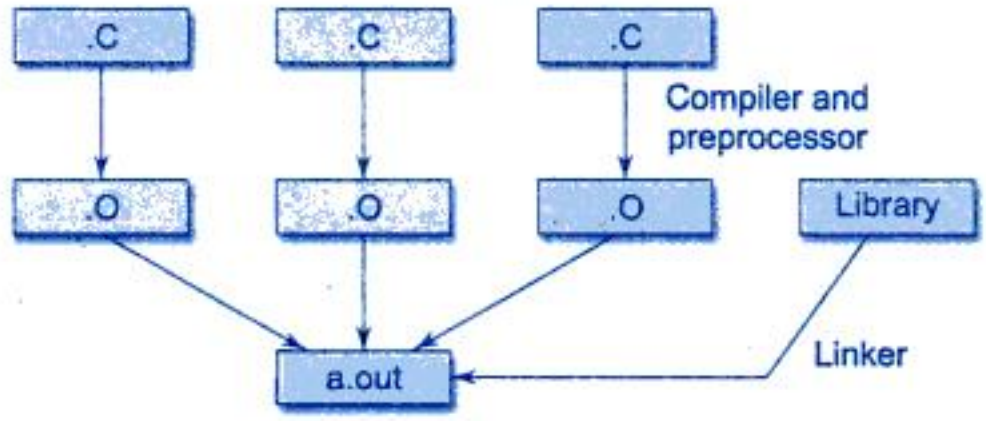


Fig. 1.11 *Compilation of multiple files*

1.12 MS-DOS SYSTEM

The program can be created using any word processing software in non-document mode. The file name should end with the characters ".c" like **program.c, pay.c,** etc. Then the command

```
MSC pay.c
```

under MS-DOS operating system would load the program stored in the file **pay.c** and generate the **object code.** This code is stored in another file under name **pay.obj**. In case any language errors are found, the compilation is not completed. The program should then be corrected and compiled again.

The linking is done by the command

```
LINK pay.obj
```

which generates the **executable code** with the filename **pay.exe**. Now the command

```
pay
```

would execute the program and give the results.

Just Remember

- Every C program requires a **main()** function (Use of more than one **main()** is illegal). The place **main** is where the program execution begins.
- The execution of a function begins at the opening brace of the function and ends at the corresponding closing brace.
- C programs are written in lowercase letters. However, uppercase letters are used for symbolic names and output strings.
- All the words in a program line must be separated from each other by at least one space, or a tab, or a punctuation mark.
- Every program statement in a C language must end with a semicolon.
- All variables must be declared for their types before they are used in the program.
- We must make sure to include header files using **#include** directive when the program refers to special names and functions that it does not define.
- Compiler directives such as **define** and **include** are special instructions

Copyrighted material

to the compiler to help it compile a program. They do not end with a semicolon.

- The sign # of compiler directives must appear in the first column of the line.
- When braces are used to group statements, make sure that the opening brace has a corresponding closing brace.
- C is a free-form language and therefore a proper form of indentation of various sections would improve legibility of the program.
- A comment can be inserted almost anywhere a space can appear. Use of appropriate comments in proper places increases readability and understandability of the program and helps users in debugging and testing. Remember to match the symbols /* and */ appropriately.

Review Questions

1.1 State whether the following statements are *true* or *false*.
   (a) Every line in a C program should end with a semicolon.
   (b) In C language lowercase letters are significant.
   (c) Every C program ends with an END word.
   (d) **main( )** is where the program begins its execution.
   (e) A line in a program may have more than one statement.
   (f) A **printf** statement can generate only one line of output.
   (g) The closing brace of the **main( )** in a program is the logical end of the program.
   (h) The purpose of the header file such as **stdio.h** is to store the source code of a program.
   (i) Comments cause the computer to print the text enclosed between /* and */ when executed.
   (j) Syntax errors will be detected by the compiler.

1.2 Which of the following statements are *true*?
   (a) Every C program must have at least one user-defined function.
   (b) Only one function may be named **main( )**.
   (c) Declaration section contains instructions to the computer.

1.3 Which of the following statements about comments are *false*?
   (a) Use of comments reduces the speed of execution of a program.
   (b) Comments serve as internal documentation for programmers.
   (c) A comment can be inserted in the middle of a statement.
   (d) In C, we can have comments inside comments.

1.4 Fill in the blanks with appropriate words in each of the following statements.
   (a) Every program statement in a C program must end with a _____
   (b) The _____ Function is used to display the output on the screen.
   (c) The _____ header file contains mathematical functions.
   (d) The escape sequence character _____ causes the cursor to move to the next line on the screen.

1.5 Remove the semicolon at the end of the **printf** statement in the program of Fig. 1.2 and execute it. What is the output?

Copyrighted material

1.6 In the Sample Program 2, delete line-5 and execute the program. How helpful is the error message?

1.7 Modify the Sample Program 3 to display the following output:

Year	Amount
1	5500.00
2	6160.00
-	———
-	———
10	14197.11

1.8 Find errors, if any, in the following program:

```
/* A simple program
int main( )
{
    /* Does nothing */
}
```

1.9 Find errors, if any, in the following program:

```
#include (stdio.h)
void main(void)
{
    print("Hello C");
}
```

1.10 Find errors, if any, in the following program:

```
Include <math.h>
main { }
(
    FLOAT X;
    X = 2.5;
    Y = exp(x);
    Print(x,y);
)
```

1.11 Why and when do we use the **#define** directive?

1.12 Why and when do we use the **#include** directive?

1.13 What does **void main(void)** mean?

1.14 Distinguish between the following pairs:

(a) main( ) and void main(void)

(b) int main( ) and void main( )

1.15 Why do we need to use comments in programs?

1.16 Why is the look of a program is important?

1.17 Where are blank spaces permitted in a C program?

1.18 Describe the structure of a C program.

1.19 Describe the process of creating and executing a C program under UNIX system.

1.20 How do we implement multiple source program files?

Programming Exercises

1.1 Write a program that will print your mailing address in the following form:

First line : Name

Copyrighted material

Second line : Door No, Street
Third line : City, Pin code

1.2 Modify the above program to provide border lines to the address.

1.3 Write a program using one print statement to print the pattern of asterisks as shown below:

```
*
*   *
*   *   *
*   *   *   *
```

1.4 Write a program that will print the following figure using suitable characters.

1.5 Given the radius of a circle, write a program to compute and display its area. Use a symbolic constant to define the π value and assume a suitable value for radius.

1.6 Write a program to output the following multiplication table:

$5 \times 1 = 5$

$5 \times 2 = 10$

$5 \times 3 = 15$

. .

. .

$5 \times 10 = 50$

1.7 Given two integers 20 and 10, write a program that uses a function add( ) to add these two numbers and sub( ) to find the difference of these two numbers and then display the sum and difference in the following form:

$20 + 10 = 30$

$20 - 10 = 10$

1.8 Given the values of three variables a, b and c, write a program to compute and display the value of x, where

$$x = \frac{a}{b - c}$$

Execute your program for the following values:
(a) a = 250, b = 85, c = 25
(b) a = 300, b = 70, c = 70
Comment on the output in each case.

1.9 Relationship between Celsius and Fahrenheit is governed by the formula

$$F = \frac{9C}{5} + 32$$

Write a program to convert the temperature

Copyrighted material

(a) from Celsius to Fahrenheit and
(b) from Fahrenheit to Celsius.

1.10 Area of a triangle is given by the formula

$$A = \sqrt{S(S-a)(S-b)(S-c)}$$

Where a, b and c are sides of the triangle and 2S = a + b + c. Write a program to compute the area of the triangle given the values of a, b and c.

1.11 Distance between two points (x_1, y_1) and (x_2, y_2) is governed by the formula

$$D^2 = (x_2 - x_1)^2 + (y_2 - y_1)^2$$

Write a program to compute D given the coordinates of the points.

1.12 A point on the circumference of a circle whose center is (o, o) is (4,5). Write a program to compute perimeter and area of the circle. (Hint: use the formula given in the Ex. 1.11)

1.13 The line joining the points (2,2) and (5,6) which lie on the circumference of a circle is the diameter of the circle. Write a program to compute the area of the circle.

1.14 Write a program to display the equation of a line in the form

$$ax + by = c$$

for a = 5, b = 8 and c = 18.

1.15 Write a program to display the following simple arithmetic calculator

x =		y =	
sum		Difference =	
Product =		Division =	

Copyrighted material

2

Constants, Variables, and Data Types

2.1 INTRODUCTION

A programming language is designed to help process certain kinds of *data* consisting of numbers, characters and strings and to provide useful output known as *information*. The task of processing of data is accomplished by executing a sequence of precise instructions called a *program*. These instructions are formed using certain symbols and words according to some rigid rules known as *syntax rules* (or *grammar*). Every program instruction must confirm precisely to the syntax rules of the language.

Like any other language, C has its own vocabulary and grammar. In this chapter, we will discuss the concepts of constants and variables and their types as they relate to C programming language.

2.2 CHARACTER SET

The characters that can be used to form words, numbers and expressions depend upon the computer on which the program is run. However, a subset of characters is available that can be used on most personal, micro, mini and mainframe computers. The characters in C are grouped into the following categories:

1. Letters
2. Digits
3. Special characters
4. White spaces

The entire character set is given in Table 2.1.

The compiler ignores white spaces unless they are a part of a string constant. White spaces may be used to separate words, but are prohibited between the characters of keywords and identifiers.

Copyrighted material

Trigraph Characters

Many non-English keyboards do not support all the characters mentioned in Table 2.1. ANSI C introduces the concept of "trigraph" sequences to provide a way to enter certain characters that are not available on some keyboards. Each trigraph sequence consists of three characters (two question marks followed by another character) as shown in Table 2.2. For example, if a keyboard does not support square brackets, we can still use them in a program using the trigraphs ??(and ??).

Table 2.1 *C Character Set*

Letters	*Digits*
Uppercase A.....Z Lowercase a.....z	All decimal digits 0 9

Special Characters

, comma	& ampersand
. period	^ caret
; semicolon	* asterisk
: colon	– minus sign
? question mark	+ plus sign
' apostrophe	< opening angle bracket
" quotation mark	(or less than sign)
! exclamation mark	> closing angle bracket
| vertical bar	(or greater than sign)
/ slash	(left parenthesis
\ backslash	) right parenthesis
~ tilde	[left bracket
_ under score	] right bracket
$ dollar sign	{ left brace
% percent sign	} right brace
	# number sign

White Spaces

Blank space
Horizontal tab
Carriage return
New line
Form feed

Table 2.2 *ANSI C Trigraph Sequences*

Trigraph sequence	*Translation*
??=	# number sign
??(	[left bracket
??)	] right bracket
??<	{ left brace
??>	} right brace
??!	| vetical bar
??/	\ back slash
??/	^ caret
??-	~ tilde

Copyrighted material

2.3 C TOKENS

In a passage of text, individual words and punctuation marks are called *tokens*. Similarly, in a C program the smallest individual units are known as C tokens. C has six types of tokens as shown in Fig. 2.1. C programs are written using these tokens and the syntax of the language.

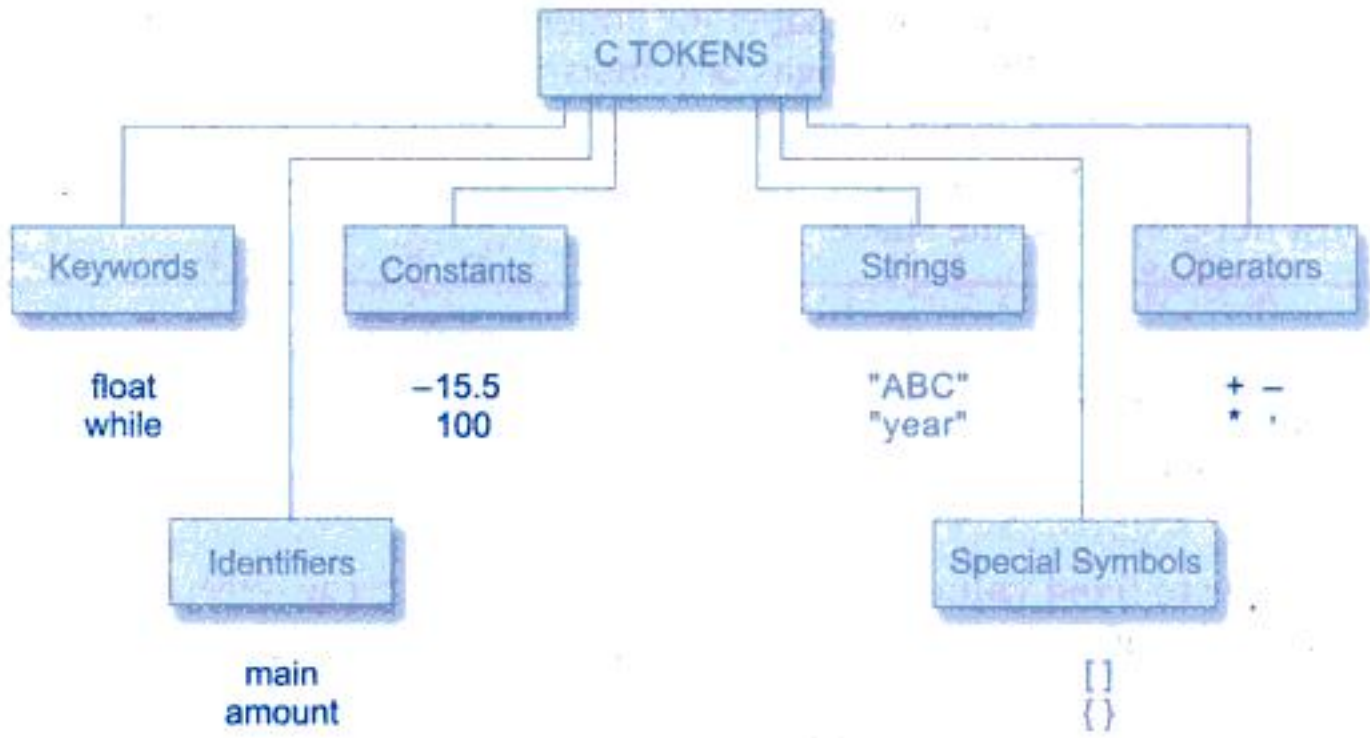


Fig. 2.1 *C tokens and examples*

2.4 KEYWORDS AND IDENTIFIERS

Every C word is classified as either a *keyword* or an *identifier*. All keywords have fixed meanings and these meanings cannot be changed. Keywords serve as basic building blocks for program statements. The list of all keywords of ANSI C are listed in Table 2.3. All keywords must be written in lowercase. Some compilers may use additional keywords that must be identified from the C manual.

NOTE: C99 adds some more keywords. See the Appendix "C99 Features".

Table 2.3 *ANSI C Keywords*

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

Identifiers refer to the names of variables, functions and arrays. These are user-defined names and consist of a sequence of letters and digits, with a letter as a first character. Both

Copyrighted material

uppercase and lowercase letters are permitted, although lowercase letters are commonly used. The underscore character is also permitted in identifiers. It is usually used as a link between two words in long identifiers.

Rules for Identifiers

1. First character must be an alphabet (or underscore).
2. Must consist of only letters, digits or underscore.
3. Only first 31 characters are significant.
4. Cannot use a keyword.
5. Must not contain white space.

2.5 CONSTANTS

Constants in C refer to fixed values that do not change during the execution of a program. C supports several types of constants as illustrated in Fig. 2.2.

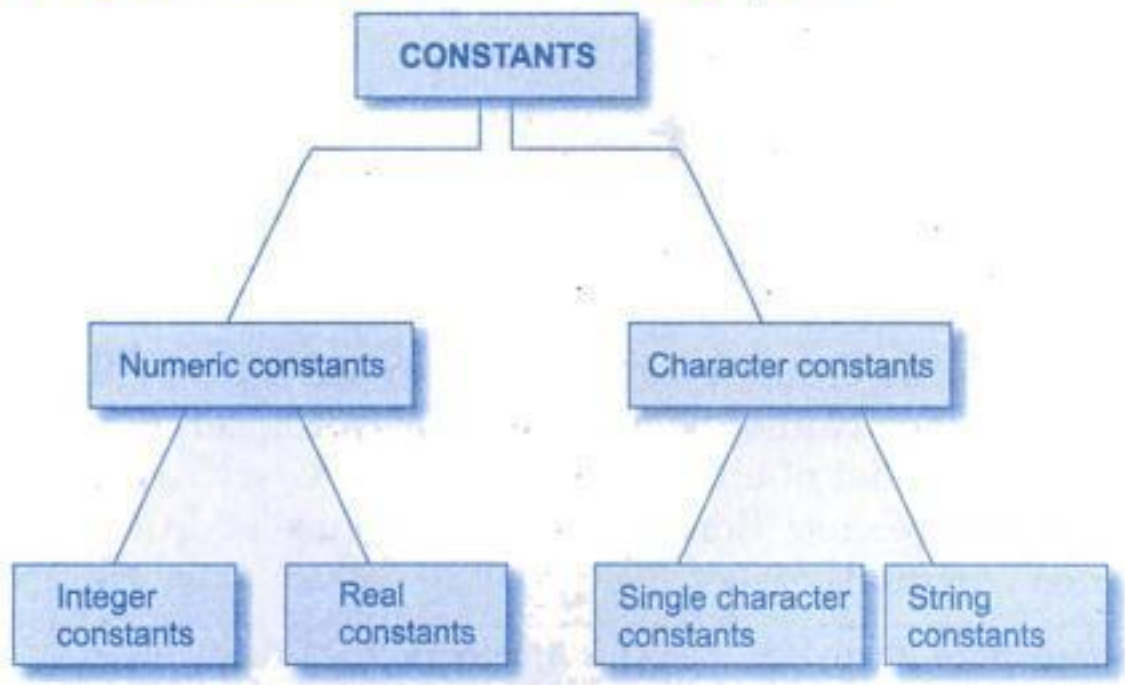


Fig. 2.2 *Basic types of C constants*

Integer Constants

An *integer* constant refers to a sequence of digits. There are three types of integers, namely, *decimal* integer, *octal* integer and *hexadecimal* integer.

Decimal integers consist of a set of digits, 0 through 9, preceded by an optional – or + sign. Valid examples of decimal integer constants are:

123 –321 0 654321 +78

Embedded spaces, commas, and non-digit characters are not permitted between digits. For example,

15 750 20,000 $1000

are illegal numbers.

Copyrighted material

Note: ANSI C supports *unary plus* which was not defined earlier.

An *octal* integer constant consists of any combination of digits from the set 0 through 7, with a leading 0. Some examples of octal integer are:

037 0 0435 0551

A sequence of digits preceded by 0x or 0X is considered as *hexadecimal* integer. They may also include alphabets A through F or a through f. The letter A through F represent the numbers 10 through 15. Following are the examples of valid hex integers:

0X2 0x9F 0Xbcd 0x

We rarely use octal and hexadecimal numbers in programming.

The largest integer value that can be stored is machine-dependent. It is 32767 on 16-bit machines and 2,147,483,647 on 32-bit machines. It is also possible to store larger integer constants on these machines by appending *qualifiers* such as U,L and UL to the constants. Examples:

56789U	or 56789u	(unsigned integer)
987612347UL	or 98761234ul	(unsigned long integer)
9876543L	or 9876543l	(long integer)

The concept of unsigned and long integers are discussed in detail in Section 2.7.

Example 2.1 Representation of integer constants on a 16-bit computer.

The program in Fig.2.3 illustrates the use of integer constants on a 16-bit machine. The output in Fig. 2.3 shows that the integer values larger than 32767 are not properly stored on a 16-bit machine. However, when they are qualified as long integer (by appending L), the values are correctly stored.

```
Program
        main()
        {
            printf("Integer values\n\n");
            printf("%d %d %d\n", 32767,32767+1,32767+10);
            printf("\n");
            printf("Long integer values\n\n");
            printf("%ld %ld %ld\n", 32767L,32767L+1L,32767L+10L);
        }
  Output
        Integer values
        32767 -32768 -32759
        Long integer values
        32767 32768 32777
```

Fig. 2.3 *Representation of integer constants on 16-bit machine*

Real Constants

Integer numbers are inadequate to represent quantities that vary continuously, such as distances, heights, temperatures, prices, and so on. These quantities are represented by numbers containing fractional parts like 17.548. Such numbers are called *real* (or *floating point*) constants. Further examples of real constants are:

Copyrighted material

0.0083 –0.75 435.36 +247.0

These numbers are shown in *decimal notation*, having a whole number followed by a decimal point and the fractional part. It is possible to omit digits before the decimal point, or digits after the decimal point. That is,

215. .95 –.71 +.5

are all valid real numbers.

A real number may also be expressed in *exponential* (or *scientific) notation*. For example, the value 215.65 may be written as 2.1565e2 in exponential notation. e2 means multiply by 10^2. The general form is:

mantissa e exponent

The *mantissa* is either a real number expressed in *decimal notation* or an integer. The *exponent* is an integer number with an optional *plus* or *minus sign*. The letter **e** separating the mantissa and the exponent can be written in either lowercase or uppercase. Since the exponent causes the decimal point to "float", this notation is said to represent a real number in *floating point form*. Examples of legal floating-point constants are:

0.65e4 12e-2 1.5e+5 3.18E3 -1.2E-1

Embedded white space is not allowed.

Exponential notation is useful for representing numbers that are either very large or very small in magnitude. For example, 7500000000 may be written as 7.5E9 or 75E8. Similarly, -0.000000368 is equivalent to –3.68E-7.

Floating-point constants are normally represented as double-precision quantities. However, the suffixes f or F may be used to force single-precision and l or L to extend double precision further.

Some examples of valid and invalid numeric constants are given in Table 2.4.

Table 2.4 *Examples of Numeric Constants*

Constant	*Valid ?*	*Remarks*
698354L	Yes	Represents long integer
25,000	No	Comma is not allowed
+5.0E3	Yes	(ANSI C supports unary plus)
3.5e-5	Yes	
7.1e 4	No	No white space is permitted
-4.5e-2	Yes	
1.5E+2.5	No	Exponent must be an integer
$255	No	$ symbol is not permitted
0X7B	Yes	Hexadecimal integer

Single Character Constants

A single character constant (or simply character constant) contains a single character enclosed within a pair of *single* quote marks. Example of character constants are:

'5' 'X' ';' ' '

Copyrighted material

Note that the character constant '5' is not the same as the *number 5*. The last constant is a blank space.

Character constants have integer values known as ASCII values. For example, the statement

```
printf("%d", 'a');
```

would print the number 97, the ASCII value of the letter a. Similarly, the statement

```
printf("%c", '97');
```

would output the letter 'a'. ASCII values for all characters are given in Appendix II.

Since each character constant represents an integer value, it is also possible to perform arithmetic operations on character constants. They are discussed in Chapter 8.

String Constants

A string constant is a sequence of characters enclosed in *double* quotes. The characters may be letters, numbers, special characters and blank space. Examples are:

"Hello!" "1987" "WELL DONE" "?...!" "5+3" "X"

Remember that a character constant (e.g., 'X') is not equivalent to the single character string constant (e.g., "X"). Further, a single character string constant does not have an equivalent integer value while a character constant has an integer value. Character strings are often used in programs to build meaningful programs. Manipulation of character strings are considered in detail in Chapter 8.

Backslash Character Constants

C supports some special backslash character constants that are used in output functions. For example, the symbol '\n' stands for newline character. A list of such backslash character constants is given in Table 2.5. Note that each one of them represents one character, although they consist of two characters. These characters combinations are known as *escape sequences*.

Table 2.5 *Backslash Character Constants*

Constant	*Meaning*
'\a'	audible alert (bell)
'\b'	back space
'\f'	form feed
'\n'	new line
'\r'	carriage return
'\t'	horizontal tab
'\v'	vertical tab
'\''	single quote
'\"'	double quote
'\?'	question mark
'\\'	backslash
'\0'	null

Copyrighted material

2.6 VARIABLES

A *variable* is a data name that may be used to store a data value. Unlike constants that remain unchanged during the execution of a program, a variable may take different values at different times during execution. In Chapter 1, we used several variables. For instance, we used the variable **amount** in Sample Program 3 to store the value of money at the end of each year (after adding the interest earned during that year).

A variable name can be chosen by the programmer in a meaningful way so as to reflect its function or nature in the program. Some examples of such names are:

Average
height
Total
Counter_1
class_strength

As mentioned earlier, variable names may consist of letters, digits, and the underscore(_) character, subject to the following conditions:

1. They must begin with a letter. Some systems permit underscore as the first character.
2. ANSI standard recognizes a length of 31 characters. However, length should not be normally more than eight characters, since only the first eight characters are treated as significant by many compilers. (In C99, at least 63 characters are significant.)
3. Uppercase and lowercase are significant. That is, the varible **Total** is not the same as **total** or **TOTAL**.
4. It should not be a keyword.
5. White space is not allowed.

Some examples of valid variable names are:

John	Value	T_raise
Delhi	x1	ph_value
mark	sum1	distance

Invalid examples include:

123	(area)
%	25th

Further examples of variable names and their correctness are given in Table 2.6.

Table 2.6 *Examples of Variable Names*

Variable name	*Valid?*	*Remark*
First_tag	Valid	
char	Not valid	char is a keyword
Price$	Not valid	Dollar sign is illegal
group one	Not valid	Blank space is not permitted
average_number	Valid	First eight characters are significant
int_type	Valid	Keyword may be part of a name

Copyrighted material

If only the first eight characters are recognized by a compiler, then the two names

average_height
average_weight

mean the same thing to the computer. Such names can be rewritten as

avg_height and avg_weight

or

ht_average and wt_average

without changing their meanings.

2.7 DATA TYPES

C language is rich in its *data types*. Storage representations and machine instructions to handle constants differ from machine to machine. The variety of data types available allow the programmer to select the type appropriate to the needs of the application as well as the machine.

ANSI C supports three classes of data types:

1. Primary (or fundamental) data types
2. Derived data types
3. User-defined data types

The primary data types and their extensions are discussed in this section. The user-defined data types are defined in the next section while the derived data types such as arrays, functions, structures and pointers are discussed as and when they are encountered.

All C compilers support five fundamental data types, namely integer (**int**), character (**char**), floating point (**float**), double-precision floating point (**double**) and **void.** Many of them also offer extended data types such as **long int** and **long double**. Various data types and the terminology used to describe them are given in Fig. 2.4. The range of the basic four types are given in Table 2.7. We discuss briefly each one of them in this section.

NOTE: C99 adds three more data types, namely **_Bool**, **_Complex**, and **_Imaginary**. See the Appendix "C99 Features".

Copyrighted material

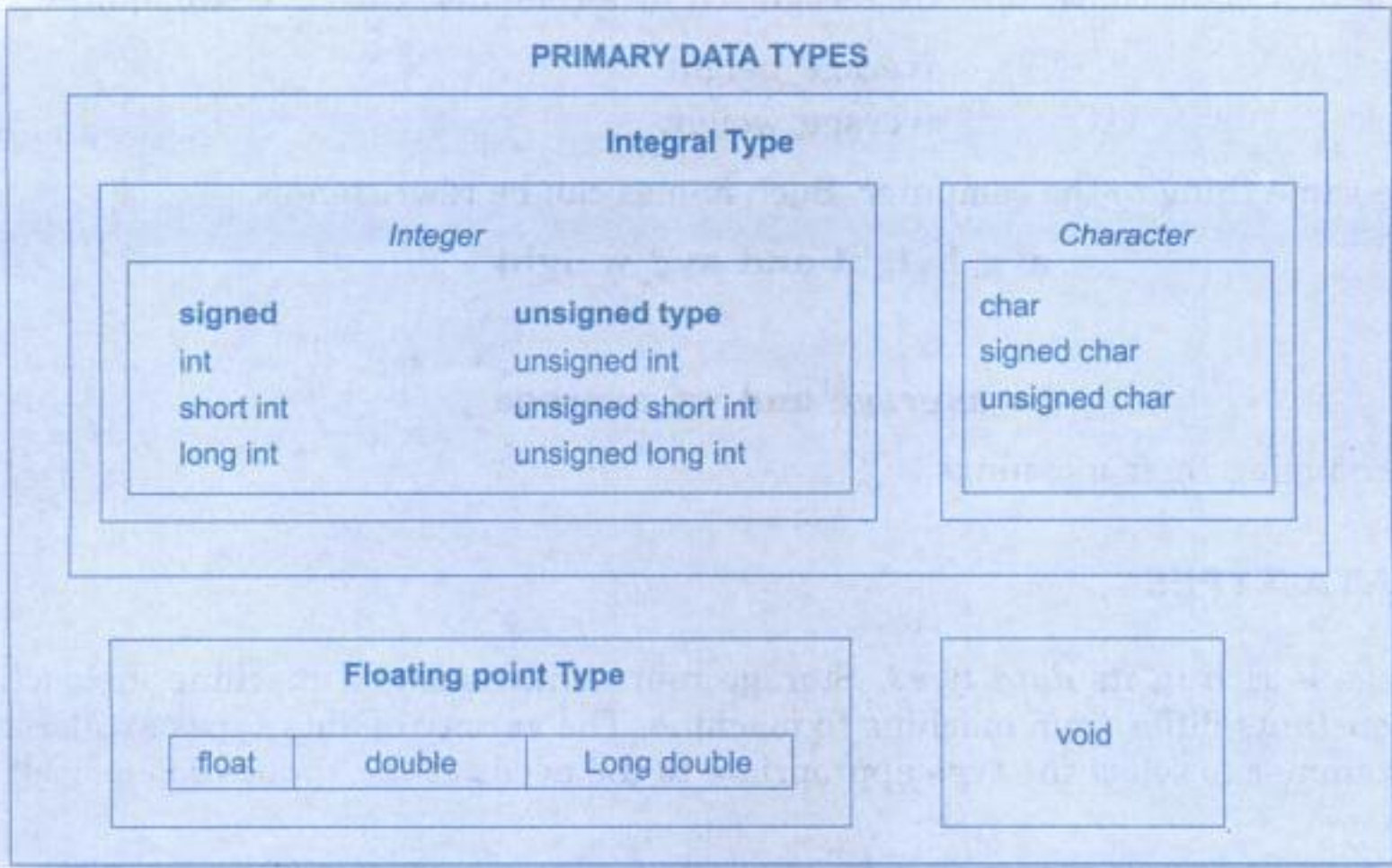


Fig. 2.4 *Primary data types in C*

Table 2.7 *Size and Range of Basic Data Types on 16-bit Machines*

Data type	*Range of values*
char	–128 to 127
int	–32,768 to 32,767
float	3.4e–38 to 3.4e+e38
double	1.7e–308 to 1.7e+308

Integer Types

Integers are whole numbers with a range of values supported by a particular machine. Generally, integers occupy one word of storage, and since the word sizes of machines vary (typically, 16 or 32 bits) the size of an integer that can be stored depends on the computer. If we use a 16 bit word length, the size of the integer value is limited to the range –32768 to +32767 (that is, -2^{15} to $+2^{15}-1$). A signed integer uses one bit for sign and 15 bits for the magnitude of the number. Similarly, a 32 bit word length can store an integer ranging from -2,147,483,648 to 2,147,483,647.

In order to provide some control over the range of numbers and storage space, C has three classes of integer storage, namely **short int, int,** and **long int,** in both **signed** and **unsigned** forms. ANSI C defines these types so that they can be organized from the smallest to the largest, as shown in Fig. 2.5. For example, **short int** represents fairly small integer values and requires half the amount of storage as a regular **int** number uses. Unlike signed

Copyrighted material

integers, unsigned integers use all the bits for the magnitude of the number and are always positive. Therefore, for a 16 bit machine, the range of unsigned integer numbers will be from 0 to 65,535.

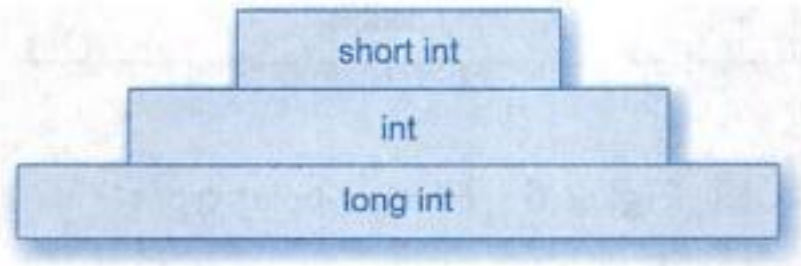


Fig. 2.5 *Integer types*

We declare **long** and **unsigned** integers to increase the range of values. The use of qualifier **signed** on integers is optional because the default declaration assumes a signed number. Table 2.8 shows all the allowed combinations of basic types and qualifiers and their size and range on a 16-bit machine.

NOTE: C99 allows **long long** integer types. See the Appendix "C99 Features".

Table 2.8 *Size and Range of Data Types on a 16-bit Machine*

Type	*Size (bits)*	*Range*
char or signed char	8	–128 to 127
unsigned char	8	0 to 255
int or signed int	16	–32,768 to 32,767
unsigned int	16	0 to 65535
short int or signed short int	8	–128 to 127
unsigned short int	8	0 to 255
long int or signed long int	32	–2,147,483,648 to 2,147,483,647
unsigned long int	32	0 to 4,294,967,295
float	32	3.4E – 38 to 3.4E + 38
double	64	1.7E – 308 to 1.7E + 308
long double	80	3.4E – 4932 to 1.1E + 4932

Floating Point Types

Floating point (or real) numbers are stored in 32 bits (on all 16 bit and 32 bit machines), with 6 digits of precision. Floating point numbers are defined in C by the keyword **float.** When the accuracy provided by a **float** number is not sufficient, the type **double** can be used to define the number. A **double** data type number uses 64 bits giving a precision of 14 digits. These are known as *double precision* numbers. Remember that double type represents the same data type that **float** represents, but with a greater precision. To extend the precision further, we may use **long double** which uses 80 bits. The relationship among floating types is illustrated in Fig. 2.6.

Copyrighted material

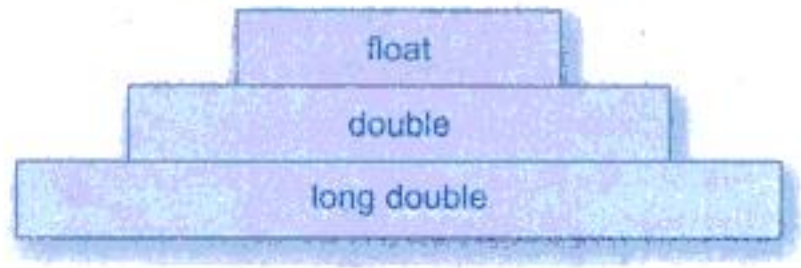


Fig. 2.6 *Floating-point types*

Void Types

The **void** type has no values. This is usually used to specify the type of functions. The type of a function is said to be **void** when it does not return any value to the calling function. It can also play the role of a generic type, meaning that it can represent any of the other standard types.

Character Types

A single character can be defined as a **character(char)** type data. Characters are usually stored in 8 bits (one byte) of internal storage. The qualifier **signed** or **unsigned** may be explicitly applied to char. While **unsigned chars** have values between 0 and 255, **signed chars** have values from –128 to 127.

2.8 DECLARATION OF VARIABLES

After designing suitable variable names, we must declare them to the compiler. Declaration does two things:

1. It tells the compiler what the variable name is.
2. It specifies what type of data the variable will hold.

The declaration of variables must be done before they are used in the program.

Primary Type Declaration

A variable can be used to store a value of any data type. That is, the name has nothing to do with its type. The syntax for declaring a variable is as follows:

data-type v1,v2,....vn ;

v1, v2, ...vn are the names of variables. Variables are separated by commas. A declaration statement must end with a semicolon. For example, valid declarations are:

```
int count;
int number, total;
double ratio;
```

int and **double** are the keywords to represent integer type and real type data values respectively. Table 2.9 shows various data types and their keyword equivalents.

Copyrighted material

Table 2.9 *Data Types and Their Keywords*

Data type	*Keyword equivalent*
Character	char
Unsigned character	unsigned char
Signed character	signed char
Signed integer	signed int (or int)
Signed short integer	signed short int (or short int or short)
Signed long integer	signed long int (or long int or long)
Unsigned integer	unsigned int (or unsigned)
Unsigned short integer	unsigned short int (or unsigned short)
Unsigned long integer	unsigned long int (or unsigned long)
Floating point	float
Double-precision floating point	double
Extended double-precision floating point	long double

The program segment given in Fig. 2.7 illustrates declaration of variables. **main()** is the beginning of the program. The opening brace { signals the execution of the program. Declaration of variables is usually done immediately after the opening brace of the program. The variables can also be declared outside (either before or after) the **main** function. The importance of place of declaration will be dealt in detail later while discussing functions.

Note: C99 permits declaration of variables at any point within a function or block, prior to their use.

```
main() /*.........Program Name........................ */
{
  /*................Declaration.......................*/
    float       x, y;
    int         code;
    short int   count;
    long int    amount;
    double      deviation;
    unsigned    n;
    char        c;
  /*...............Computation....................... */
    . . . .
    . . . .
    . . . .
} /*.............Program ends........................*/
```

Fig. 2.7 *Declaration of variables*

Copyrighted material

When an adjective (qualifier) **short, long, or unsigned** is used without a basic data type specifier, C compilers treat the data type as an **int.** If we want to declare a character variable as unsigned, then we must do so using both the terms like **unsigned char**.

Default values of Constants

Integer constants, by default, represent **int** type data. We can override this default by specifying unsigned or long after the number (by appending U or L) as shown below:

Literal	Type	Value
+111	int	111
–222	int	–222
45678U	unsigned int	45,678
–56789L	long int	–56,789
987654UL	unsigned long int	9,87,654

Similarly, floating point constants, by default represent **double** type data. If we want the resulting data type to be **float** or **long double**, we must append the letter f or F to the number for **float** and letter l or L for **long double** as shown below:

Literal	Type	Value
0.	double	0.0
.0	double	0.0
12.0	double	12.0
1.234	double	1.234
–1.2f	float	–1.2
1.23456789L	long double	1.23456789

User-Defined Type Declaration

C supports a feature known as "type definition" that allows users to define an identifier that would represent an existing data type. The user-defined data type identifier can later be used to declare variables . It takes the general form:

typedef ***type*** **identifier;**

Where *type* refers to an existing data type and "identifier" refers to the "new" name given to the data type. The existing data type may belong to any class of type, including the user-defined ones. Remember that the new type is 'new' only in name, but not the data type. **typedef** cannot create a new type. Some examples of type definition are:

```
typedef int units;
typedef float marks;
```

Here, **units** symbolizes **int** and **marks** symbolizes **float.** They can be later used to declare variables as follows:

```
units batch1, batch2;
marks name1[50], name2[50];
```

Copyrighted material

batch1 and batch2 are inclared as **int** variable and name1[50] and name2[50] are declared as 50 element floating point array variables. The main advantage of **typedef** is that we can create meaningful data type names for increasing the readability of the program.

Another user-defined data type is enumerated data type provided by ANSI standard. It is defined as follows:

enum identifier {*value1, value2, ... valuen*};

The "identifier" is a user-defined enumerated data type which can be used to declare variables that can have one of the values enclosed within the braces (known as *enumeration constants*). After this definition, we can declare variables to be of this 'new' type as below:

```
enum identifier v1, v2, ... vn;
```

The enumerated variables v1, v2, ... vn can only have one of the values *value1, value2, ... valuen*. The assignments of the following types are valid:

```
v1 = value3;
v5 = value1;
```

An example:

```
enum day {Monday,Tuesday, ... Sunday};
enum day week_st, week_end;
week_st = Monday;
week_end = Friday;
if(week_st == Tuesday)
week_end = Saturday;
```

The compiler automatically assigns integer digits beginning with 0 to all the enumeration constants. That is, the enumeration constant value1 is assigned 0, value2 is assigned 1, and so on. However, the automatic assignments can be overridden by assigning values explicitly to the enumeration constants. For example:

```
enum day {Monday = 1, Tuesday, ... Sunday};
```

Here, the constant Monday is assigned the value of 1. The remaining constants are assigned values that increase successively by 1.

The definition and declaration of enumerated variables can be combined in one statement. Example:

```
enum day {Monday, ... Sunday} week_st, week_end;
```

2.9 DECLARATION OF STORAGE CLASS

Variables in C can have not only *data type* but also *storage class* that provides information about their location and visibility. The storage class decides the portion of the program within which the variables are recognized. Consider the following example:

```
  /* Example of storage classes */
int m;
main()
{
  int i;
  float balance;
  ....
```

Copyrighted material

```
    ....
    function1();
}
function1()
{
    int i;
    float sum;
    ....
    ....
}
```

The variable **m** which has been declared before the **main** is called *global* variable. It can be used in all the functions in the program. It need not be declared in other functions. A global variable is also known as an *external* variable.

The variables **i**, **balance** and **sum** are called *local* variables because they are declared inside a function. Local variables are visible and meaningful only inside the functions in which they are declared. They are not known to other functions. Note that the variable **i** has been declared in both the functions. Any change in the value of **i** in one function does not affect its value in the other.

C provides a variety of storage class specifiers that can be used to declare explicitly the scope and lifetime of variables. The concepts of scope and lifetime are important only in multifunction and multiple file programs and therefore the storage classes are considered in detail later when functions are discussed. For now, remember that there are four storage class specifiers (**auto, register, static**, and **extern**) whose meanings are given in Table 2.10.

The storage class is another qualifier (like **long** or **unsigned**) that can be added to a variable declaration as shown below:

```
auto int count;
register char ch;
static int x;
extern long total;
```

Static and external (**extern**) variables are automatically initialized to zero. Automatic (**auto**) variables contain undefined values (known as 'garbage') unless they are initialized explicitly.

Table 2.10 *Storage Classes and Their Meaning*

Storage class	*Meaning*
auto	Local variable known only to the function in which it is declared. *Default is auto.*
static	Local variable which exists and retains its value even after the control is transferred to the calling function.
extern	Global variable known to all functions in the file.
register	Local variable which is stored in the register.

2.10 ASSIGNING VALUES TO VARIABLES

Variables are created for use in program statements such as,

Copyrighted material

```
value = amount + inrate * amount;
while (year <= PERIOD)
{
   ....
   ....
   year = year + 1;
}
```

In the first statement, the numeric value stored in the variable **inrate** is multiplied by the value stored in **amount** and the product is added to **amount**. The result is stored in the variable value. This process is possible only if the variables **amount** and inrate have already been given values. The variable **value** is called the *target variable*. While all the variables are declared for their type, the variables that are used in expressions (on the right side of equal (=) sign of a computational statement) *must* be assigned values before they are encountered in the program. Similarly, the variable **year** and the symbolic constant **PERIOD** in the **while** statement must be assigned values before this statement is encountered.

Assignment Statement

Values can be assigned to variables using the assignment operator = as follows:

variable_name = constant;

We have already used such statements in Chapter 1. Further examples are:

```
initial_value   = 0;
final_value     = 100;
balance         = 75.84;
yes             = 'x';
```

C permits multiple assignments in one line. For example

```
initial_value = 0; final_value = 100;
```

are valid statements.

An assignment statement implies that the value of the variable on the left of the 'equal sign' is set equal to the value of the quantity (or the expression) on the right. The statement

```
year = year + 1;
```

means that the 'new value' of **year** is equal to the 'old value' of **year** plus 1.

During assignment operation, C converts the type of value on the right-hand side to the type on the left. This may involve truncation when real value is converted to an integer.

It is also possible to assign a value to a variable at the time the variable is declared. This takes the following form:

data-type variable_name = constant;

Some examples are:

```
int final_value  = 100;
char yes         = 'x';
double balance   = 75.84;
```

Copyrighted material

The process of giving initial values to variables is called *initialization*. C permits the *initialization* of more than one variables in one statement using multiple assignment operators. For example the statements

```
p = q = s = 0;
x = y = z = MAX;
```

are valid. The first statement initializes the variables **p, q,** and **s** to zero while the second initializes **x, y,** and **z** with **MAX.** Note that **MAX** is a symbolic constant defined at the beginning.

Remember that external and static variables are initialized to zero by *default*. Automatic variables that are not initialized explicitly will contain garbage.

Example 2.2 Program in Fig. 2.8 shows typical declarations, assignments and values stored in various types of variables.

The variables **x** and **p** have been declared as floating-point variables. Note that the way the value of 1.234567890000 that we assigned to **x** is displayed under different output formats. The value of **x** is displayed as 1.234567880630 under %.12lf format, while the actual value assigned is 1.234567890000. This is because the variable **x** has been declared as a **float** that can store values only up to six decimal places.

The variable **m** that has been declared as **int** is not able to store the value 54321 correctly. Instead, it contains some garbage. Since this program was run on a 16-bit machine, the maximum value that an **int** variable can store is only 32767. However, the variable **k** (declared as **unsigned**) has stored the value 54321 correctly. Similarly, the **long int** variable **n** has stored the value 1234567890 correctly.

The value 9.87654321 assigned to **y** declared as double has been stored correctly but the value is printed as 9.876543 under %lf format. Note that unless specified otherwise, the **printf** function will always display a **float** or **double** value to six decimal places. We will discuss later the output formats for displaying numbers.

Program

```
main()
{
/*..........DECLARATIONS..........................*/
    float    x,p ;
    double   y,q ;
    unsigned k ;
/*..........DECLARATIONS AND ASSIGNMENTS............*/
    int      m = 54321 ;
    long int n = 1234567890 ;
/*..........ASSIGNMENTS............................*/
    x = 1.234567890000 ;
    y = 9.87654321 ;
    k = 54321 ;
    p = q = 1.0 ;
/*..........PRINTING...............................*/
```

Copyrighted material

```
        printf("m = %d\n", m) ;
        printf("n = %ld\n", n) ;
        printf("x = %.12lf\n", x) ;
        printf("x = %f\n", x) ;
        printf("y = %.12lf\n",y) ;
        printf("y = %lf\n", y) ;
        printf("k = %u p = %f q = %.12lf\n", k, p, q) ;
    }
```

Output

```
m = -11215
n = 1234567890
x = 1.234567880630
x = 1.234568
y = 9.876543210000
y = 9.876543
k = 54321 p = 1.000000 q = 1.000000000000
```

Fig. 2.8 *Examples of assignments*

Reading Data from Keyboard

Another way of giving values to variables is to input data through keyboard using the **scanf** function. It is a general input function available in C and is very similar in concept to the **printf** function. It works much like an INPUT statement in BASIC. The general format of **scanf** is as follows:

scanf("control string", &variable1,&variable2,....);

The control string contains the format of data being received. The ampersand symbol **&** before each variable name is an operator that specifies the variable name's *address*. We must always use this operator, otherwise unexpected results may occur. Let us look at an example:

```
scanf("%d", &number);
```

When this statement is encountered by the computer, the execution stops and waits for the value of the variable **number** to be typed in. Since the control string "%d" specifies that an integer value is to be read from the terminal, we have to type in the value in integer form. Once the number is typed in and the 'Return' Key is pressed, the computer then proceeds to the next statement. Thus, the use of **scanf** provides an interactive feature and makes the program 'user friendly'. The value is assigned to the variable **number**.

Example 2.3 The program in Fig. 2.9 illustrates the use of **scanf** function.

The first executable statement in the program is a **printf,** requesting the user to enter an integer number. This is known as "prompt message" and appears on the screen like

Enter an integer number

As soon as the user types in an integer number, the computer proceeds to compare the

Copyrighted material

value with 100. If the value typed in is less than 100, then a message

Your number is smaller than 100

is printed on the screen. Otherwise, the message

Your number contains more than two digits

is printed. Outputs of the program run for two different inputs are also shown in Fig. 2.9.

Program

```
main()
{
     int number;

     printf("Enter an integer number\n");
     scanf ("%d", &number);

     if ( number < 100 )
       printf("Your number is smaller than 100\n\n");
     else
       printf("Your number contains more than two digits\n");
}
```

Output

```
Enter an integer number
54
Your number is smaller than 100
Enter an integer number
108
Your number contains more than two digits
```

Fig. 2.9 *Use of **scanf** function for interactive computing*

Some compilers permit the use of the 'prompt message' as a part of the control string in **scanf**, like

```
scanf("Enter a number %d",&number);
```

We discuss more about **scanf** in Chapter 4.

In Fig. 2.9 we have used a decision statement **if...else** to decide whether the number is less than 100. Decision statements are discussed in depth in Chapter 5.

Example 2.4 Sample program 3 discussed in Chapter 1 can be converted into a more flexible interactive program using **scanf** as shown in Fig. 2.10.

In this case, computer requests the user to input the values of the amount to be invested, interest rate and period of investment by printing a prompt message

Input amount, interest rate, and period

Copyrighted material

and then waits for input values. As soon as we finish entering the three values corresponding to the

Program

```
main()
{
    int year, period ;
    float amount, inrate, value ;

    printf("Input amount, interest rate, and period\n\n") ;
    scanf ("%f %f %d", &amount, &inrate, &period) ;
    printf("\n") ;
    year = 1 ;

    while( year <= period )
    {
        value = amount + inrate * amount ;
        printf("%2d Rs %8.2f\n", year, value) ;
        amount = value ;
        year = year + 1 ;
    }
}
```

Output

```
Input amount, interest rate, and period

  10000 0.14 5

    1 Rs 11400.00
    2 Rs 12996.00
    3 Rs 14815.44
    4 Rs 16889.60
    5 Rs 19254.15
Input amount, interest rate, and period

  20000 0.12 7

    1 Rs 22400.00
    2 Rs 25088.00
    3 Rs 28098.56
    4 Rs 31470.39
    5 Rs 35246.84
    6 Rs 39476.46
    7 Rs 44213.63
```

Fig. 2.10 *Interactive investment program*

Copyrighted material

three variables **amount, inrate,** and **period,** the computer begins to calculate the amount at the end of each year, up to 'period' and produces output as shown in Fig. 2.10.

Note that the **scanf** function contains three variables. In such cases, care should be exercised to see that the values entered match the *order* and *type* of the variables in the list. Any mismatch might lead to unexpected results. The compiler may not detect such errors.

2.11 DEFINING SYMBOLIC CONSTANTS

We often use certain unique constants in a program. These constants may appear repeatedly in a number of places in the program. One example of such a constant is 3.142, representing the value of the mathematical constant **"pi".** Another example is the total number of students whose mark-sheets are analysed by a 'test analysis program'. The number of students, say 50, may be used for calculating the class total, class average, standard deviation, etc. We face two problems in the subsequent use of such programs. These are

1. problem in modification of the program and
2. problem in understanding the program.

Modifiability

We may like to change the value of "pi" from 3.142 to 3.14159 to improve the accuracy of calculations or the number 50 to 100 to process the test results of another class. In both the cases, we will have to search throughout the program and explicitly change the value of the constant wherever it has been used. If any value is left unchanged, the program may produce disastrous outputs.

Understandability

When a numeric value appears in a program, its use is not always clear, especially when the same value means different things in different places. For example, the number 50 may mean the number of students at one place and the 'pass marks' at another place of the same program. We may forget what a certain number meant, when we read the program some days later.

Assignment of such constants to a *symbolic name* frees us from these problems. For example, we may use the name **STRENGTH** to define the number of students and **PASS_MARK** to define the pass marks required in a subject. Constant values are assigned to these names at the beginning of the program. Subsequent use of the names **STRENGTH** and **PASS_MARK** in the program has the effect of causing their defined values to be automatically substituted at the appropriate points. A constant is defined as follows:

#define symbolic-name value of constant

Valid examples of constant definitions are:

```
#define STRENGTH 100
#define PASS_MARK 50
#define MAX 200
#define PI 3.14159
```

Symbolic names are sometimes called *constant identifiers.* Since the symbolic names are constants (not variables), they do not appear in declarations. The following rules apply to a **#define** statement which define a symbolic constant:

Copyrighted material

1. Symbolic names have the same form as variable names. (Symbolic names are written in CAPITALS to visually distinguish them from the normal variable names, which are written in lowercase letters. This is only a convention, not a rule.)
2. No blank space between the pound sign **'#'** and the word **define** is permitted.
3. **'#'** must be the first character in the line.
4. A blank space is required between **#define** and *symbolic name* and between the ***symbolic name*** and the *constant*.
5. **#define** statements must not end with a semicolon.
6. After definition, the *symbolic name* should not be assigned any other value within the program by using an assignment statement. For example, STRENGTH = 200; is illegal.
7. Symbolic names are NOT declared for data types. Its data type depends on the type of constant.
8. **#define** statements may appear *anywhere* in the program but before it is referenced in the program (the usual practice is to place them in the beginning of the program).

#define statement is a *preprocessor* compiler directive and is much more powerful than what has been mentioned here. More advanced types of definitions will be discussed later. Table 2.11 illustrates some invalid statements of **#define.**

Table 2.11 *Examples of Invalid #define Statements*

Statement	*Validity*	*Remark*
#define X = 2.5	Invalid	'=' sign is not allowed
# define MAX 10	Invalid	No white space between # and define
#define N 25;	Invalid	No semicolon at the end
#define N 5, M 10	Invalid	A statement can define only one name.
#Define ARRAY 11	Invalid	define should be in lowercase letters
#define PRICE$ 100	Invalid	$ symbol is not permitted in name

2.12 DECLARING A VARIABLE AS CONSTANT

We may like the value of certain variables to remain constant during the execution of a program. We can achieve this by declaring the variable with the qualifier **const** at the time of initialization. Example:

```
const int class_size = 40;
```

const is a new data type qualifier defined by ANSI standard. This tells the compiler that the value of the **int** variable **class_size** must not be modified by the program. However, it can be used on the right_hand side of an assignment statement like any other variable.

2.13 DECLARING A VARIABLE AS VOLATILE

ANSI standard defines another qualifier **volatile** that could be used to tell explicitly the compiler that a variable's value may be changed at any time by some external sources (from outside the program). For example:

```
volatile int date;
```

Copyrighted material

The value of **date** may be altered by some external factors even if it does not appear on the left-hand side of an assignment statement. When we declare a variable as **volatile**, the compiler will examine the value of the variable each time it is encountered to see whether any external alteration has changed the value.

Remember that the value of a variable declared as **volatile** can be modified by its own program as well. If we wish that the value must not be modified by the program while it may be altered by some other process, then we may declare the variable as both **const** and **volatile** as shown below:

```
volatile const int location = 100;
```

NOTE: C99 adds another qualifier called **restrict**. See the Appendix "C99 Features".

2.14 OVERFLOW AND UNDERFLOW OF DATA

Problem of data overflow occurs when the value of a variable is either too big or too small for the data type to hold. The largest value that a variable can hold also depends on the machine. Since floating-point values are rounded off to the number of significant digits allowed (or specified), an overflow normally results in the largest possible real value, whereas an underflow results in zero.

Integers are always exact within the limits of the range of the integral data types used. However, an overflow which is a serious problem may occur if the data type does not match the value of the constant. C does not provide any warning or indication of integer overflow. It simply gives incorrect results. (Overflow normally produces a negative number.) We should therefore exercise a greater care to define correct data types for handling the input/output values.

Just Remember

- Do not use the underscore as the first character of identifiers (or variable names) because many of the identifiers in the system library start with underscore.
- Use only 31 or less characters for identifiers. This helps ensure portability of programs.
- Do not use keywords or any system library names for identifiers.
- Use meaningful and intelligent variable names.
- Do not create variable names that differ only by one or two letters.
- Each variable used must be declared for its type at the beginning of the program or function.
- All variables must be initialized before they are used in the program.
- Integer constants, by default, assume **int** types. To make the numbers **long** or **unsigned**, we must append the letters L and U to them.
- Floating point constants default to **double**. To make them to denote **float** or **long double**, we must append the letters F or L to the numbers.
- Do not use lowercase l for long as it is usually confused with the number 1.

Copyrighted material

- Use single quote for character constants and double quotes for string constants.
- A character is stored as an integer. It is therefore possible to perform arithmetic operations on characters.
- Do not combine declarations with executable statements.
- A variable can be made constant either by using the preprocessor command **#define** at the beginning of the program or by declaring it with the qualifier **const** at the time of initialization.
- Do not use semicolon at the end of **#define** directive.
- The character # should be in the first column.
- Do not give any space between # and **define.**
- C does not provide any warning or indication of overflow. It simply gives incorrect results. Care should be exercised in defining correct data type.
- A variable defined before the main function is available to all the functions in the program.
- A variable defined inside a function is local to that function and not available to other functions.

Case Studies

1. Calculation of Average of Numbers

A program to calculate the average of a set of N numbers is given in Fig. 2.11.

Program

```
#define    N    10                    /* SYMBOLIC CONSTANT */
main()
{
   int    count ;                     /* DECLARATION OF */
   float  sum, average, number ;      /* VARIABLES */
   sum    = 0 ;                       /* INITIALIZATION */
   count  = 0 ;                       /* OF VARIABLES */
   while( count < N )
   {
         scanf("%f", &number) ;
         sum = sum + number ;
         count = count + 1 ;
   }
   average = sum/N ;
   printf("N = %d Sum = %f", N, sum);
   printf(" Average = %f", average);
}
```

Output

```
1
2.3
```

Copyrighted material

```
4.67
1.42
7
3.67
4.08
2.2
4.25
8.21
N = 10    Sum = 38.799999 Average = 3.880
```

Fig. 2.11 *Average of N numbers*

The variable **number** is declared as **float** and therefore it can take both integer and real numbers. Since the symbolic constant **N** is assigned the value of 10 using the **#define** statement, the program accepts ten values and calculates their sum using the **while** loop. The variable **count** counts the number of values and as soon as it becomes 11, the **while** loop is exited and then the average is calculated.

Notice that the actual value of sum is 38.8 but the value displayed is 38.799999. In fact, the actual value that is displayed is quite dependent on the computer system. Such an inaccuracy is due to the way the floating point numbers are internally represented inside the computer.

2. Temperature Conversion Problem

The program presented in Fig. 2.12 converts the given temperature in fahrenheit to celsius using the following conversion formula:

$$C = \frac{F - 32}{1.8}$$

Program

```
#define F_LOW     0        /* ---------------- */
#define F_MAX   250      /* SYMBOLIC CONSTANTS    */
#define STEP     25      /* --------------        */

main()
{
    typedef float REAL ;          /* TYPE DEFINITION */
    REAL fahrenheit, celsius ;    /* DECLARATION */

    fahrenheit = F_LOW ;          /* INITIALIZATION */
    printf("Fahrenheit Celsius\n\n") ;
    while( fahrenheit <= F_MAX )
    {
        celsius = ( fahrenheit - 32.0 ) / 1.8 ;
        printf(" %5.1f %7.2f\n", fahrenheit, celsius);
```

Copyrighted material

```
            fahrenheit = fahrenheit + STEP ;
        }
    }
Output
  Fahrenheit          Celsius

      0.0              -17.78
     25.0               -3.89
     50.0               10.00
     75.0               23.89
    100.0               37.78
    125.0               51.67
    150.0               65.56
    175.0               79.44
    200.0               93.33
    225.0              107.22
    250.0              121.11
```

Fig. 2.12 *Temperature conversion—fahrenheit-celsius*

The program prints a conversion table for reading temperature in celsius, given the fahrenheit values. The minimum and maximum values and step size are defined as symbolic constants. These values can be changed by redefining the **#define** statements. An user-defined data type name **REAL** is used to declare the variables **fahrenheit** and **celsius**.

The formation specifications %5.1f and %7.2 in the second **printf** statement produces two-column output as shown.

Review Questions

2.1 State whether the following statements are *true* or *false*.

(a) Any valid printable ASCII character can be used in an identifier.
(b) All variables must be given a type when they are declared.
(c) Declarations can appear anywhere in a program.
(d) ANSI C treats the variables **name** and **Name** to be same.
(e) The underscore can be used anywhere in an identifier.
(f) The keyword **void** is a data type in C.
(g) Floating point constants, by default, denote **float** type values.
(h) Like variables, constants have a type.
(i) Character constants are coded using double quotes.
(j) Initialization is the process of assigning a value to a variable at the time of declaration.
(k) All **static** variables are automatically initialized to zero.
(l) The **scanf** function can be used to read only one value at a time.

2.2 Fill in the blanks with appropriate words.

Copyrighted material

(a) The keyword _____ can be used to create a data type identifier.
(b) _____ is the largest value that an unsigned short int type variable can store.
(c) A global variable is also known as _____ variable.
(d) A variable can be made constant by declaring it with the qualifier _____ at the time of initialization.

2.3 What are trigraph characters? How are they useful?

2.4 Describe the four basic data types. How could we extend the range of values they represent?

2.5 What is an unsigned integer constant? What is the significance of declaring a constant unsigned?

2.6 Describe the characteristics and purpose of escape sequence characters.

2.7 What is a variable and what is meant by the "value" of a variable?

2.8 How do variables and symbolic names differ?

2.9 State the differences between the declaration of a variable and the definition of a symbolic name.

2.10 What is initialization? Why is it important?

2.11 What are the qualifiers that an **int** can have at a time?

2.12 A programmer would like to use the word DPR to declare all the double-precision floating point values in his program. How could he achieve this?

2.13 What are enumeration variables? How are they declared? What is the advantage of using them in a program?

2.14 Describe the purpose of the qualifiers **const** and **volatile**.

2.15 When dealing with very small or very large numbers, what steps would you take to improve the accuracy of the calculations?

2.16 Which of the following are invalid constants and why?

0.0001	5 × 1.5	99999
+100	75.45 E-2	"15.75"
–45.6	–1.79 e + 4	0.00001234

2.17 Which of the following are invalid variable names and why?

Minimum	First.name	n1+n2	&name
doubles	3rd_row	n$	Row1
float	Sum Total	Row Total	Column-total

2.18 Find errors, if any, in the following declaration statements.

```
Int x;
float letter,DIGIT;
double = p,q
exponent alpha,beta;
m,n,z: INTEGER
short char c;
long int m; count;
long float temp;
```

2.19 What would be the value of x after execution of the following statements?

```
int x, y = 10;
char z = 'a';
x = y + z;
```

2.20 Identify syntax errors in the following program. After corrections, what output would you expect when you execute it?

Copyrighted material

```
#define PI 3.14159
main()
{
    int R,C;            /* R-Radius of circle
    float perimeter;    /* Circumference of circle */
    float area;         /* Area of circle */
    C = PI
    R = 5;
    Perimeter = 2.0 * C *R;
    Area   = C*R*R;
    printf("%f", "%d",&perimeter,&area)
    }
```

Programming Exercises

2.1 Write a program to determine and print the sum of the following harmonic series for a given value of n:

$$1+ 1/2 +1/3 +....+ 1/n$$

The value of n should be given interactively through the terminal.

2.2 Write a program to read the price of an item in decimal form (like 15.95) and print the output in paise (like 1595 paise).

2.3 Write a program that prints the even numbers from 1 to 100.

2.4 Write a program that requests two float type numbers from the user and then divides the first number by the second and display the result along with the numbers.

2.5 The price of one kg of rice is Rs. 16.75 and one kg of sugar is Rs. 15. Write a program to get these values from the user and display the prices as follows:

*** LIST OF ITEMS ***

Item	Price
Rice	Rs 16.75
Sugar	Rs 15.00

2.6 Write program to count and print the number of negative and positive numbers in a given set of numbers. Test your program with a suitable set of numbers. Use **scanf** to read the numbers. Reading should be terminated when the value 0 is encountered.

2.7 Write a program to do the following:

(a) Declare x and y as integer variables and z as a short integer variable.
(b) Assign two 6 digit numbers to x and y
(c) Assign the sum of x and y to z
(d) Output the values of x, y and z

Comment on the output.

2.8 Write a program to read two floating point numbers using a **scanf** statement, assign their sum to an integer variable and then output the values of all the three variables.

2.9 Write a program to illustrate the use of **typedef** declaration in a program.

2.10 Write a program to illustrate the use of symbolic constants in a real-life application.

Copyrighted material

3

Operators and Expressions

3.1 INTRODUCTION

C supports a rich set of built-in operators. We have already used several of them, such as =, +, –, *, & and <. An *operator* is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in programs to manipulate data and variables. They usually form a part of the mathematical or logical *expressions*.

C operators can be classified into a number of categories. They include:

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Assignment operators
5. Increment and decrement operators
6. Conditional operators
7. Bitwise operators
8. Special operators

An expression is a sequence of operands and operators that reduces to a single value. For example,

$$10 + 15$$

is an expression whose value is 25. The value can be any type other than ***void***.

3.2 ARITHMETIC OPERATORS

C provides all the basic arithmetic operators. They are listed in Table 3.1. The operators +, –, *, and / all work the same way as they do in other languages. These can operate on any built-in data type allowed in C. The unary minus operator, in effect, multiplies its single operand by –1. Therefore, a number preceded by a minus sign changes its sign.

Copyrighted material

Table 3.1 *Arithmetic Operators*

Operator	*Meaning*
+	Addition or unary plus
–	Subtraction or unary minus
*	Multiplication
/	Division
%	Modulo division

Integer division truncates any fractional part. The modulo division operation produces the remainder of an integer division. Examples of use of arithmetic operators are:

```
a - b   a + b
a * b   a / b
a % b   -a * b
```

Here **a** and **b** are variables and are known as *operands*. The modulo division operator % cannot be used on floating point data. Note that C does not have an operator for *exponentiation*. Older versions of C does not support unary plus but ANSI C supports it.

Integer Arithmetic

When both the operands in a single arithmetic expression such as a+b are integers, the expression is called an *integer expression*, and the operation is called *integer arithmetic*. Integer arithmetic always yields an integer value. The largest integer value depends on the machine, as pointed out earlier. In the above examples, if **a** and **b** are integers, then for **a** = 14 and **b** = 4 we have the following results:

```
a - b  =  10
a + b  =  18
a * b  =  56
a / b  =  3 (decimal part truncated)
a % b  =  2 (remainder of division)
```

During integer division, if both the operands are of the same sign, the result is truncated towards zero. If one of them is negative, the direction of trunction is implementation dependent. That is,

$$6/7 = 0 \text{ and } -6/-7 = 0$$

but –6/7 may be zero or –1. (Machine dependent)

Similarly, during modulo division, the sign of the result is always the sign of the first operand (the dividend). That is

```
-14 % 3   =  -2
-14 % -3  =  -2
14 % -3   =  2
```

Example 3.1 The program in Fig. 3.1 shows the use of integer arithmetic to convert a given number of days into months and days.

Copyrighted material

```
Program
  main ()
  {
    int months, days ;

    printf("Enter days\n") ;
    scanf("%d", &days) ;

    months = days / 30 ;
    days = days % 30 ;
    printf("Months = %d Days = %d", months, days) ;
  }

Output
  Enter days
  265
  Months = 8 Days = 25
  Enter days
  364
  Months = 12 Days = 4
  Enter days
  45
  Months = 1 Days = 15
```

Fig. 3.1 *Illustration of integer arithmetic*

The variables months and days are declared as integers. Therefore, the statement

```
months = days/30;
```

truncates the decimal part and assigns the integer part to months. Similarly, the statement

```
days = days%30;
```

assigns the remainder part of the division to days. Thus the given number of days is converted into an equivalent number of months and days and the result is printed as shown in the output.

Real Arithmetic

An arithmetic operation involving only real operands is called *real arithmetic*. A real operand may assume values either in decimal or exponential notation. Since floating point values are rounded to the number of significant digits permissible, the final value is an approximation of the correct result. If **x, y,** and **z** are **floats**, then we will have:

$$x = 6.0/7.0 = 0.857143$$
$$y = 1.0/3.0 = 0.333333$$
$$z = -2.0/3.0 = -0.666667$$

The operator % cannot be used with real operands.

Copyrighted material

Mixed-mode Arithmetic

When one of the operands is real and the other is integer, the expression is called a *mixed-mode arithmetic* expression. If either operand is of the real type, then only the real operation is performed and the result is always a real number. Thus

$$15/10.0 = 1.5$$

whereas

$$15/10 = 1$$

More about mixed operations will be discussed later when we deal with the evaluation of expressions.

3.3 RELATIONAL OPERATORS

We often compare two quantities and depending on their relation, take certain decisions. For example, we may compare the age of two persons, or the price of two items, and so on. These comparisons can be done with the help of *relational operators*. We have already used the symbol '<', meaning 'less than'. An expression such as

$$a < b \text{ or } 1 < 20$$

containing a relational operator is termed as a *relational expression*. The value of a relational expression is either *one* or *zero*. It is *one* if the specified relation is *true* and *zero* if the relation is *false*. For example

$$10 < 20 \text{ is true}$$

but

$$20 < 10 \text{ is false}$$

C supports six relational operators in all. These operators and their meanings are shown in Table 3.2.

Table 3.2 *Relational Operators*

Operator	*Meaning*
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
==	is equal to
!=	is not equal to

A simple relational expression contains only one relational operator and takes the following form:

Copyrighted material

ae-1 relational operator ae-2

ae-1 and *ae-2* are arithmetic expressions, which may be simple constants, variables or combination of them. Given below are some examples of simple relational expressions and their values:

4.5 <= 10 TRUE

4.5 < –10 FALSE

–35 >= 0 FALSE

10 < 7+5 TRUE

a+b = c+d TRUE only if the sum of values of a and b is equal to the sum of values of c and d.

When arithmetic expressions are used on either side of a relational operator, the arithmetic expressions will be evaluated first and then the results compared. That is, arithmetic operators have a higher priority over relational operators.

Relational expressions are used in *decision statements* such as **if** and **while** to decide the course of action of a running program. We have already used the **while** statement in Chapter 1. Decision statements are discussed in detail in Chapters 5 and 6.

Relational Operator Complements

Among the six relational operators, each one is a complement of another operator.

>	is complement of	<=
<	is complement of	>=
==	is complement of	!=

We can simplify an expression involving the *not* and the *less than* operators using the complements as shown below:

Actual one	*Simplified one*
!(x<y)	x >= y
!(x>y)	x <= y
!(x!=y)	x == y
!(x<=y)	x > y
!(x>=y)	x < y
!(x == y)	x != y

Copyrighted material

3.4 LOGICAL OPERATORS

In addition to the relational operators, C has the following three *logical operators*.

&& meaning logical AND
|| meaning logical OR
! meaning logical NOT

The logical operators && and || are used when we want to test more than one condition and make decisions. An example is:

a > b && x == 10

An expression of this kind, which combines two or more relational expressions, is termed as a *logical expression* or a *compound relational expression*. Like the simple relational expressions, a logical expression also yields a value of *one* or *zero*, according to the truth table shown in Table 3.3. The logical expression given above is true only if **a > b** is *true* and **x** == 10 is *true*. If either (or both) of them are false, the expression is *false*.

Table 3.3 *Truth Table*

op-1	*op-2*	*Value of the expression*	
		op-1 && op-2	*op-1 || op-2*
Non-zero	Non-zero	1	1
Non-zero	0	0	1
0	Non-zero	0	1
0	0	0	0

Some examples of the usage of logical expressions are:

1. if (age > 55 && salary < 1000)
2. if (number < 0 || number > 100)

We shall see more of them when we discuss decision statements.

NOTE: Relative precedence of the relational and logical operators is as follows:

```
Highest      !
             > >=  < <=
             == !=
             &&
Lowest       ||
```

It is important to remember this when we use these operators in compound expressions.

3.5 ASSIGNMENT OPERATORS

Assignment operators are used to assign the result of an expression to a variable. We have seen the usual assignment operator, '='. In addition, C has a set of '*shorthand*' assignment operators of the form

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Example 3.2 Program of Fig. 3.2 prints a sequence of squares of numbers. Note the use of the shorthand operator *= .

The program attempts to print a sequence of squares of numbers starting from 2. The statement

```
a *= a;
```

which is identical to

```
a = a*a;
```

replaces the current value of **a** by its square. When the value of **a** becomes equal or greater than **N** (=100) the **while** is terminated. Note that the output contains only three values 2, 4 and 16.

Program

```
#define   N   100
#define   A   2
main()
{
        int a;
        a = A;
        while( a < N )
        {
            printf("%d\n", a);
            a *= a;
        }
}
```

Output

```
2
4
16
```

Fig. 3.2 *Use of shorthand operator* *=

3.6 INCREMENT AND DECREMENT OPERATORS

C allows two very useful operators not generally found in other languages. These are the increment and decrement operators:

++ and --

The operator ++ adds 1 to the operand, while -- subtracts 1. Both are unary operators and takes the following form:

Copyrighted material

```
++m; or m++;
--m; or m--;
```

```
++m; is equivalent to m = m+1; (or m += 1;)
--m; is equivalent to m = m-1; (or m -= 1;)
```

We use the increment and decrement statements in **for** and **while** loops extensively.

While ++m and m++ mean the same thing when they form statements independently, they behave differently when they are used in expressions on the right-hand side of an assignment statement. Consider the following:

```
m = 5;
y = ++m;
```

In this case, the value of y and m would be 6. Suppose, if we rewrite the above statements as

```
m = 5;
y = m++;
```

then, the value of y would be 5 and m would be 6. *A prefix operator first adds 1 to the operand and then the result is assigned to the variable on left. On the other hand, a postfix operator first assigns the value to the variable on left and then increments the operand.*

Similar is the case, when we use ++ (or – –) in subscripted variables. That is, the statement

```
a[i++] = 10;
```

is equivalent to

```
a[i] = 10;
i = i+1;
```

The increment and decrement operators can be used in complex statements. Example:

```
m = n++ -j+10;
```

Old value of n is used in evaluating the expression. n is incremented after the evaluation. Some compilers require a space on either side of n++ or ++n.

Rules for + + and – – Operators

- Increment and decrement operators are unary operators and they require variable as their operands.
- When postfix ++ (or – –) is used with a variable in an expression, the expression is evaluated first using the original value of the variable and then the variable is incremented (or decremented) by one.
- When prefix + + (or – –) is used in an expression, the variable is incremented (or decremented) first and then the expression is evaluated using the new value of the variable.
- The precedence and associatively of + + and – – operators are the same as those of unary + and unary –.

Copyrighted material

3.7 CONDITIONAL OPERATOR

A ternary operator pair "? :" is available in C to construct conditional expressions of the form

exp1 ? exp2 : exp3

where *exp1*, *exp2*, and *exp3* are expressions.

The operator ? : works as follows: *exp1* is evaluated first. If it is nonzero (true), then the expression *exp2* is evaluated and becomes the value of the expression. If *exp1* is false, *exp3* is evaluated and its value becomes the value of the expression. Note that only one of the expressions (either *exp2* or *exp3*) is evaluated. For example, consider the following statements.

```
a = 10;
b = 15;
x = (a > b) ? a : b;
```

In this example, x will be assigned the value of b. This can be achieved using the **if..else** statements as follows:

```
if (a > b)
   x = a;
else
   x = b;
```

3.8 BITWISE OPERATORS

C has a distinction of supporting special operators known as *bitwise operators* for manipulation of data at bit level. These operators are used for testing the bits, or shifting them right or left. Bitwise operators may not be applied to **float** or **double**. Table 3.5 lists the bitwise operators and their meanings. They are discussed in detail in Appendix I.

Table 3.5 *Bitwise Operators*

Operator	*Meaning*
&	bitwise AND
|	bitwise OR
^	bitwise exclusive OR
<<	shift left
>>	shift right

3.9 SPECIAL OPERATORS

C supports some special operators of interest such as comma operator, **sizeof** operator, pointer operators (**&** and *****) and member selection operators (. and –>). The comma and **sizeof** operators are discussed in this section while the pointer operators are discussed in

Copyrighted material

Chapter 11. Member selection operators which are used to select members of a structure are discussed in Chapters 10 and 11. ANSI committee has introduced two preprocessor operators known as "string-izing" and "token-pasting" operators (# and ##). They will be discussed in Chapter 14.

The Comma Operator

The comma operator can be used to link the related expressions together. A comma-linked list of expressions are evaluated *left to right* and the value of *right-most* expression is the value of the combined expression. For example, the statement

```
value = (x = 10, y = 5, x+y);
```

first assigns the value 10 to **x**, then assigns 5 to **y**, and finally assigns 15 (i.e. 10 + 5) to **value**. Since comma operator has the lowest precedence of all operators, the parentheses are necessary. Some applications of comma operator are:

In **for** loops:

```
for ( n = 1, m = 10, n <=m; n++, m++)
```

In **while** loops:

```
while (c = getchar( ), c != '10')
```

Exchanging values:

```
t = x, x = y, y = t;
```

The sizeof Operator

The **sizeof** is a compile time operator and, when used with an operand, it returns the number of bytes the operand occupies. The operand may be a variable, a constant or a data type qualifier.

Examples:

m = **sizeof** (sum);
n = **sizeof** (long int);
k = **sizeof** (235L);

The **sizeof** operator is normally used to determine the lengths of arrays and structures when their sizes are not known to the programmer. It is also used to allocate memory space dynamically to variables during execution of a program.

Example 3.3 In Fig. 3.3, the program employs different kinds of operators. The results of their evaluation are also shown for comparison.

Notice the way the increment operator **++** works when used in an expression. In the statement

```
c = ++a - b;
```

new value of **a** (= 16) is used thus giving the value 6 to c. That is, a is incremented by 1 before it is used in the expression. However, in the statement

```
d = b++ + a;
```

the old value of **b** (=10) is used in the expression. Here, b is incremented by 1 after it is used in the expression.

Copyrighted material

We can print the character % by placing it immediately after another % character in the control string. This is illustrated by the statement

```
printf("a%%b = %d\n", a%b);
```

The program also illustrates that the expression

```
c > d ? 1 : 0
```

assumes the value 0 when c is less than d and 1 when c is greater than d.

Program

```
main()
{
    int a, b, c, d;

    a = 15;
    b = 10;
    c = ++a - b;

    printf("a = %d b = %d c = %d\n",a, b, c);

    d = b++ +a;

    printf("a = %d b = %d d = %d\n",a, b, d);
    printf("a/b = %d\n", a/b);
    printf("a%%b = %d\n", a%b);
    printf("a *= b = %d\n", a*=b);
    printf("%d\n", (c>d) ? 1 : 0);
    printf("%d\n", (c<d) ? 1 : 0);
}
```

Output

```
a = 16 b = 10 c = 6
a = 16 b = 11 d = 26
a/b = 1
a%b = 5
a *= b = 176
0
1
```

Fig. 3.3 *Further illustration of arithmetic operators*

3.10 ARITHMETIC EXPRESSIONS

An arithmetic expression is a combination of variables, constants, and operators arranged as per the syntax of the language. We have used a number of simple expressions in the examples discussed so far. C can handle any complex mathematical expressions. Some of the examples of C expressions are shown in Table 3.6. Remember that C does not have an operator for exponentiation.

Copyrighted material

Table 3.6 *Expressions*

Algebraic expression	*C expression*
a x b - c	a * b - c
(m+n) (x+y)	(m+n) * (x+y)
$\left(\frac{ab}{c}\right)$	a * b/c
$3x^2+2x+1$	3 * x * x + 2 * x + 1
$\left(\frac{x}{y}\right)+c$	x/y+c

3.11 EVALUATION OF EXPRESSIONS

Expressions are evaluated using an assignment statement of the form:

variable = expression;

Variable is any valid C variable name. When the statement is encountered, the expression is evaluated first and the result then replaces the previous value of the variable on the left-hand side. All variables used in the expression must be assigned values before evaluation is attempted. Examples of evaluation statements are

```
x = a * b - c;
y = b / c * a;
z = a - b / c + d;
```

The blank space around an operator is optional and adds only to improve readability. When these statements are used in a program, the variables a, b, c, and d must be defined before they are used in the expressions.

Example 3.4 The program in Fig. 3.4 illustrates the use of variables in expressions and their evaluation.

Output of the program also illustrates the effect of presence of parentheses in expressions. This is discussed in the next section.

Program

```
main()
{
        float a, b, c, x, y, z;
```

Copyrighted material

```
        a = 9;
        b = 12;
        c = 3;

        x = a - b / 3 + c * 2 - 1;
        y = a - b / (3 + c) * (2 - 1);
        z = a - (b / (3 + c) * 2) - 1;

        printf("x = %f\n", x);
        printf("y = %f\n", y);
        printf("z = %f\n", z);
    }
```

Output

```
x = 10.000000
y = 7.000000
z = 4.000000
```

Fig. 3.4 *Illustrations of evaluation of expressions*

3.12 PRECEDENCE OF ARITHMETIC OPERATORS

An arithmetic expression without parentheses will be evaluated from *left to right* using the rules of precedence of operators. There are two distinct priority levels of arithmetic operators in C:

High priority * / %

Low priority + –

The basic evaluation procedure includes 'two' left-to-right passes through the expression. During the first pass, the high priority operators (if any) are applied as they are encountered. During the second pass, the low priority operators (if any) are applied as they are encountered. Consider the following evaluation statement that has been used in the program of Fig. 3.4.

x = a–b/3 + c*2–1

When a = 9, b = 12, and c = 3, the statement becomes

x = 9–12/3 + 3*2–1

and is evaluated as follows

First pass

Step1: x = 9–4+3*2–1

Step2: x = 9–4+6–1

Copyrighted material

Second pass

Step3: x = 5+6–1
Step4: x = 11–1
Step5: x = 10

These steps are illustrated in Fig. 3.5. The numbers inside parentheses refer to step numbers.

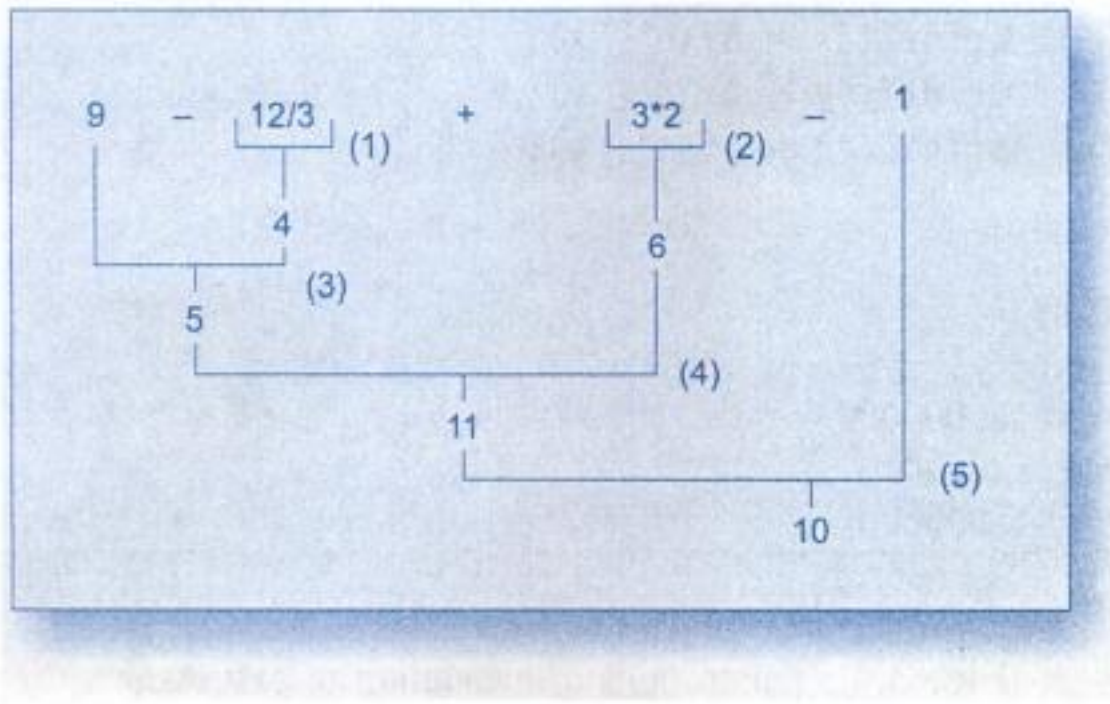


Fig. 3.5 *Illustration of hierarchy of operations*

However, the order of evaluation can be changed by introducing parentheses into an expression. Consider the same expression with parentheses as shown below:

9–12/(3+3)*(2–1)

Whenever parentheses are used, the expressions within parentheses assume highest priority. If two or more sets of parentheses appear one after another as shown above, the expression contained in the left-most set is evaluated first and the right-most in the last. Given below are the new steps.

First pass

Step1: 9-12/6 * (2-1)
Step2: 9-12/6 * 1

Second pass

Step3: 9-2 * 1
Step4: 9-2

Third pass

Step5: 7

This time, the procedure consists of three left-to-right passes. However, the number of evaluation steps remains the same as 5 (i.e equal to the number of arithmetic operators).

Copyrighted material

Parentheses may be nested, and in such cases, evaluation of the expression will proceed outward from the innermost set of parentheses. Just make sure that every opening parenthesis has a matching closing parenthesis. For example

$$9 - (12/(3+3) * 2) - 1 = 4$$

whereas

$$9 - ((12/3) + 3 * 2) - 1 = -2$$

While parentheses allow us to change the order of priority, we may also use them to improve understandability of the program. When in doubt, we can always add an extra pair just to make sure that the priority assumed is the one we require.

Rules for Evaluation of Expression

- First, parenthesized sub expression from left to right are evaluated.
- If parentheses are nested, the evaluation begins with the innermost sub-expression.
- The precedence rule is applied in determining the order of application of operators in evaluating sub-expressions
- The associativity rule is applied when two or more operators of the same precedence level appear in a sub-expression.
- Arithmetic expressions are evaluated from left to right using the rules of precedence.
- When parentheses are used, the expressions within parentheses assume highest priority.

3.13 SOME COMPUTATIONAL PROBLEMS

When expressions include real values, then it is important to take necessary precautions to guard against certain computational errors. We know that the computer gives approximate values for real numbers and the errors due to such approximations may lead to serious problems. For example, consider the following statements:

```
a = 1.0/3.0;
b = a * 3.0;
```

We know that (1.0/3.0) 3.0 is equal to 1. But there is no guarantee that the value of **b** computed in a program will equal 1.

Another problem is division by zero. On most computers, any attempt to divide a number by zero will result in abnormal termination of the program. In some cases such a division may produce meaningless results. Care should be taken to test the denominator that is likely to assume zero value and avoid any division by zero.

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The operator **(float)** converts the **female_number** to floating point for the purpose of evaluation of the expression. Then using the rule of automatic conversion, the division is performed in floating point mode, thus retaining the fractional part of result.

Note that in no way does the operator **(float)** affect the value of the variable **female number.** And also, the type of **female number** remains as **int** in the other parts of the program.

The process of such a local conversion is known as *explicit conversion* or *casting a value.* The general form of a cast is:

(type-name)*expression*

where *type-name* is one of the standard C data types. The expression may be a constant, variable or an expression. Some examples of casts and their actions are shown in Table 3.7.

Table 3.7 *Use of Casts*

Example	*Action*
x = **(int)** 7.5	7.5 is converted to integer by truncation.
a = **(int)** 21.3/(int)4.5	Evaluated as 21/4 and the result would be 5.
b = **(double)**sum/n	Division is done in floating point mode.
y = **(int)** (a+b)	The result of a+b is converted to integer.
z = **(int)**a+b	a is converted to integer and then added to b.
p = cos(**(double)**x)	Converts x to double before using it.

Casting can be used to round-off a given value. Consider the following statement:

x = **(int)** (y+0.5);

If **y** is 27.6, **y**+0.5 is 28.1 and on casting, the result becomes 28, the value that is assigned to **x**. Of course, the expression, being cast is not changed.

Example 3.6 Figure 3.8 shows a program using a cast to evaluate the equation

$$\text{sum} = \sum_{i=1}^{n}(1/i)$$

Program

```
main()
{
    float   sum ;
    int         n ;

    sum = 0 ;

    for( n = 1 ; n <= 10 ; ++n )
    {
        sum = sum + 1/(float)n ;
        printf("%2d %6.4f\n", n, sum) ;
    }
}
```

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

4. C99 has added **float** and **long double** versions of these fuctions.
5. C99 has added many more mathematical functions.
6. See the Appendix "C99 Features" for details.

As pointed out earlier in Chapter 1, to use any of these functions in a program, we should include the line:

```
# include <math.h>
```

in the beginning of the program.

Just Remember

- Use *decrement* and *increment* operators carefully. Understand the difference between **postfix** and **prefix** operations before using them.
- Add parentheses wherever you feel they would help to make the evaluation order clear.
- Be aware of side effects produced by some expressions.
- Avoid any attempt to divide by zero. It is normally undefined. It will either result in a fatal error or in incorrect results.
- Do not forget a semicolon at the end of an expression.
- Understand clearly the precedence of operators in an expression. Use parentheses, if necessary.
- Associativity is applied when more than one operator of the same precedence are used in an expression. Understand which operators associate from right to left and which associate from left to right.
- Do not use *increment* or *decrement* operators with any expression other than a *variable identifier*.
- It is illegal to apply modules operator % with anything other than integers.
- Do not use a variable in an expression before it has been assigned a value.
- Integer division always truncates the decimal part of the result. Use it carefully. Use casting where necessary.
- The result of an expression is converted to the type of the variable on the left of the assignment before assigning the value to it. Be careful about the loss of information during the conversion.
- All mathematical functions implement *double* type parameters and return *double* type values.
- It is an error if any space appears between the two symbols of the operators ==, !=, <= and >=.
- It is an error if the two symbols of the operators !=, <= and >= are reversed.
- Use spaces on either side of binary operator to improve the readability of the code.
- Do not use increment and decrement operators to floating point variables.
- Do not confuse the equality operator == with the assignment operator =.

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

3.11 Write a program to read a four digit integer and print the sum of its digits.
Hint: Use / and % operators.

3.12 Write a program to print the size of various data types in C.

3.13 Given three values, write a program to read three values from keyboard and print out the largest of them without using **if** statement.

3.14 Write a program to read two integer values m and n and to decide and print whether m is a multiple of n.

3.15 Write a program to read three values using **scanf** statement and print the following results:

(a) Sum of the values
(b) Average of the three values
(c) Largest of the three
(d) Smallest of the three

3.16 The cost of one type of mobile service is Rs. 250 plus Rs. 1.25 for each call made over and above 100 calls. Write a program to read customer codes and calls made and print the bill for each customer.

3.17 Write a program to print a table of **sin** and **cos** functions for the interval from 0 to 180 degrees in increments of 15 as shown below.

x (degrees)	*sin (x)*	*cos (x)*
0	.....	.....
15	.....	.....
...		
180	.....	.....

3.18 Write a program to compute the values of square-roots and squares of the numbers 0 to 100 in steps 10 and print the output in a tabular form as shown below.

Number	*Square-root*	*Square*
0	0	0
100	10	10000

3.19 Write a program that determines whether a given integer is odd or even and displays the number and description on the same line.

3.20 Write a program to illustrate the use of cast operator in a real life situation.

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Example 4.2 The program of Fig. 4.2 requests the user to enter a character and displays a message on the screen telling the user whether the character is an alphabet or digit, or any other special character.

This program receives a character from the keyboard and tests whether it is a letter or digit and prints out a message accordingly. These tests are done with the help of the following functions:

```
isalpha(character)
isdigit(character)
```

For example, **isalpha** assumes a value non-zero (TRUE) if the argument **character** contains an alphabet; otherwise it assumes 0 (FALSE). Similar is the case with the function **isdigit**.

Program:

```
#include <stdio.h>
#include <ctype.h>
main()
{
  char character;
  printf("Press any key\n");
  character = getchar();
  if (isalpha(character) > 0)/* Test for letter */
    printf("The character is a letter.");
  else
    if (isdigit (character) > 0)/* Test for digit */
      printf("The character is a digit.");
    else
      printf("The character is not alphanumeric.");
}
```

Output

```
Press any key
h
The character is a letter.
Press any key
5
The character is a digit.
Press any key
*
The character is not alphanumeric.
```

Fig. 4.2 *Program to test the* ***character*** *type*

C supports many other similar functions, which are given in Table 4.1. These character functions are contained in the file **ctype.h** and therefore the statement

```
#include <ctype.h>
```

must be included in the program.

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Example 4.4 Various input formatting options for reading integers are experimented in the program shown in Fig. 4.4.

Program

```
main()
{
   int a,b,c,x,y,z;
   int p,q,r;
   printf("Enter three integer numbers\n");
   scanf("%d %*d %d",&a,&b,&c);
   printf("%d %d %d \n\n",a,b,c);
   printf("Enter two 4-digit numbers\n");
   scanf("%2d %4d",&x,&y);
   printf("%d %d\n\n", x,y);
   printf("Enter two integers\n");
   scanf("%d %d", &a,&x);
   printf("%d %d \n\n",a,x);
   printf("Enter a nine digit number\n");
   scanf("%3d %4d %3d",&p,&q,&r);
   printf("%d %d %d \n\n",p,q,r);
   printf("Enter two three digit numbers\n");
   scanf("%d %d",&x,&y);
   printf("%d %d",x,y);
}
```

Output

```
   Enter three integer numbers
   1 2 3
   1 3 -3577
   Enter two 4-digit numbers
   6789 4321
   67 89
   Enter two integers
   44 66
   4321 44
Enter a nine-digit number
123456789
66 1234 567
Enter two three-digit numbers
123 456
89 123
```

Fig. 4.4 *Reading integers using* ***scanf***

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
        printf("Enter address\n");
        scanf("%[a-z]", address);
        printf("%-80s\n\n", address);
    }
```

Output

```
        Enter address
        new delhi 110002
        new delhi
```

Program-B

```
    main()
    {
        char address[80];
        printf("Enter address\n");
        scanf("%[^\n]", address);
        printf("%-80s", address);
    }
```

Output

```
        Enter address
        New Delhi 110 002
        New Delhi 110 002
```

Fig. 4.7 *Illustration of conversion specification%[] for strings*

Reading Blank Spaces

We have earlier seen that %s specifier cannot be used to read strings with blank spaces. But, this can be done with the help of %[ ] specification. Blank spaces may be included within the brackets, thus enabling the **scanf** to read strings with spaces. Remember that the lowercase and uppercase letters are distinct. See Fig. 4.7.

Reading Mixed Data Types

It is possible to use one **scanf** statement to input a data line containing mixed mode data. In such cases, care should be exercised to ensure that the input data items match the control specifications *in order* and *type*. When an attempt is made to read an item that does not match the type expected, the **scanf** function does not read any further and immediately returns the values read. The statement

```
scanf ("%d %c %f %s", &count, &code, &ratio, name);
```

will read the data

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The **printf** statement provides certain features that can be effectively exploited to control the alignment and spacing of print-outs on the terminals. The general form of **printf** statement is:

```
printf("control string", arg1, arg2, ....., argn);
```

Control string consists of three types of items:

1. Characters that will be printed on the screen as they appear.
2. Format specifications that define the output format for display of each item.
3. *Escape sequence* characters such as \n, \t, and \b.

The control string indicates how many arguments follow and what their types are. The arguments *arg1, arg2, ...,* *argn* are the variables whose values are formatted and printed according to the specifications of the control string. The arguments should match in number, order and type with the format specifications.

A simple format specification has the following form:

```
% w.p type-specifier
```

where *w* is an integer number that specifies the total number of columns for the output value and *p* is another integer number that specifies the number of digits to the right of the decimal point (of a real number) or the number of characters to be printed from a string. Both ***w*** and *p* are optional. Some examples of formatted **printf** statement are:

```
printf("Programming in C");
printf(" ");
printf("\n");
printf("%d", x);
printf("a = %f\n b = %f", a, b);
printf("sum = %d", 1234);
printf("\n\n");
```

printf never supplies a *newline* automatically and therefore multiple **printf** statements may be used to build one line of output. A *newline* can be introduced by the help of a newline character '\n' as shown in some of the examples above.

Output of Integer Numbers

The format specification for printing an integer number is:

```
% w d
```

where ***w*** specifies the minimum field width for the output. However, if a number is greater than the specified field width, it will be printed in full, overriding the minimum specification. ***d*** specifies that the value to be printed is an integer. The number is written *right-justified* in the given field width. Leading blanks will appear as necessary. The following examples illustrate the output of the number 9876 under different formats:

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Printing of a Single Character

A single character can be displayed in a desired position using the format:

%wc

The character will be displayed *right-justified* in the field of *w* columns. We can make the display *left-justified* by placing a minus sign before the integer w. The default value for w is 1.

Printing of Strings

The format specification for outputting strings is similar to that of real numbers. It is of the form

%w.ps

where *w* specifies the field width for display and *p* instructs that only the first p characters of the string are to be displayed. The display is *right-justified*.

The following examples show the effect of variety of specifications in printing a string "NEW DELHI 110001", containing 16 characters (including blanks).

Specification	Output																			
	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0
%s	N	E	W		D	E	L	H	I		1	1	0	0	0	1				
%20s					N	E	W		D	E	L	H	I		1	1	0	0	0	1
%20.10s											N	E	W		D	E	L	H	I	
%.5s	N	E	W		D															
%-20.10s	N	E	W		D	E	L	H	I											
%5s	N	E	W		D	E	L	H	I		1	1	0	0	0	1				

Example 4.11 Printing of characters and strings is illustrated in Fig. 4.11.

Program

```
main()
{
  char x = 'A';
  char name[20] = "ANIL KUMAR GUPTA";
  printf("OUTPUT OF CHARACTERS\n\n");
  printf("%c\n%3c\n%5c\n", x,x,x);
  printf("%3c\n%c\n", x,x);
```

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Code	*Quantity*	*Rate (Rs)*
F105	275	575.00
H220	107	99.95
I019	321	215.50
M315	89	725.00

It is required to prepare the inventory report table in the following format:

INVENTORY REPORT

Code	*Quantity*	*Rate*	*Value*
____	____	____	____
____	____	____	____
____	____	____	____
____	____	____	____
		Total Value:	_____

The value of each item is given by the product of quantity and rate.

Program: The program given in Fig. 4.12 reads the data from the terminal and generates the required output. The program uses subscripted variables which are discussed in Chapter 7.

```
Program
     #define ITEMS 4
     main()
     { /* BEGIN */
       int i, quantity[5];
       float rate[5], value, total_value;
       char code[5][5];
       /* READING VALUES */
       i = 1;
       while ( i <= ITEMS)
       {
         printf("Enter code, quantity, and rate:");
         scanf("%s %d %f", code[i], &quantity[i],&rate[i]);
         i++;
       }
     /*.......Printing of Table and Column Headings.......*/
       printf("\n\n");
       printf("      INVENTORY REPORT      \n");
       printf("-----------------------\n");
       printf("  Code Quantity Rate Value  \n");
       printf("-----------------------\n");
     /*.......Preparation of Inventory Position..........*/
       total_value = 0;
       i = 1;
       while ( i <= ITEMS)
       {
```

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

(i) To print the data left-justified, we must use _____in the field specification.
(j) The specifier _____ prints floating-point values in the scientific notation.

4.3 Distinguish between the following pairs:

(a) ***getchar*** and ***scanf*** functions.
(b) %s and %c specifications for reading.
(c) %s and %[ ] specifications for reading.
(d) %g and %f specification for printing.
(e) %f and %e specifications for printing.

4.4 Write **scanf** statements to read the following data lists:

(a) 78 B 45
(b) 123 1.23 45A
(c) 15-10-2002
(d) 10 TRUE 20

4.5 State the outputs produced by the following **printf** statements.

(a) printf ("%d%c%f", 10, 'x', 1.23);
(b) printf ("%2d %c %4.2f", 1234,, 'x', 1.23);
(c) printf ("%d\t%4.2f", 1234, 456);
(d) printf ("\"%08.2f\"", 123.4);
(e) printf ("%d%d %d", 10, 20);

For questions 4.6 to 4.10 assume that the following declarations have been made in the program:

```
int year, count;
float amount, price;
char code, city[10];
double root;
```

4.6 State errors, if any, in the following input statements.

(a) scanf("%c%f%d", city, &price, &year);
(b) scanf("%s%d", city, amount);
(c) scanf("%f, %d, &amount, &year);
(d) scanf(\n"%f", root);
(e) scanf("%c %d %ld", *code, &count, Root);

4.7 What will be the values stored in the variables **year** and **code** when the data

1988, x

is keyed in as a response to the following statements:

(a) scanf("%d %c", &year, &code);
(b) scanf("%c %d", &year, &code);
(c) scanf("%d %c", &code, &year);
(d) scanf("%s %c", &year, &code);

4.8 The variables **count, price**, and **city** have the following values:

count <—— 1275
price <—— –235.74
city <—— Cambridge

Show the exact output that the following output statements will produce:

(a) printf("%d %f", count, price);
(b) printf("%2d\n%f", count, price);
(c) printf("%d %f", price, count);
(d) printf("%10dxxxx%5.2f",count, price);

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

transfers the control to a particular statement. This point of program has two *paths* to follow, one for the *true* condition and the other for the *false* condition as shown in Fig. 5.1.

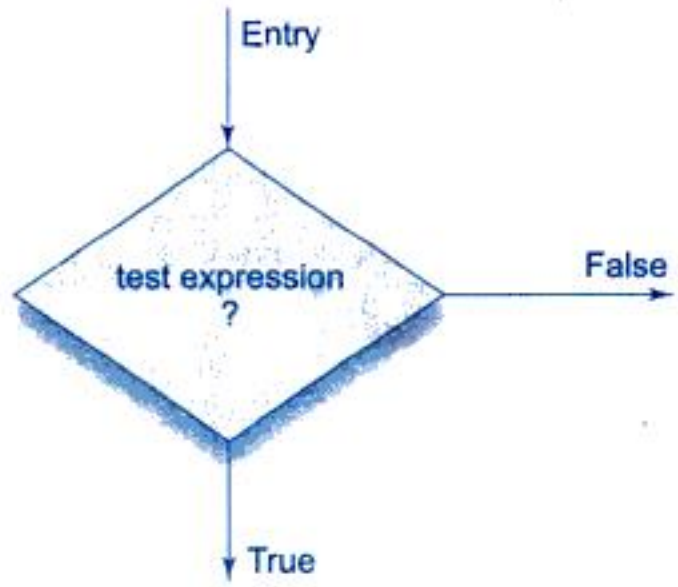


Fig. 5.1 *Two-way branching*

Some examples of decision making, using **if** statements are:

1. **if** (bank balance is zero)
   borrow money
2. **if** (room is dark)
   put on lights
3. **if** (code is 1)
   person is male
4. **if** (age is more than 55)
   person is retired

The **if** statement may be implemented in different forms depending on the complexity of conditions to be tested. The different forms are:

1. Simple **if** statement
2. **if.....else** statement
3. Nested **if....else** statement
4. **else if** ladder.

We shall discuss each one of them in the next few sections.

5.3 SIMPLE IF STATEMENT

The general form of a simple **if** statement is

```
if (test expression)
 {
     statement-block;
 }
statement-x;
```

The 'statement-block' may be a single statement or a group of statements. If the *test expression* is true, the *statement-block* will be executed; otherwise the statement-block will be skipped and the execution will jump to the *statement-x*. Remember, when the condition is

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Applying De Morgan's Rule

While designing decision statements, we often come across a situation where the logical NOT operator is applied to a compound logical expression, like !(x&&y||!z). However, a positive logic is always easy to read and comprehend than a negative logic. In such cases, we may apply what is known as **De Morgan's** rule to make the total expression positive. This rule is as follows:

"Remove the parentheses by applying the NOT operator to every logical expression component, while complementing the relational operators"

That is,

x becomes !x

!x becomes x

&& becomes ||

|| becomes &&

Examples:

!(x && y || !z) becomes !x || !y && z

!(x <=0 || !condition) becomes x >0&& condition

5.4 THE IF.....ELSE STATEMENT

The **if...else** statement is an extension of the simple **if** statement. The general form is

```
If (test expression)
    {
      True-block statement(s)
    }
else
    {
      False-block statement(s)
    }
statement-x
```

If the *test expression* is true, then the *true-block statement(s),* immediately following the **if** statements are executed; otherwise, the *false-block statement(s)* are executed. In either case, either *true-block* or *false-block* will be executed, not both. This is illustrated in Fig. 5.5. In both the cases, the control is transferred subsequently to the statement-x.

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
if (test condition-1)
{
  if (test condition-2);
  {
    statement -1;
  }
  else
  {
    statement -2;
  }
}
else
{
  statement -3;
}
statement -x;
```

statement-1 will be evaluated; otherwise the statement-2 will be evaluated and then the control is transferred to the statement-x.

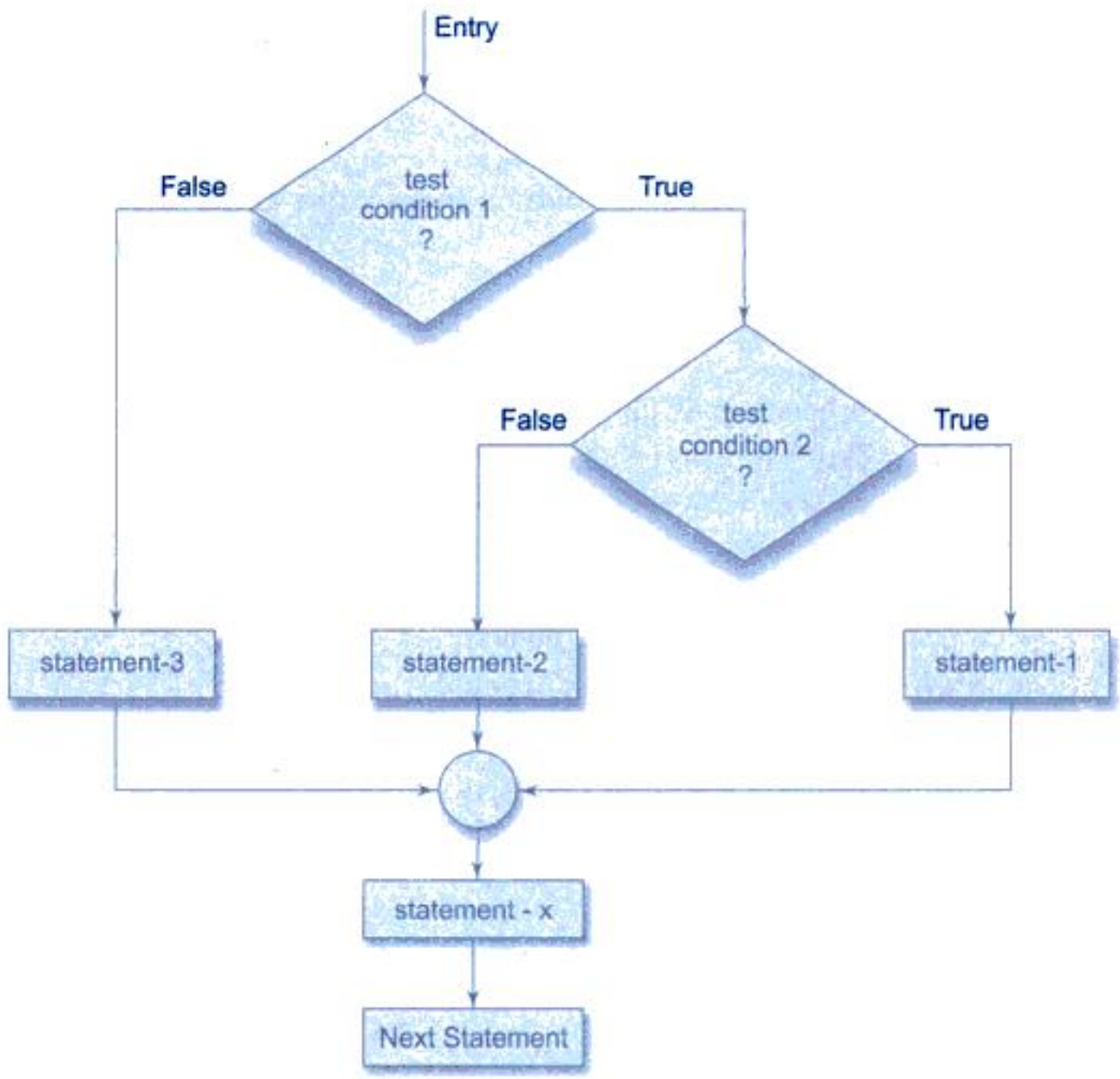


Fig. 5.7 *Flow chart of nested* ***if...else*** *statements*

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
                grade = "Fail";
        printf ("%s\n", grade);
```

Consider another example given below:

```
----
----
if (code == 1)
  colour = "RED";
else if (code == 2)
     colour = "GREEN";
  else if (code == 3)
         colour = "WHITE";
       else
         colour = "YELLOW";
---
---
```

Code numbers other than 1, 2 or 3 are considered to represent YELLOW colour. The same results can be obtained by using nested **if...else** statements.

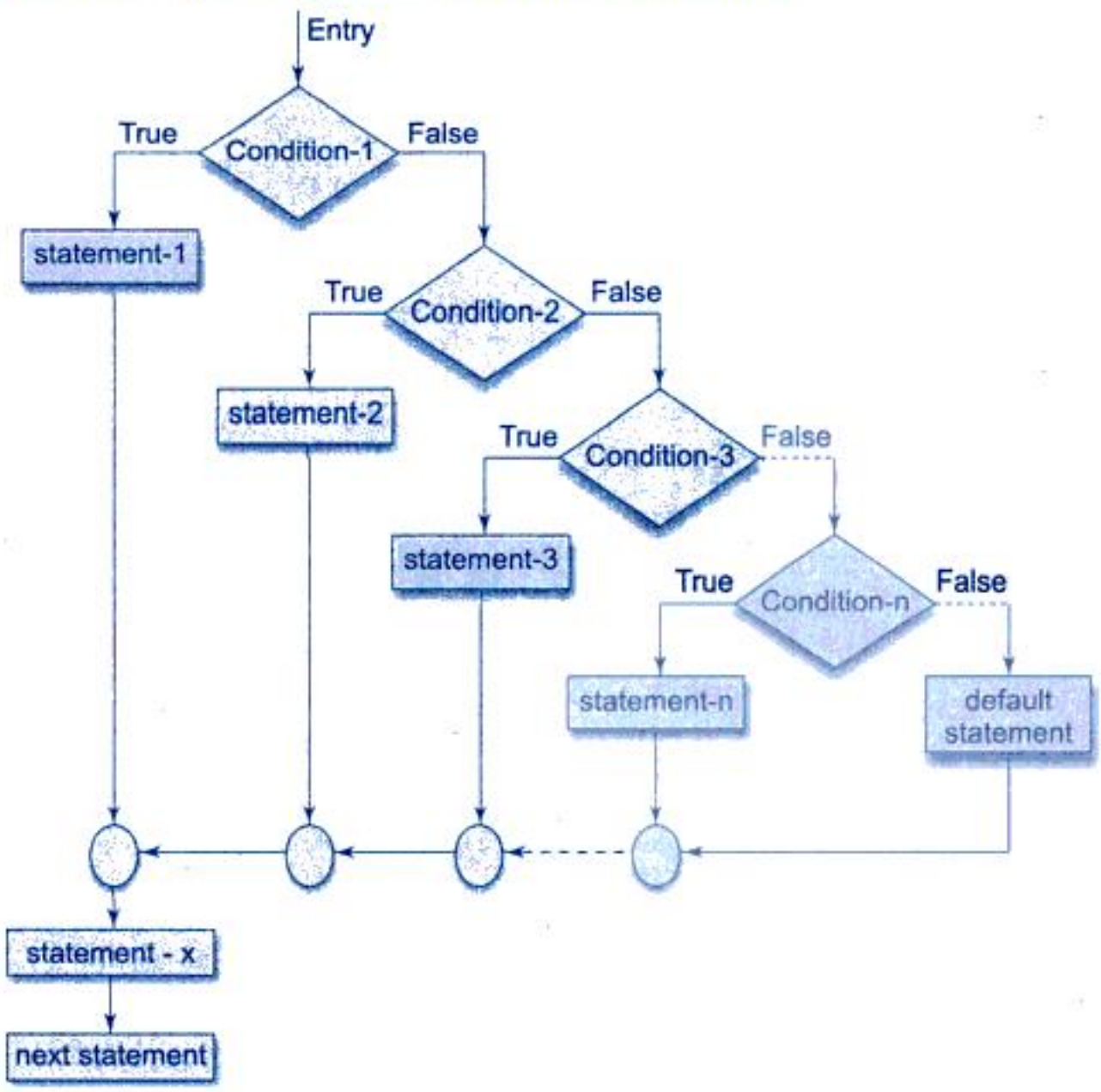


Fig. 5.9 *Flow chart of **else..if** ladder*

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

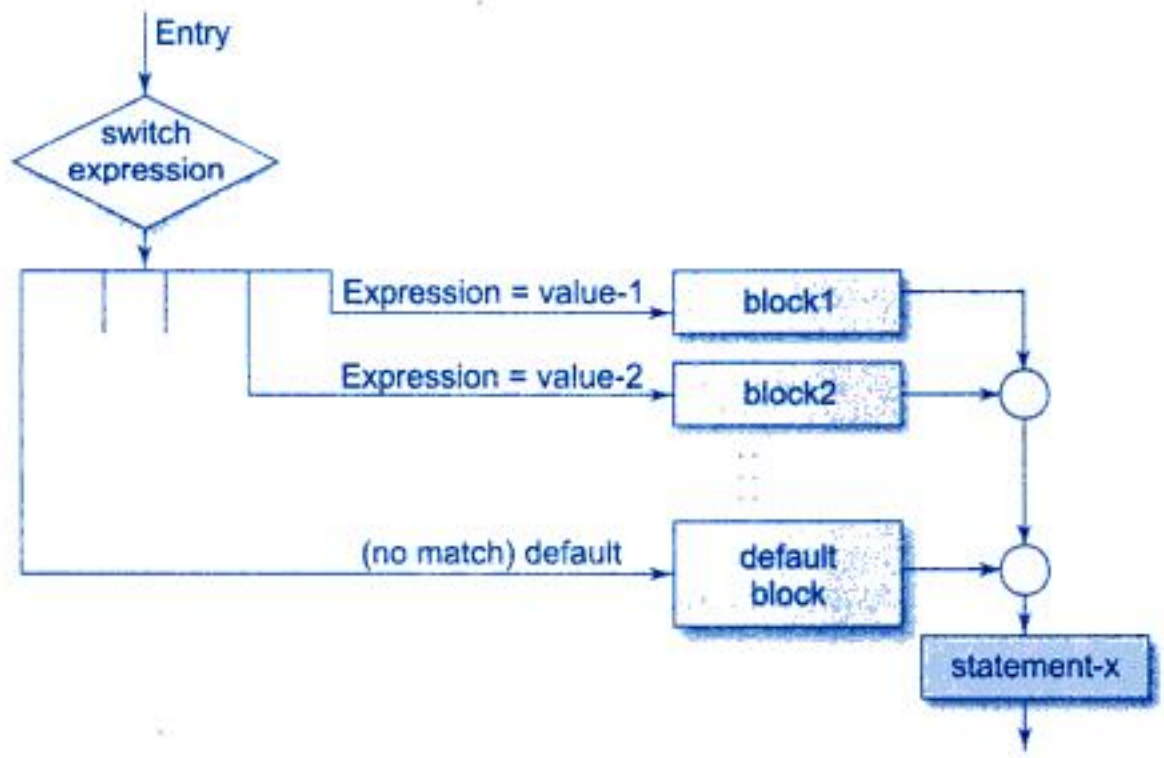


Fig. 5.11 *Selection process of the* ***switch*** *statement*

The **switch** statement can be used to grade the students as discussed in the last section. This is illustrated below:

```
---
---
index = marks/10
switch (index)
{
   case 10:
   case 9:
   case 8:
        grade = "Honours";
        break;
   case 7:
   case 6:
        grade = "First Division";
        break;
   case 5:
        grade = "Second Division";
        break;
   case 4:
        grade = "Third Division";
        break;
   default:
        grade = "Fail";
        break;
}
printf("%s\n", grade);
---
---
```

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
                    MAXLOAN - loan2 : loan3);
        printf("\n\n");
        printf("Previous loans pending:\n%ld %ld\n",loan1,loan2);
        printf("Loan requested = %ld\n", loan3);
        printf("Loan sanctioned = %ld\n", sancloan);
    }
```

Output

```
Enter the values of previous two loans:
0 20000
Enter the value of new loan:
45000
Previous loans pending:
0 20000
Loan requested = 45000
Loan sanctioned = 30000
Enter the values of previous two loans:
1000 15000
Enter the value of new loan:
25000
Previous loans pending:
1000 15000
Loan requested  = 25000
Loan sanctioned = 0
```

Fig. 5.12 *Illustration of the conditional operator*

The program uses the following variables:

loan3 - present loan amount requested
loan2 - previous loan amount pending
loan1 - previous to previous loan pending
sum23 - sum of loan2 and loan3
sancloan - loan sanctioned

The rules for sanctioning new loan are:

1. loan1 should be zero.
2. loan2 + loan3 should not be more than MAXLOAN.

Note the use of **long int** type to declare variables.

Some Guidelines for Writing Multiway Selection Statements

Complex multiway selection statements require special attention. The readers should be able to understand the logic easily. Given below are some guidelines that would help improve readability and facilitate maintenance.

- Avoid compound negative statements. Use positive statements wherever possible.

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Just Remember

- Be aware of dangling **else** statements.
- Be aware of any side effects in the control expression such as if(x++).
- Use braces to encapsulate the statements in **if** and **else** clauses of an if.... else statement.
- Check the use of =operator in place of the equal operator = =.
- Do not give any spaces between the two symbols of relational operators = =, !=, >= and <=.
- Writing !=, >= and <= operators like =!, => and =< is an error.
- Remember to use two ampersands (&&) and two bars (||) for logical operators. Use of single operators will result in logical errors.
- Do not forget to place parentheses for the if expression.
- It is an error to place a semicolon after the if expression.
- Do not use the equal operator to compare two floating-point values. They are seldom exactly equal.
- Do not forget to use a break statement when the cases in a switch statement are exclusive.
- Although it is optional, it is a good programming practice to use the default clause in a switch statement.
- It is an error to use a variable as the value in a case label of a switch statement. (Only integral constants are allowed.)
- Do not use the same constant in two case labels in a switch statement.
- Avoid using operands that have side effects in a logical binary expression such as (x--&&++y). The second operand may not be evaluated at all.
- Try to use simple logical expressions.

Case Studies

1. Range of Numbers

Problem: A survey of the computer market shows that personal computers are sold at varying costs by the vendors. The following is the list of costs (in hundreds) quoted by some vendors:

35.00,	40.50,	25.00,	31.25,	68.15,
47.00,	26.65,	29.00	53.45,	62.50

Determine the average cost and the range of values.

Problem analysis: Range is one of the measures of dispersion used in statistical analysis of a series of values. The range of any series is the difference between the highest and the lowest values in the series. That is

Range = highest value – lowest value

It is therefore necessary to find the highest and the lowest values in the series.

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
            goto stop;
        }
        house_rent = 0.25 * basic;
        gross = basic + house_rent + perks;
        if (gross <= 2000)
          incometax = 0;
        else if (gross <= 4000)
              incometax = 0.03 * gross;
            else if (gross <= 5000)
                 incometax = 0.05 * gross;
              else
                 incometax = 0.08 * gross;
        net = gross - incometax;
        printf("%d %d %.2f\n", level, jobnumber, net);
        goto input;
        stop: printf("\n\nEND OF THE PROGRAM");
    }
```

Output

```
Enter level, job number, and basic pay
Enter 0 (zero) for level to END
1 1111 4000
1 1111 5980.00
Enter level, job number, and basic pay
Enter 0 (zero) for level to END
2 2222 3000
2 2222 4465.00
Enter level, job number, and basic pay
Enter 0 (zero) for level to END
3 3333 2000
3 3333 3007.00
Enter level, job number, and basic pay
Enter 0 (zero) for level to END
4 4444 1000
4 4444 1500.00
Enter level, job number, and basic pay
Enter 0 (zero) for level to END
0
END OF THE PROGRAM
```

Fig. 5.15 *Pay-bill calculations*

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

5.13 What is the output of the following program?

```
main ( )
{
        int m = 1;
        if ( m==1)
        {
                printf ( " Delhi " ) ;
                if (m == 2)
                printf( "Chennai" ) ;
                else
                printf("Bangalore") ;
        }
        else;
        printf(" END");
 }
```

5.14 What is the output of the following program?

```
main( )
{
        int m ;
        for (m = 1; m<5; m++)
                printf(%d\n", (m%2) ? m : m*2);
}
```

5.15 What is the output of the following program?

```
main( )
{
        int m, n, p ;
        for ( m = 0; m < 3; m++ )
        for (n = 0; n<3; n++ )
        for ( p = 0; p < 3;; p++ )
        if ( m + n + p == 2 )
        goto print;

        print :
        printf("%d, %d, %d", m, n, p);
}
```

5.16 What will be the value of x when the following segment is executed?

```
        int x = 10, y = 15;
        x = (x<y)? (y+x) : (y-x) ;
```

5.17 What will be the output when the following segment is executed?

```
int x = 0;
if (x >= 0)
if ( x > 0 )
```

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

$$x_2 = -b - \frac{\sqrt{b^2 - 4\,ac}}{2a}$$

The program should request for the values of the constants a, b and c and print the values of x_1 and x_2. Use the following rules:

(a) No solution, if both a and b are zero
(b) There is only one root, if a = 0 (x = –c/b)
(c) There are no real roots, if $b^2 - 4$ ac is negative
(d) Otherwise, there are two real roots

Test your program with appropriate data so that all logical paths are working as per your design. Incorporate appropriate output messages.

5.11 Write a program to read three integer values from the keyboard and displays the output stating that they are the sides of right-angled triangle.

5.12 An electricity board charges the following rates for the use of electricity:
For the first 200 units: 80 P per unit
For the next 100 units: 90 P per unit
Beyond 300 units: Rs 1.00 per unit
All users are charged a minimum of Rs. 100 as meter charge. If the total amount is more than Rs. 400, then an additional surcharge of 15% of total amount is charged.
Write a program to read the names of users and number of units consumed and print out the charges with names.

5.13 Write a program to compute and display the sum of all integers that are divisible by 6 but not divisible by 4 and lie between 0 and 100. The program should also count and display the number of such values.

5.14 Write an interactive program that could read a positive integer number and decide whether the number is a prime number and display the output accordingly.
Modify the program to count all the prime numbers that lie between 100 and 200.
NOTE: A prime number is a positive integer that is divisible only by 1 or by itself.

5.15 Write a program to read a double-type value x that represents angle in radians and a character-type variable T that represents the type of trigonometric function and display the value of

(a) sin(x), if s or S is assigned to T,
(b) cos (x), if c or C is assigned to T, and
(c) tan (x), if t or T is assigned to T

using (i) **if.....else** statement and (ii) **switch** statement.

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
while (test condition)
{
    body of the loop
}
```

The **while** is an *entry-controlled* loop statement. The *test-condition* is evaluated and if the condition is *true*, then the body of the loop is executed. After execution of the body, the test-condition is once again evaluated and if it is true, the body is executed once again. This process of repeated execution of the body continues until the test-condition finally becomes *false* and the control is transferred out of the loop. On exit, the program continues with the statement immediately after the body of the loop.

The body of the loop may have one or more statements. The braces are needed only if the body contains two or more statements. However, it is a good practice to use braces even if the body has only one statement.

We can rewrite the program loop discussed in Section 6.1 as follows:

```
========
sum = 0;
n = 1;                          /* Initialization */
while(n <= 10)                  /* Testing */
{
  sum = sum + n * n;
    n = n+1;                    /* Incrementing */
}
printf("sum = %d\n", sum);
========
```

loop

The body of the loop is executed 10 times for n = 1, 2, ..., 10, each time adding the square of the value of n, which is incremented inside the loop. The test condition may also be written as **n < 11**; the result would be the same. This is a typical example of counter-controlled loops. The variable n is called ***counter*** or ***control variable***.

Another example of **while** statement, which uses the keyboard input is shown below:

```
=========
character = ' ' ;
while (character != 'Y')
        character = getchar();
xxxxxxx;
=========
```

First the **character** is initialized to ' '. The **while** statement then begins by testing whether **character** is not equal to Y. Since the **character** was initialized to ' ', the test is true and the loop statement

```
character = getchar();
```

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

nated as soon as the desired name is found. C permits a *jump* from one statement to another within a loop as well as a *jump* out of a loop.

Jumping Out of a Loop

An early exiti from a loop can be accomplished by using the **break** statement or the **goto** statement. We have already seen the use of the **break** in the **switch** statement and the **goto** in the **if...else** construct. These statements can also be used within **while, do**, or **for** loops. They are illustrated in Fig. 6.6 and Fig. 6.7.

When a **break** statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop. When the loops are nested, the **break** would only exit from the loop containing it. That is, the **break** will exit only a single loop.

Since a **goto** statement can transfer the control to any place in a program, it is useful to provide branching within a loop. Another important use of **goto** is to exit from deeply nested loops when an error occurs. A simple **break** statement would not work here.

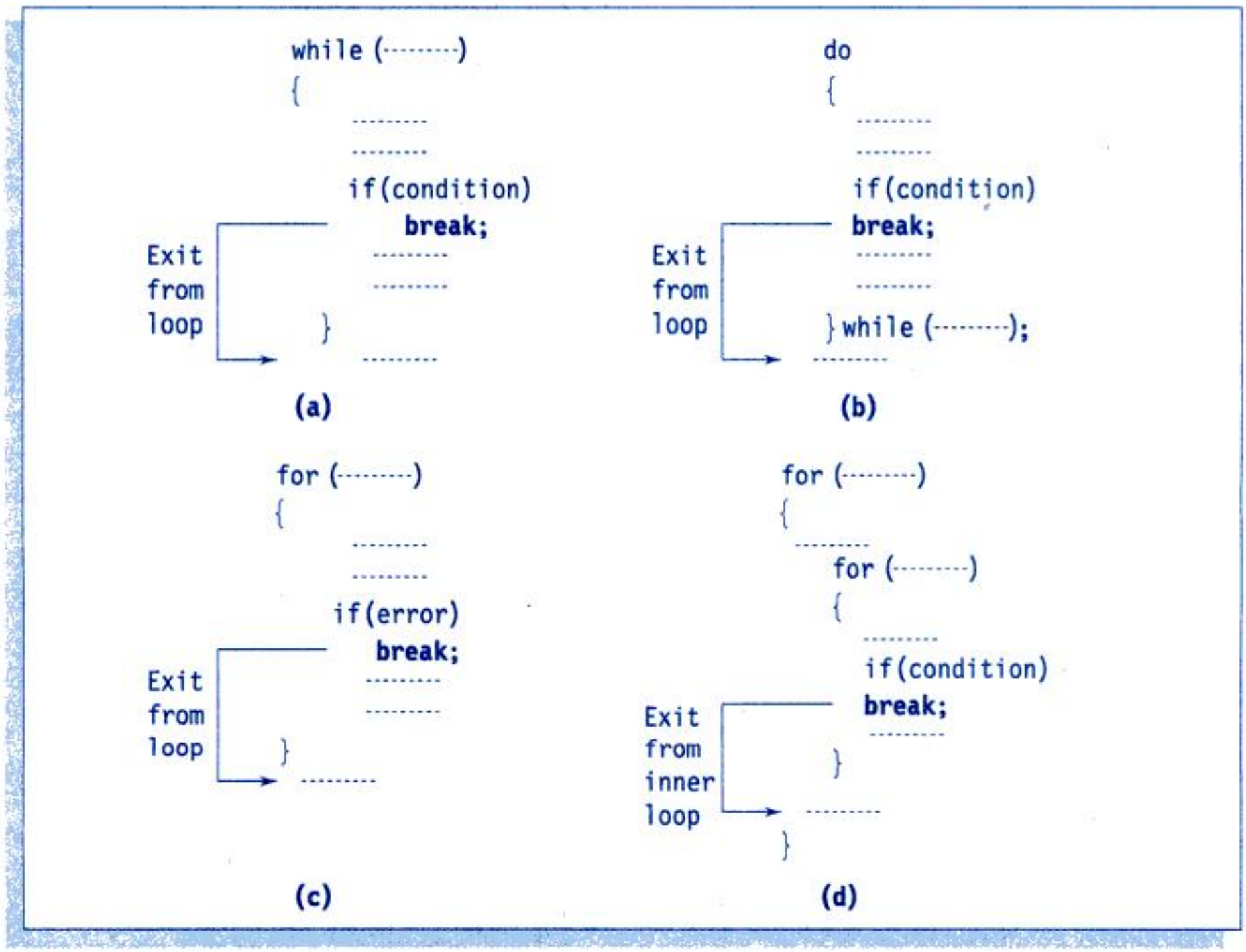


Fig. 6.6 *Exiting a loop with* ***break*** *statement*

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

- Repetition (looping) structure

While sequence and loop structures are sufficient to meet all the requirements of programming, the selection structure proves to be more convenient in some situations.

The use of structured programming techniques helps ensure well-designed programs that are easier to write, read, debug and maintain compared to those that are unstructured.

Structured programming discourages the implementation of unconditional branching using jump statements such as **goto**, **break** and **continue**. In its purest form, structured programming is synonymous with *"goto less programming"*.

Do not go to goto statement!

Skipping a Part of a Loop

During the loop operations, it may be necessary to skip a part of the body of the loop under certain conditions. For example, in processing of applications for some job, we might like to exclude the processing of data of applicants belonging to a certain category. On reading the category code of an applicant, a test is made to see whether his application should be considered or not. If it is not to be considered, the part of the program loop that processes the application details is skipped and the execution continues with the next loop operation.

Like the **break** statement, C supports another similar statement called the **continue** statement. However, unlike the **break** which causes the loop to be terminated, the **continue**, as the name implies, causes the loop to be continued with the next iteration after skipping any statements in between. The **continue** statement tells the compiler, "SKIP THE FOLLOWING STATEMENTS AND CONTINUE WITH THE NEXT ITERATION". The format of the **continue** statement is simply

```
continue;
```

The use of the **continue** statement in loops is illustrated in Fig. 6.10. In **while** and **do** loops, **continue** causes the control to go directly to the test-condition and then to continue the iteration process. In the case of **for** loop, the increment section of the loop is executed before the test-condition is evaluated.

```
while (test-condition)
{
 ---------
 if (---------)
   continue;
 ---------
 ---------
}
```

(a)

```
do
{
  ---------
  if (---------)
    continue;
  ---------
  ---------
} while (test-condition);
```

(b)

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Output

```
                    Y ------->
0 ..................................................................
  :                                                              #
  :                                                        0 --- *
  :                                                    0------ *
  :                                                0------- *
  :                                            0------ *
  :                                        0------ *
  :                                    0---- *
  :                                0- *
  :                            #
  :                        * - 0
X :                    * --- 0
  :                *----- 0
  :            *------0
  :          *-------0
  :       *-------0
  :      *-------0
  :    *-------0
  :   *-------0
  :  *------0
  :  *-----0
```

Fig. 6.15 *Plotting of two functions*

Review Questions

6.1 State whether the following statements are *true* or *false*.

(a) The **do...while** statement first executes the loop body and then evaluate the loop control expression.

(b) In a pretest loop, if the body is executed **n** times, the test expression is executed **n** + 1 times.

(c) The number of times a control variable is updated always equals the number of loop iterations.

(d) Both the pretest loops include initialization within the statement.

(e) In a **for** loop expression, the starting value of the control variable must be less than its ending value.

(f) The initialization, test condition and increment parts may be missing in a **for** statement.

(g) **while** loops can be used to replace **for** loops without any change in the body of the loop.

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

6.17 Write a program to graph the function

$$y = \sin(x)$$

in the interval 0 to 180 degrees in steps of 15 degrees. Use the concepts discussed in the Case Study 4 in Chapter 6.

6.18 Write a program to print all integers that are **not divisible** by either 2 or 3 and lie between 1 and 100. Program should also account the number of such integers and print the result.

6.19 Modify the program of Exercise 6.16 to print the character O instead of S at the center of the square as shown below.

```
S S S S S
S S S S S
S S O S S
S S S S S
S S S S S
```

6.20 Given a set of 10 two-digit integers containing both positive and negative values, write a program using **for** loop to compute the sum of all positive values and print the sum and the number of values added. The program should use **scanf** to read the values and terminate when the sum exceeds 999. Do not use **goto** statement.

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The subscripts of an array can be integer constants, integer variables like i, or expressions that yield integers. *C performs no bounds checking and, therefore, care should be exercised to ensure that the array indices are within the declared limits.*

7.3 DECLARATION OF ONE-DIMENSIONAL ARRAYS

Like any other variable, arrays must be declared before they are used so that the compiler can allocate space for them in memory. The general form of array declaration is

type *variable-name[size];*

The *type* specifies the type of element that will be contained in the array, such as **int, float,** or **char** and the *size* indicates the maximum number of elements that can be stored inside the array. For example,

```
float height[50];
```

declares the **height** to be an array containing 50 real elements. Any subscripts 0 to 49 are valid. Similarly,

```
int group[10];
```

declares the **group** as an array to contain a maximum of 10 integer constants. Remember:

- Any reference to the arrays outside the declared limits would not necessarily cause an error. Rather, it might result in unpredictable program results.
- The size should be either a numeric constant or a symbolic constant.

The C language treats character strings simply as arrays of characters. The *size* in a character string represents the maximum number of characters that the string can hold. For instance,

```
char name[10];
```

declares the **name** as a character array (string) variable that can hold a maximum of 10 characters. Suppose we read the following string constant into the string variable **name.**

"WELL DONE"

Each character of the string is treated as an element of the array **name** and is stored in the memory as follows:

'W'
'E'
'L'
'L'
' '
'D'
'O'
'N'
'E'
'\0'

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

We can also use a read function such as **scanf** to initialize an array. For example, the statements

```
int x [3];
scanf("%d%d%d", &x[0], &[1], &x[2]);
```

will initialize array elements with the values entered through the keyboard.

Example 7.2 Given below is the list of marks obtained by a class of 50 students in an annual examination.

43 65 51 27 79 11 56 61 82 09 25 36 07 49 55 63 74 81 49 37
40 49 16 75 87 91 33 24 58 78 65 56 76 67 45 54 36 63 12 21
73 49 51 19 39 49 68 93 85 59

Write a program to count the number of students belonging to each of following groups of marks: 0–9, 10–19, 20–29,.....,100.

The program coded in Fig. 7.2 uses the array **group** containing 11 elements, one for each range of marks. Each element counts those values falling within the range of values it represents.

For any value, we can determine the correct group element by dividing the value by 10. For example, consider the value 59. The integer division of 59 by 10 yields 5. This is the element into which 59 is counted.

Program

```
#define  MAXVAL   50
#define  COUNTER  11
main()
{
     float      value[MAXVAL];
     int        i, low, high;
     int  group[COUNTER] = {0,0,0,0,0,0,0,0,0,0,0};
     /* . . . . . . . .READING AND COUNTING . . . . . .*/
     for( i = 0 ; i < MAXVAL ; i++ )
     {
     /*. . . . . . . .READING OF VALUES . . . . . . . . */
       scanf("%f", &value[i]) ;
     /*. . . . . .COUNTING FREQUENCY OF GROUPS. . . . . */
       ++ group[ (int) ( value[i]) / 10] ;
     }
     /* . . . .PRINTING OF FREQUENCY TABLE . . . . . . .*/
     printf("\n");
     printf(" GROUP     RANGE     FREQUENCY\n\n") ;
     for( i = 0 ; i < COUNTER ; i++ )
     {
          low = i * 10 ;
          if(i == 10)
            high = 100 ;
```

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

$$= \sum_{i=0}^{3} \text{girl_total}[i]$$

$$= \sum_{j=0}^{2} \text{item_total}[j]$$

Program

```
#define  MAXGIRLS 4
#define  MAXITEMS 3

main()
{
     int  value[MAXGIRLS][MAXITEMS];
     int  girl_total[MAXGIRLS] , item_total[MAXITEMS];
     int  i, j, grand_total;
/*.......READING OF VALUES AND COMPUTING girl_total ...*/

     printf("Input data\n");
     printf("Enter values, one at a time, row-wise\n\n");

     for( i = 0 ; i < MAXGIRLS ; i++ )
     {
          girl_total[i] = 0;
          for( j = 0 ; j < MAXITEMS ; j++ )
          {
               scanf("%d", &value[i][j]);
               girl_total[i] = girl_total[i] + value[i][j];
          }
     }
/*.......COMPUTING item_total.........................*/

     for( j = 0 ; j < MAXITEMS ; j++ )
     {
          item_total[j] = 0;
          for( i =0 ; i < MAXGIRLS ; i++ )
               item_total[j] = item_total[j] + value[i][j];
     }
/*.......COMPUTING grand_total.......................*/

     grand_total = 0;
     for( i =0 ; i < MAXGIRLS ; i++ )
       grand_total = grand_total + girl_total[i];
/* .......PRINTING OF RESULTS.........................*/

     printf("\n GIRLS TOTALS\n\n");
```

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

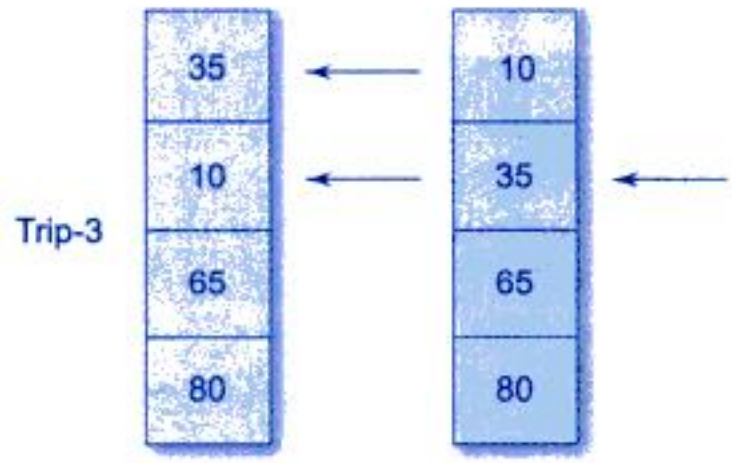
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



During the first trip, three pairs of items are compared and interchanged whenever needed. It should be noted that the number 80, the largest among the items, has been moved to the bottom at the end of the first trip. This means that the element 80 (the last item in the new list) need not be considered any further. Therefore, trip-2 requires only two pairs to be compared. This time, the number 65 (the second largest value) has been moved down the list. Notice that each trip brings the smallest value 10 up by one level.

The number of steps required in a trip is reduced by one for each trip made. The entire process will be over when a trip contains only one step. If the list contains **n** elements, then the number of comparisons involved would be **n(n–1)/2.**

Program

```
#define N 10
main( )
{
   int i,j,n;
   float median,a[N],t;
   printf("Enter the number of items\n");
   scanf("%d", &n);
/*Reading items into array a */
   printf("Input %d values \n",n);
   for (i = 1; i <= n ; i++)
      scanf("%f", &a[i]);
/*Sorting begins */
   for (i = 1 ; i <= n-1 ; i++)
   {    /* Trip-i begins */
      for (j = 1 ; j <= n-i ; j++)
      {
            if (a[j] <= a[j+1])
            { /* Interchanging values */
               t = a[j];
               a[j] = a[j+1];
               a[j+1] = t;
            }
            else
               continue ;
```

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
    for(i=1; i<=4; i++)
      for(j=1;j<=5; j++)
        scanf("%d",&M[i][j]);
    printf (" Enter products sold week_wise\n");
    printf (" S11,S12,—, S21,S22,— etc\n");
    for(i=1; i<=4; i++)
      for(j=1; j<=5; j++)
        scanf("%d", &S[i][j]);
    printf(" Enter cost of each product\n");
      for(j=1; j <=5; j++)
        scanf("%d",&C[j]);
  /*Value matrices of production and sales */
    for(i=1; i<=4; i++)
      for(j=1; j<=5; j++)
      {
        Mvalue[i][j] = M[i][j] * C[j];
        Svalue[i][j] = S[i][j] * C[j];
      }
  /*Total value of weekly production and sales */
    for(i=1; i<=4; i++)
    {
      Mweek[i] = 0 ;
      Sweek[i] = 0 ;
      for(j=1; j<=5; j++)
      {
        Mweek[i] += Mvalue[i][j];
        Sweek[i] += Svalue[i][j];
      }
    }
  /*Monthly value of product_wise production and sales */
    for(j=1; j<=5; j++)
    {
      Mproduct[j] = 0 ;
      Sproduct[j] = 0 ;
      for(i=1; i<=4; i++)
      {
        Mproduct[j] += Mvalue[i][j];
        Sproduct[j] += Svalue[i][j];
      }
    }
  /*Grand total of production and sales values */
    Mtotal = Stotal = 0;
    for(i=1; i<=4; i++)
    {
      Mtotal += Mweek[i];
      Stotal += Sweek[i];
```

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
Total production     = 5260
Total sales    = 4550
ENTER YOUR CHOICE:5
GOOD  BYE
Exit from the program
```

Fig. 7.10 *Program for production and sales analysis*

Review Questions

7.1 State whether the following statements are *true* or *false*.

(a) The type of all elements in an array must be the same.
(b) When an array is declared, C automatically initializes its elements to zero.
(c) An expression that evaluates to an integral value may be used as a subscript.
(d) Accessing an array outside its range is a compile time error.
(e) A **char** type variable cannot be used as a subscript in an array.
(f) An unsigned long int type can be used as a subscript in an array.
(g) In C, by default, the first subscript is zero.
(h) When initializing a multidimensional array, not specifying all its dimensions is an error.
(i) When we use expressions as a subscript, its result should be always greater than zero.
(j) In C, we can use a maximum of 4 dimensions for an array.
(k) In declaring an array, the array size can be a constant or variable or an expression.
(l) The declaration int x[2] = {1,2,3}; is illegal.

7.2 Fill in the blanks in the following statements.

(a) The variable used as a subscript in an array is popularly known as _____ variable.
(b) An array can be initialized either at compile time or at _____.
(c) An array created using **malloc** function at run time is referred to as _____ array.
(d) An array that uses more than two subscript is referred to as _____ array.
(e) _____is the process of arranging the elements of an array in order.

7.3 Identify errors, if any, in each of the following array declaration statements, assuming that ROW and COLUMN are declared as symbolic constants:

(a) `int score (100);`
(b) `float values [10,15];`
(c) `float average[ROW],[COLUMN];`
(d) `char name[15];`
(e) `int sum[ ];`
(f) `double salary [i + ROW]`
(g) `long int number [ROW]`
(h) `int array x[COLUMN];`

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Write a program to read the data and determine the following:

(a) Total marks obtained by each student.
(b) The highest marks in each subject and the Roll No. of the student who secured it.
(c) The student who obtained the highest total marks.

7.6 Given are two one-dimensional arrays A and B which are sorted in ascending order. Write a program to **merge** them into a single sorted array C that contains every item from arrays A and B, in ascending order.

7.7 Two matrices that have the same number of rows and columns can be multiplied to produce a third matrix. Consider the following two matrices.

$$A = \begin{bmatrix} a_{11}\ a_{12} \ldots\ldots a_{1n} \\ a_{12}\ a_{22} \ldots\ldots a_{2n} \\ \cdot \qquad\qquad \cdot \\ \cdot \qquad\qquad \cdot \\ \cdot \qquad\qquad \cdot \\ a_{n1} \ldots\ldots\ a_{nn} \end{bmatrix}$$

$$B = \begin{bmatrix} b_{11}\ b_{12} \ldots\ldots b_{1n} \\ b_{12}\ b_{22} \ldots\ldots b_{2n} \\ \cdot \qquad\qquad \cdot \\ \cdot \qquad\qquad \cdot \\ \cdot \qquad\qquad \cdot \\ b_{n1} \ldots\ldots\ b_{nn} \end{bmatrix}$$

The product of **A** and **B** is a third matrix C of size n×n where each element of C is given by the following equation.

$$C_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Write a program that will read the values of elements of A and B and produce the product matrix **C.**

7.8 Write a program that fills a five-by-five matrix as follows:

- Upper left triangle with +1s
- Lower right triangle with –1s
- Right to left diagonal with zeros

Display the contents of the matrix using not more than two **printf** statements

7.9 Selection sort is based on the following idea:
Selecting the largest array element and swapping it with the last array element leaves an unsorted list whose size is 1 less than the size of the original list. If we repeat this step again on the unsorted list we will have an ordered list of size 2 and an unordered list size n–2 . When we repeat this until the size of the unsorted list becomes one, the result will be a sorted list.
Write a program to implement this algorithm.

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

This will result in a compile time error. Also note that we cannot separate the initialization from declaration. That is,

```
char str3[5];
str3 = "GOOD";
```

is not allowed. Similarly,

```
char s1[4] = "abc";
char s2[4];
s2 = s1; /* Error */
```

is not allowed. An array name cannot be used as the left operand of an assignment operator.

Terminating Null Character

You must be wondering, "why do we need a terminating null character?" As we know, a string is not a data type in C, but it is considered a data structure stored in an array. The string is a variable-length structure and is stored in a fixed-length array. The array size is not always the size of the string and most often it is much larger than the string stored in it. Therefore, the last element of the array need not represent the end of the string. We need some way to determine the end of the string data and the null character serves as the "end-of-string" marker.

8.3 READING STRINGS FROM TERMINAL

Using scanf Function

The familiar input function **scanf** can be used with **%s** format specification to read in a string of characters. Example:

```
char address[10]
scanf("%s", address);
```

The problem with the **scanf** function is that it terminates its input on the first white space it finds. A white space includes blanks, tabs, carriage returns, form feeds, and new lines. Therefore, if the following line of text is typed in at the terminal,

NEW YORK

then only the string "NEW" will be read into the array **address**, since the blank space after the word 'NEW' will terminate the reading of string.

The **scanf** function automatically terminates the string that is read with a null character and therefore the character array should be large enough to hold the input string plus the null character. Note that unlike previous **scanf** calls, in the case of character arrays, the ampersand (&) is not required before the variable name.

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
Output
    Enter text. Press <Return> at end
    Programming in C is interesting.
    Programming in C is interesting.
    Enter text. Press <Return> at end
    National Centre for Expert Systems, Hyderabad.
    National Centre for Expert Systems, Hyderabad.
```

Fig. 8.2 *Program to **read a line of text** from terminal*

Another and more convenient method of reading a string of text containing whitespaces is to use the library function **gets** available in the *<stdio.h>* header file. This is a simple function with one string parameter and called as under:

gets (str);

str is a string variable declared properly. It reads characters into **str** from the keyboard until a new-line character is encountered and then appends a null character to the string. Unlike **scanf,** it does not skip whitespaces. For example the code segment

```
char line [80];
gets (line);
printf ("%s", line);
```

reads a line of text from the keyboard and displays it on the screen. The last two statements may be combined as follows:

```
printf("%s", gets(line));
```

(Be careful not to input more character that can be stored in the string variable used. Since C does not check array-bounds, it may cause problems.)

C does not provide operators that work on strings directly. For instance we cannot assign one string to another directly. For example, the assignment statements.

```
string = "ABC";
string1 = string2;
```

are not valid. If we really want to copy the characters in **string2** into **string1**, we may do so on a character-by-character basis.

Example 8.3 Write a program to copy one string into another and count the number of characters copied.

The program is shown in Fig. 8.3. We use a **for** loop to copy the characters contained inside **string2** into the **string1**. The loop is terminated when the *null* character is reached. Note that we are again assigning a null character to the **string1.**

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
        d = c + 1;
        printf("|%-12.*s|\n", d, string);
      }
      printf("------------\n");
    }
```

Output

```
C
CP
CPr
CPro
CProg
CProgr
CProgra
CProgram
CProgramm
CProgrammi
CProgrammin
CProgramming
CProgramming

CProgrammin
CProgrammi
CProgramm
CProgram
CProgra
CProgr
CProg
CPro
CPr
CP
C
```

Fig. 8.5 *Illustration of variable field specifications by printing sequences of characters*

```
           C
          CP
         CPr
        CPro
       CProg
      CProgr
     CProgra
    CProgram
```

```
C|
CP|
CPr|
CPro|
CProg|
CProgr|
CProgra|
CProgram|
```

```
C|
 C|
  C|
   C|
    C|
     C|
      C|
       C|
```

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

this compares the left-most n characters of **s1** to **s2** and returns.

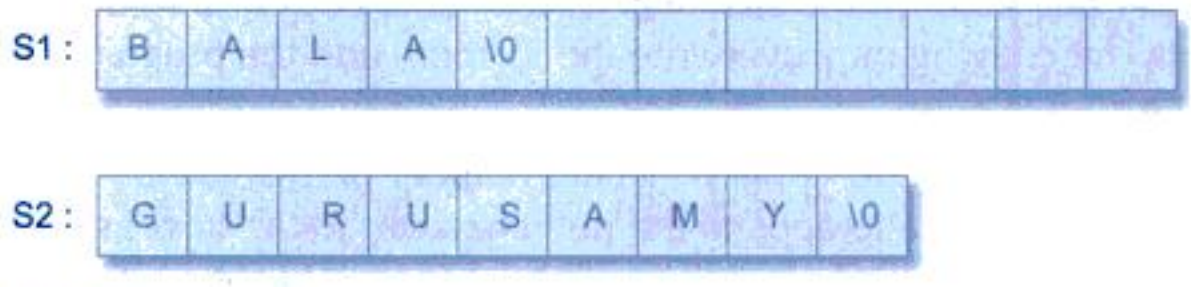
(a) 0 if they are equal;
(b) negative number, if s1 sub-string is less than s2; and
(c) positive number, otherwise.

strncat

This is another concatenation function that takes three parameters as shown below:

```
strncat (s1, s2, n);
```

This call will concatenate the left-most n characters of **s2** to the end of **s1.** Example:



After **strncat** (s1, s2, 4); execution:

strstr

It is a two-parameter function that can be used to locate a sub-string in a string. This takes the forms:

```
strstr (s1, s2);
strstr (s1, "ABC");
```

The function **strstr** searches the string **s1** to see whether the string **s2** is contained in **s1**. If yes, the function returns the position of the first occurrence of the sub-string. Otherwise, it returns a NULL pointer. Example.

```
if (strstr (s1, s2) == NULL)
  printf("substring is not found");
else
  printf("s2 is a substring of s1");
```

We also have functions to determine the existence of a character in a string. The function call

```
strchr(s1, 'm');
```

will locate the first occurrence of the character 'm' and the call

```
strrchr(s1, 'm');
```

will locate the last occurrence of the character 'm' in the string **s1**.

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

- Be aware the return values when using the functions **strcmp** and **strncmp** for comparing strings.
- When using string functions for copying and concatenating strings, make sure that the target string has enough space to store the resulting string. Otherwise memory overwriting may occur.
- The header file <stdio.h> is required when using standard I/O functions.
- The header file <ctype.h> is required when using character handling functions.
- The header file <stdlib.h> is required when using general utility functions.
- The header file <string.h> is required when using string manipulation functions.

Case Studies

1. Counting Words in a Text

One of the practical applications of string manipulations is counting the words in a text. We assume that a word is a sequence of any characters, except escape characters and blanks, and that two words are separated by one blank character. The algorithm for counting words is as follows:

1. Read a line of text.
2. Beginning from the first character in the line, look for a blank. If a blank is found, increment words by 1.
3. Continue steps 1 and 2 until the last line is completed.

The implementation of this algorithm is shown in Fig. 8.11. The first **while** loop will be executed once for each line of text. The end of text is indicated by pressing the 'Return' key an extra time after the entire text has been entered. The extra 'Return' key causes a newline character as input to the last line and as a result, the last line contains only the null character.

The program checks for this special line using the test

if (line[0] == '\0')

and if the first (and only the first) character in the line is a null character, then counting is terminated. Note the difference between a null character and a blank character.

Program

```
#include <stdio.h>
main()
{
   char line[81], ctr;
   int i,c,
        end = 0,
        characters = 0,
        words = 0,
        lines = 0;
```

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
Argand,J.R    900823
Bessel,F.W    719731
Gandhi,M.K    362718
Gauss,C.F     806788
Gibbs,J.W     123145
Lagrange,J.L  869245
Leibniz,G.W   711518
Poisson,S.D   853240
Stokes,G.G    545454
Sturm,C.F     222031
```

Fig. 8.12 *Program to alphabetize a customer list*

Review Questions

8.1 State whether the following statements are *true* or *false*

(a) When initializing a string variable during its declaration, we must include the null character as part of the string constant, like "GOOD\0".

(b) The **gets** function automatically appends the null character at the end of the string read from the keyboard.

(c) When reading a string with **scanf**, it automatically inserts the terminating null character.

(d) String variables cannot be used with the assignment operator.

(e) We cannot perform arithmetic operations on character variables.

(f) We can assign a character constant or a character variable to an **int** type variable.

(g) The function **scanf** cannot be used in any way to read a line of text with the white-spaces.

(h) The ASCII character set consists of 128 distinct characters.

(i) In the ASCII collating sequence, the uppercase letters precede lowercase letters.

(j) In C, it is illegal to mix character data with numeric data in arithmetic operations.

(k) The function **getchar** skips white-space during input.

(l) In C, strings cannot be initialized at run time.

(m) The input function **gets** has one string parameter.

(n) The function call **strcpy(s2, s1);** copies string s2 into string s1.

(o) The function call **strcmp("abc", "ABC");** returns a positive number.

8.2 Fill in the blanks in the following statements.

(a) We can use the conversion specification _____in **scanf** to read a line of text.

(b) We can initialize a string using the string manipulation function_____.

(c) The function **strncat** has _____ parameters.

(d) To use the function **atoi** in a program, we must include the header file _____.

(e) The function _____does not require any conversion specification to read a string from the keyboard.

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

8.14 Write a program to create a directory of students with roll numbers. The program should display the roll number for a specified name and vice-versa.

8.15 Given a string

```
char str [ ] = "123456789" ;
```

Write a program that displays the following:

```
    1
  2 3 2
 3 4 5 4 3
4 5 6 7 6 5 4
5 6 7 8 9 8 7 6 5
```

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Except the starting point, there are no other predetermined relationships, rules of precedence, or hierarchies among the functions that make up a complete program. The functions can be placed in any order. A called function can be placed either before or after the calling function. However, it is the usual practice to put all the called functions at the end. See the box "Modular Programming"

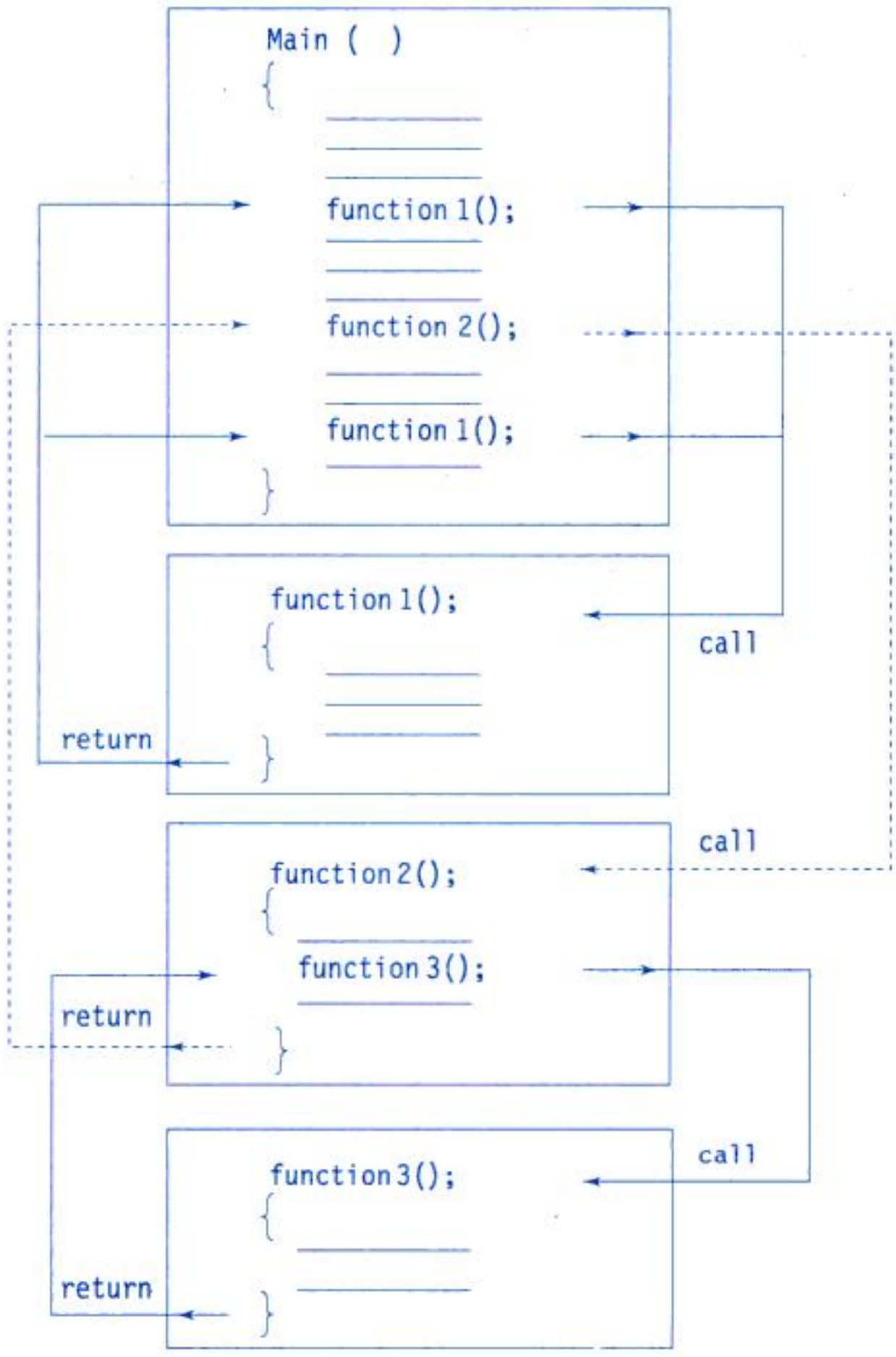


Fig. 9.2 *Flow of control in a multi-function program*

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Some examples of typical function definitions are:

```
(a) float mul (float x, float y)
    {
      float result;          /* local variable */
      result = x * y;        /* computes the product */
      return (result);       /* returns the result */
    }
(b) void sum (int a, int b)
    {
      printf ("sum = %s", a + b); /* no local variables */
      return;                /* optional */
    }
(c) void display (void)
      {                      /* no local variables */
        printf ("No type, no parameters");
                             /* no return statement */
      }
```

NOTE:

1. When a function reaches its return statement, the control is transferred back to the calling program. In the absence of a return statement, the closing brace acts as a *void return*.
2. A *local variable* is a variable that is defined inside a function and used without having any role in the communication between functions.

9.6 RETURN VALUES AND THEIR TYPES

As pointed out earlier, a function may or may not send back any value to the calling function. If it does, it is done through the **return** statement. While it is possible to pass to the called function any number of values, the called function can only return *one value* per call, at the most.

The **return** statement can take one of the following forms:

```
return;
or
return(expression);
```

The first, the 'plain' **return** does not return any value; it acts much as the closing brace of the function. When a **return** is encountered, the control is immediately passed back to the calling function. An example of the use of a simple **return** is as follows:

```
if(error)
return;
```

NOTE: In C99, if a function is specified as returning a value, the **return** must have value associated with it.

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

assumptions are wrong, the linker will fail and we will have to change the program. The moral is that we must always include prototype declarations, preferably in global declaration section.

Parameters Everywhere!

Parameters (also known as arguments) are used in three places:

1. in declaration (prototypes),
2. in function call, and
3. in function definition.

The parameters used in prototypes and function definitions are called *formal parameters* and those used in function calls are called *actual parameters*. Actual parameters used in a calling statement may be simple constants, variables or expressions.

The formal and actual parameters must match exactly in type, order and number. Their names, however, do not need to match.

9.9 CATEGORY OF FUNCTIONS

A function, depending on whether arguments are present or not and whether a value is returned or not, may belong to one of the following categories:

Category 1:	Functions with no arguments and no return values.
Category 2:	Functions with arguments and no return values.
Category 3:	Functions with arguments and one return value.
Category 4:	Functions with no arguments but return a value.
Category 5:	Functions that return multiple values.

In the sections to follow, we shall discuss these categories with examples. Note that, from now on, we shall use the term arguments (rather than parameters) more frequently:

9.10 NO ARGUMENTS AND NO RETURN VALUES

When a function has no arguments, it does not receive any data from the calling function. Similarly, when it does not return a value, the calling function does not receive any data from the called function. In effect, there is no data transfer between the calling function and the called function. This is depicted in Fig. 9.3. The dotted lines indicate that there is only a transfer of control but not data.

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

operations. For example, different programs may require different output formats for display of results. These shortcomings can be overcome by handing over the result of a function to its calling function where the returned value can be used as required by the program.

A self-contained and independent function should behave like a 'black box' that receives a predefined form of input and outputs a desired value. Such functions will have two-way data communication as shown in Fig. 9.8.

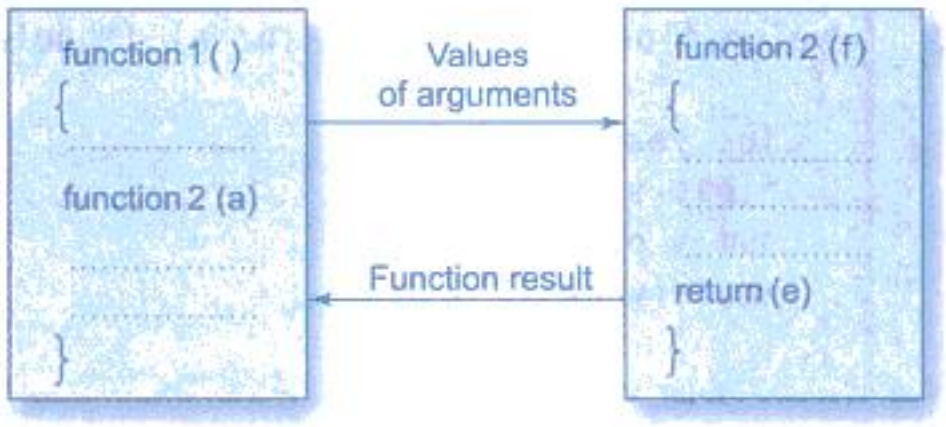


Fig. 9.8 *Two-way data communication between functions*

We shall modify the program in Fig. 9.7 to illustrate the use of two-way data communication between the *calling* and the *called functions*.

Example 9.3 In the program presented in Fig. 9.7 modify the function **value**, to return the final amount calculated to the **main**, which will display the required output at the terminal. Also extend the versatility of the function **printline** by having it to take the length of the line as an argument.

The modified program with the proposed changes is presented in Fig. 9.9. One major change is the movement of the **printf** statement from **value** to **main.**

Program

```
void printline (char ch, int len);
value (float, float, int);

main( )
{
  float principal, inrate, amount;
  int period;
  printf("Enter principal amount, interest");
  printf("rate, and period\n");
  scanf(%f %f %d", &principal, &inrate, &period);
  printline ('*' , 52);
  amount = value (principal, inrate, period);
  printf("\n%f\t%f\t%d\t%f\n\n",principal,
    inrate,period,amount);
  printline('=',52);
}
```

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

When dealing with array arguments, we should remember one major distinction. If a function changes the values of the elements of an array, then these changes will be made to the original array that passed to the function. When an entire array is passed as an argument, the contents of the array are not copied into the formal parameter array; instead, information about the addresses of array elements are passed on to the function. Therefore, any changes introduced to the array elements are truly reflected in the original array in the calling function. However, this does not apply when an individual element is passed on as argument. Example 9.6 highlights these concepts.

Example 9.6 Write a program that uses a function to sort an array of integers.

A program to sort an array of integers using the function **sort()** is given in Fig. 9.12. Its output clearly shows that a function can change the values in an array passed as an argument.

Program

```
void sort(int m, int x[ ]);
main()
{
    int i;
    int marks[5] = {40, 90, 73, 81, 35};

    printf("Marks before sorting\n");
    for(i = 0; i < 5; i++)
       printf("%d ", marks[i]);
    printf("\n\n");

    sort (5, marks);

    printf("Marks after sorting\n");
    for(i = 0; i < 5; i++)
       printf("%4d", marks[i]);
    printf("\n");
}

void sort(int m, int x[ ])

{
    int i, j, t;

    for(i = 1; i <= m-1; i++)
       for(j = 1; j <= m-i; j++)
          if(x[j-1] >= x[j])
          {
             t = x[j-1];
             x[j-1] = x[j];
```

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
main( )
{
     int number;
     -----
     -----
}
```

We may also use the keyword **auto** to declare automatic variables explicitly.

```
main( )
{
     auto int number;
     -----
     -----
}
```

One important feature of automatic variables is that their value cannot be changed accidentally by what happens in some other function in the program. This assures that we may declare and use the same variable name in different functions in the same program without causing any confusion to the compiler.

Example 9.7 Write a multifunction to illustrate how automatic variables work.

A program with two subprograms **function1** and **function2** is shown in Fig. 9.13. **m** is an automatic variable and it is declared at the beginning of each function. **m** is initialized to 10, 100, and 1000 in function1, function2, and **main** respectively.

When executed, **main** calls **function2** which in turn calls **function1**. When **main** is active, m = 1000; but when **function2** is called, the **main**'s **m** is temporarily put on the shelf and the new local **m** = 100 becomes active. Similarly, when **function1** is called, both the previous values of **m** are put on the shelf and the latest value of **m** (=10) becomes active. As soon as **function1** (m=10) is finished, **function2** (m=100) takes over again. As soon it is done, **main** (m=1000) takes over. The output clearly shows that the value assigned to **m** in one function does not affect its value in the other functions; and the local value of **m** is destroyed when it leaves a function.

Program

```
void function1(void);
void function2(void);
main( )
{
     int m = 1000;
     function2();

     printf("%d\n",m); /* Third output */
}
 void function1(void)
{
```

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Function declarations outside of any function behave the same way as variable declarations.

Static Variables

As the name suggests, the value of static variables persists until the end of the program. A variable can be declared *static* using the keyword **static** like

```
static int x;
static float y;
```

A static variable may be either an internal type or an external type depending on the place of declaration.

Internal static variables are those which are declared inside a function. The scope of internal static variables extend up to the end of the function in which they are defined. Therefore, internal **static** variables are similar to **auto** variables, except that they remain in existence (alive) throughout the remainder of the program. Therefore, internal **static** variables can be used to retain values between function calls. For example, it can be used to count the number of calls made to a function.

Example 9.9 Write a program to illustrate the properties of a static variable.

The program in Fig. 9.15 explains the behaviour of a static variable.

Program

```
void stat(void);
main ( )
{
   int i;
   for(i=1; i<=3; i++)
   stat( );
}
void stat(void)
{
   static int x = 0;

   x = x+1;
   printf("x = %d\n", x);
}
```

Output

```
x = 1
x = 2
x = 3
```

Fig. 9.15 *Illustration of static variable*

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Program

```
#include <stdio.h>
float  start_point,                /* GLOBAL VARIABLES */
       end_point,
       total_area;
int    numtraps;
main( )
{
     void   input(void);
     float  find_area(float a,float b,int n); /* prototype */

     print("AREA UNDER A CURVE");
     input( );
     total_area = find_area(start_point, end_point, numtraps);
     printf("TOTAL AREA = %f", total_area);
}
void input(void)
{
     printf("\n Enter lower limit:");
     scanf("%f", &start_point);
     printf("Enter upper limit:");
     scanf("%f", &end_point);
     printf("Enter number of trapezoids:");
     scanf("%d", &numtraps);
}
float find_area(float a, float b, int n)
{
     float base, lower, h1, h2;  /* LOCAL VARIABLES */
     float function_x(float x);  /* prototype */
     float trap_area(float h1,float h2,float base);/*prototype*/

     base = (b-1)/n;
     lower = a;

      for(lower =a; lower <= b-base; lower = lower + base)
     {
          h1   = function_x(lower);
          h1   = function_x(lower + base);
          total_area += trap_area(h1, h2, base);
     }
          return(total_area);
float trap_area(float height_1,float height_2,float base)
{
  float area;      /* LOCAL VARIABLE */
  area = 0.5 * (height_1 + height_2) * base;
  return(area);
}
```

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
        return (x / y);
    }
```

What will be the value of the following function calls"

(a) divide (10, 2)
(b) divide (9, 2)
(c) divide (4.5, 1.5)
(d) divide (2.0, 3.0)

9.13 What will be the effect on the above function calls if we change the header line as follows:

(a) int divide (int x, int y)
(b) double divide (float x, float y)

9.14 Determine the output of the following program?

```
int prod( int m, int n);
main ( )
{
    int x = 10;
    int y = 20;
    int p, q;
    p = prod (x,y);
    q = prod (p, prod (x,z));
    printf ("%d %d\n", p,q);
}
int prod( int a, int b)
{
    return (a * b);
}
```

9.15 What will be the output of the following program?

```
void test (int *a);
main ( )
{
        int x = 50;
        test ( &x);
        printf("%d\n", x);
}
void test (int *a);
{
        *a = *a + 50;
}
```

9.16 The function **test** is coded as follows:

```
int test (int number)
{
        int m, n = 0;
        while (number)
        {
                m = number % 10;
                if (m % 2)
                  n = n + 1;
```

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

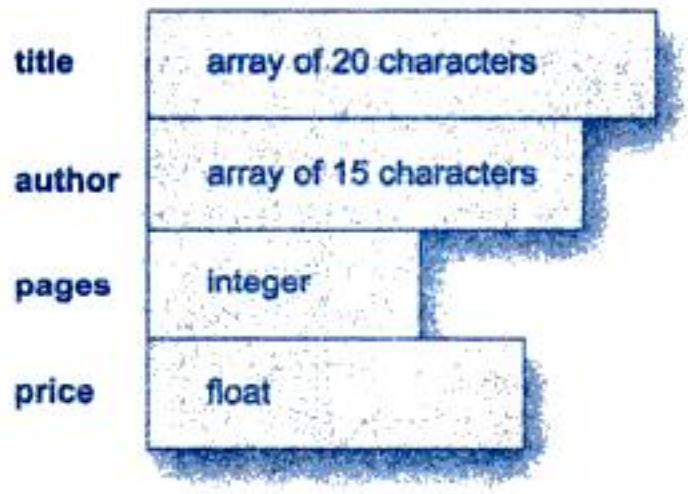
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

book name, author, number of pages, and price. We can define a structure to hold this information as follows:

```
struct book_bank
{
    char      title[20];
    char      author[15];
    int       pages;
    float     price;
};
```

The keyword **struct** declares a structure to hold the details of four data fields, namely **title, author, pages**, and **price**. These fields are called *structure elements* or *members*. Each member may belong to a different type of data. **book_bank** is the name of the structure and is called the *structure tag*. The tag name may be used subsequently to declare variables that have the tag's structure.

Note that the above definition has not declared any variables. It simply describes a format called *template* to represent information as shown below:



The general format of a structure definition is as follows:

```
struct        tag_name
{
    data_type                 member1;
    data_type                 member2;
        ----                  ----
        ----                  ----
};
```

In defining a structure you may note the following syntax:

1. The template is terminated with a semicolon.
2. While the entire definition is considered as a statement, each member is declared independently for its name and type in a separate statement inside the template.
3. The tag name such as **book_bank** can be used to declare structure variables of its type, later in the program.

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
        scanf("%s %d %s %d %f",
                 person.name,
                &person.day,
                 person.month,
                &person.year,
                &person.salary);
        printf("%s %d %s %d %f\n",
                 person.name,
                 person.day,
                 person.month,
                 person.year,
                 person.salary);
    }
```

Output

```
Input Values
M.L.Goel 10 January 1945 4500
M.L.Goel 10 January 1945 4500.00
```

Fig. 10.1 *Defining and accessing structure members*

10.5 STRUCTURE INITIALIZATION

Like any other data type, a structure variable can be initialized at compile time.

```
main()
{
    struct
    {
       int weight;
       float height;
    }
    student = {60, 180.75};
    .....
    .....
}
```

This assigns the value 60 to **student. weight** and 180.75 to **student. height.** There is a one-to-one correspondence between the members and their initializing values.

A lot of variation is possible in initializing a structure. The following statements initialize two structure variables. Here, it is essential to use a tag name.

```
main()
{
   struct st_record
   {
```

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Programs

```
#include   <stdio.h>
#include   <string.h>
struct  record
{
    char    author[20];
    char    title[30];
    float   price;
    struct

    {
        char   month[10];
        int    year;
    }
    date;
    char   publisher[10];
    int    quantity;
};
 int look_up(struct record table[],char s1[],char s2[],int m);
 void get (char string [ ] );
 main()
 {
      char title[30], author[20];
      int  index, no_of_records;
      char response[10], quantity[10];
      struct record book[] =  {
      {"Ritche","C Language",45.00,"May",1977,"PHI",10},
      {"Kochan","Programming in C",75.50,"July",1983,"Hayden",5},
      {"Balagurusamy","BASIC",30.00,"January",1984,"TMH",0},
      {"Balagurusamy","COBOL",60.00,"December",1988,"Macmillan",25}
                                  };

no_of_records = sizeof(book)/ sizeof(struct record);
 do
 {
   printf("Enter title and author name as per the list\n");
   printf("\nTitle:    ");
   get(title);
   printf("Author:   ");
   get(author);
   index = look_up(book, title, author, no_of_records);
   if(index != -1)     /*  Book found  */
   {
       printf("\n%s %s %.2f %s %d %s\n\n",
              book[index].author,
              book[index].title,
```

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

- Remember that the definition for a pointer variable allocates memory only for the pointer variable, not for the variable to which it is pointing.
- If we want a called function to change the value of a variable in the calling function, we must pass the address of that variable to the called function.
- When we pass a parameter by address, the corresponding formal parameter must be a pointer variable.
- It is an error to assign a numeric constant to a pointer variable.
- It is an error to assign the address of a variable to a variable of any basic data types.
- It is an error to assign a pointer of one type to a pointer of another type without a cast (with an exception of void pointer).
- A proper understanding of a precedence and associativity rules is very important in pointer applications. For example, expressions like *p++, *p[ ], (*p)[ ], (p).member should be carefully used.
- When an array is passed as an argument to a function, a pointer is actually passed. In the header function, we must declare such arrays with proper size, except the first, which is optional.
- A very common error is to use (or not to use) the address operator (&) and the indirection operator (*) in certain places. Be careful. The compiler may not warn such mistakes.

Case Studies

1. Processing of Examination Marks

Marks obtained by a batch of students in the Annual Examination are tabulated as follows:

Student name	Marks obtained
S. Laxmi	45 67 38 55
V.S. Rao	77 89 56 69
-	- - - -

It is required to compute the total marks obtained by each student and print the rank list based on the total marks.

The program in Fig. 11.14 stores the student names in the array **name** and the marks in the array **marks.** After computing the total marks obtained by all the students, the program prepares and prints the rank list. The declaration

```
int marks[STUDENTS][SUBJECTS+1];
```

defines **marks** as a pointer to the array's first row. We use **rowptr** as the pointer to the row of **marks.** The **rowptr** is initialized as follows:

```
int (*rowptr)[SUBJECTS+1] = array;
```

Note that **array** is the formal argument whose values are replaced by the values of the actual argument **marks.** The parentheses around ***rowptr** makes the **rowptr** as a pointer to an array of **SUBJECTS+1** integers. Remember, the statement

```
int *rowptr[SUBJECTS+1];
```

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Programming Exercises

11.1 Write a program using pointers to read in an array of integers and print its elements in reverse order.

11.2 We know that the roots of a quadratic equation of the form

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

are given by the following equations:

$$x_1 = \frac{-b + \text{square-root}\,(b^2 - 4ac)}{2a}$$

$$x_2 = \frac{-b - \text{square-root}\,(b^2 - 4ac)}{2a}$$

Write a function to calculate the roots. The function must use two pointer parameters, one to receive the coefficients a, b, and c, and the other to send the roots to the calling function.

11.3 Write a function that receives a sorted array of integers and an integer value, and inserts the value in its correct place.

11.4 Write a function using pointers to add two matrices and to return the resultant matrix to the calling function.

11.5 Using pointers, write a function that receives a character string and a character as argument and deletes all occurrences of this character in the string. The function should return the corrected string with no holes.

11.6 Write a function **day_name** that receives a number n and returns a pointer to a character string containing the name of the corresponding day. The day names should be kept in a **static** table of character strings local to the function.

11.7 Write a program to read in an array of names and to sort them in alphabetical order. Use **sort** function that receives pointers to the functions **strcmp** and **swap.sort** in turn should call these functions via the pointers.

11.8 Given an array of sorted list of integer numbers, write a function to search for a particular item, using the method of *binary search*. And also show how this function may be used in a program. Use pointers and pointer arithmetic.
(Hint: In binary search, the target value is compared with the array's middle element. Since the table is sorted, if the required value is smaller, we know that all values greater than the middle element can be ignored. That is, in one attempt, we eliminate one half the list. This search can be applied recursively till the target value is found.)

11.9 Write a function (using a pointer parameter) that reverses the elements of a given array.

11.10 Write a function (using pointer parameters) that compares two integer arrays to see whether they are identical. The function returns 1 if they are identical, 0 otherwise.

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Program

```
#include  <stdio.h>

main()
{
    FILE *f1;
    char c;
    printf("Data Input\n\n");
    /* Open the file INPUT */
    f1 = fopen("INPUT", "w");

    /* Get a character from keyboard   */
    while((c=getchar()) != EOF)

        /* Write a character to INPUT  */
        putc(c,f1);

    /* Close the file INPUT   */

    fclose(f1);
    printf("\nData Output\n\n");

    /* Reopen the file INPUT    */
    f1 = fopen("INPUT","r");

   /* Read a character from INPUT*/
    while((c=getc(f1)) != EOF)

        /* Display a character on screen */
        printf("%c",c);

    /* Close the file INPUT       */
    fclose(f1);
}
```

Output

```
Data Input
This is a program to test the file handling
features on this system^Z

Data Output
This is a program to test the file handling
features on this system
```

Fig. 12.1 *Character oriented read/write operations on a file*

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

ftell takes a file pointer and return a number of type **long,** that corresponds to the current position. This function is useful in saving the current position of a file, which can be used later in the program. It takes the following form:

```
n = ftell(fp);
```

n would give the relative offset (in bytes) of the current position. This means that **n** bytes have already been read (or written).

rewind takes a file pointer and resets the position to the start of the file. For example, the statement

```
rewind(fp);
n = ftell(fp);
```

would assign **0** to **n** because the file position has been set to the start of the file by **rewind.** Remember, the first byte in the file is numbered as 0, second as 1, and so on. This function helps us in reading a file more than once, without having to close and open the file. Remember that whenever a file is opened for reading or writing, a **rewind** is done implicitly.

fseek function is used to move the file position to a desired location within the file. It takes the following form:

```
fseek(file_ptr, offset, position);
```

file_ptr is a pointer to the file concerned, *offset* is a number or variable of type long, and *position* is an integer number. The *offset* specifies the number of positions (bytes) to be moved from the location specified by *position.* The *position* can take one of the following three values:

Value	*Meaning*
0	Beginning of file
1	Current position
2	End of file

The offset may be positive, meaning move forwards, or negative, meaning move backwards.

Examples in Table 12.2 illustrate the operations of the **fseek** function:

Table 12.2 *Operations of fseek Function*

Statement	*Meaning*
fseek(fp,0L,0);	Go to the beginning. (Similar to rewind)
fseek(fp,0L,1);	Stay at the current position. (Rarely used)
fseek(fp,0L,2);	Go to the end of the file, past the last character of the file.
fseek(fp,m,0);	Move to (m+1)th byte in the file.
fseek(fp,m,1);	Go forward by m bytes.
fseek(fp,-m,1);	Go backward by m bytes from the current position.
fseek(fp,-m,2);	Go backward by m bytes from the end. (Positions the file to the mth character from the end.)

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

When we no longer need the data we stored in a block of memory, and we do not intend to use that block for storing any other information, we may release that block of memory for future use, using the **free** function:

```
free (ptr);
```

ptr is a pointer to a memory block, which has already been created by **malloc** or **calloc**. Use of an invalid pointer in the call may create problems and cause system crash. We should remember two things here:

1. It is not the pointer that is being released but rather what it points to.
2. To release an array of memory that was allocated by **calloc** we need only to release the pointer once. It is an error to attempt to release elements individually.

The use of **free** function has been illustrated in Example 13.2.

13.6 ALTERING THE SIZE OF A BLOCK: REALLOC

It is likely that we discover later, the previously allocated memory is not sufficient and we need additional space for more elements. It is also possible that the memory allocated is much larger than necessary and we want to reduce it. In both the cases, we can change the memory size already allocated with the help of the function **realloc**. This process is called the *reallocation* of memory. For example, if the original allocation is done by the statement

```
ptr = malloc(size);
```

then reallocation of space may be done by the statement

```
ptr = realloc(ptr, newsize);
```

This function allocates a new memory space of size *newsize* to the pointer variable **ptr** and returns a pointer to the first byte of the new memory block. The *newsize* may be larger or smaller than the *size*. Remember, the new memory block may or may not begin at the same place as the old one. In case, it is not able to find additional space in the same region, it will create the same in an entirely new region and move the contents of the old block into the new block. The function guarantees that the old data will remain intact.

If the function is unsuccessful in locating additional space, it returns a NULL pointer and the original block is freed (lost). This implies that it is necessary to test the success of operation before proceeding further. This is illustrated in the program of Example 13.2.

Example 13.2 Write a program to store a character string in a block of memory space created by **malloc** and then modify the same to store a larger string.

The program is shown in Fig. 13.3. The output illustrates that the original buffer size obtained is modified to contain a larger string. Note that the original contents of the buffer remains same even after modification of the original size.

Program

```
#include <stdio.h>
#include<stdlib.h>
#define NULL 0
main()
```

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
new->number = x;
new->next = head;
head = new;
```

```
node *insert(node *head)
{
        node *find(node *p, int a);
        node *new;    /* pointer to new node */
        node *n1;   /* pointer to node preceding key node */
        int key;
        int x;      /* new item (number) to be inserted */

        printf("Value of new item?");
        scanf("%d", &x);
        printf("Value of key item ? (type -999 if last) ");
        scanf("%d", &key);

        if(head->number == key)  /* new node is first */
        {
            new = (node *)malloc(size of(node));
            new->number = x;
            new->next = head;
            head = new;
        }
        else   /* find key node and insert new node */
        {      /* before the key node */
          n1 = find(head, key);    /* find key node */

          if(n1 == NULL)
             printf("\n key is not found \n");
          else   /* insert new node */
          {
              new = (node *)malloc(sizeof(node));
              new->number = x;
              new->next = n1->next;
              n1->next = new;
          }
        }
        return(head);
}
node *find(node *lists, int key)
{
          if(list->next->number == key) /* key found */
            return(list);
        else
```

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
void print(node *list)
{
        if(list->next != NULL)
        {
          printf("%d -->", list->number);

        if(list ->next->next = = NULL)
          printf("%d", list->next->number);

        print(list->next);
        }
        return:
}
node *insert(node *head, int x)
{
        node *p1, *p2, *p;
        p1 = NULL;
        p2 = head;  /* p2 points to first node */

        for( ; p2->number < x; p2 = p2->next)
        {
          p1 = p2;

          if(p2->next->next == NULL)
            {
              p2 = p2->next;   /* insertion at end */
              break;
            }
        }

        /*key node found and insert new node */

          p = (node )malloc(sizeof(node));  / space for new node */

          p->number = x; /* place value in the new node */

          p->next = p2; /*link new node to key node */

          if (p1 == NULL)
            head = p; /* new node becomes the first node */
          else
            p1->next = p; /* new node inserted in middle */

          return (head);
}
```

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

14

The Preprocessor

14.1 INTRODUCTION

C is a unique language in many respects. We have already seen features such as structures and pointers. Yet another unique feature of the C language is the *preprocessor*. The C preprocessor provides several tools that are unavailable in other high-level languages. The programmer can use these tools to make his program easy to read, easy to modify, portable, and more efficient.

The preprocessor, as its name implies, is a program that processes the source code before it passes through the compiler. It operates under the control of what is known as *preprocessor command lines or directives*. Preprocessor directives are placed in the source program before the main line. Before the source code passes through the compiler, it is examined by the preprocessor for any preprocessor directives. If there are any, appropriate actions (as per the directives) are taken and then the source program is handed over to the compiler.

Preprocessor directives follow special syntax rules that are different from the normal C syntax. They all begin with the symbol # in column one and do not require a semicolon at the end. We have already used the directives **#define** and **#include** to a limited extent. A set of commonly used preprocessor directives and their functions is given in Table 14.1.

Table 14.1 *Preprocessor Directives*

Directive	*Function*
#define	Defines a macro substitution
#undef	Undefines a macro
#include	Specifies the files to be included
#ifdef	Test for a macro definition
#endif	Specifies the end of #if.
#ifndef	Tests whether a macro is not defined.
#if	Test a compile-time condition
#else	Specifies alternatives when #if test fails.

These directives can be divided into three categories:

1. Macro substitution directives.
2. File inclusion directives.
3. Compiler control directives.

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
  char tmp[8];
  float tst;
  _setcursortype(_NORMALCURSOR);
  print2screen(3,33,"Enter Item Code: ",BROWN,BLUE,0);fflush(stdin);gotopos(3,53);
  scanf("%s",&tmp);
  if(CheckId(tmp)==0 && fEdit == FALSE)
  {
    messagebox(10,33,"The id already exists. ","Error ",'
               ',mboxbrdrclr,mboxbgclr,mboxfgclr,0);
    return 0;
  }
  strcpy(inv_stock.itemcode,tmp);   /*Means got a correct item code*/
  print2screen(4,33,"Name of the Item: ",BROWN,BLUE,0);fflush(stdin);gotopos(4,53);
  gets(inv_stock.itemname);
  print2screen(5,33,"Price of Each Unit: ",BROWN,BLUE,0);fflush(stdin);gotopos(5,53);
  scanf("%f",&inv_stock.itemrate);
  print2screen(6,33,"Quantity:  ",BROWN,BLUE,0);fflush(stdin);gotopos(6,53);
  scanf("%f",&inv_stock.itemqty);
  print2screen(7,33,"Reorder Level: ",BROWN,BLUE,0);fflush(stdin);gotopos(7,53);
  scanf("%d",&inv_stock.minqty);
  _setcursortype(_NOCURSOR);
  return 1;
}

/*Returns 0 if the id already exists in the database, else returns 1*/
int CheckId(char item[8])
{
  rewind(dbfp);
  while(fread(&inv_stock,stocksize,1,dbfp)==1)
    if(strcmp(inv_stock.itemcode,item)==0)
      return(0);
  return(1);
}

/*Displays an Item*/
DisplayItemRecord(char idno[8])
{
 rewind(dbfp);
 while(fread(&inv_stock,stocksize,1,dbfp)==1)
   if(strcmp(idno,inv_stock.itemcode)==0)
     DisplayItemInfo();
 return;
}

/*Displays an Item information*/
DisplayItemInfo()
{
  int r=7;
  textcolor(menutxtfgclr);
  textbackground(menutxtbgclr);
```

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

PROGRAMMING IN ANSI

The hallmark of this classic text is its simple and lucid presentation of the C programming concepts. The concept of 'learning by example' used in the book helps the beginners in better understanding the implementation and applications of the C language. This is achieved through sample programs, case studies, programming problems and projects.

Salient features

- Codes with comments provided throughout the book illustrate how the various features of the language are put together to accomplish specified tasks
- Case studies at the end of the chapters illustrate common ways C features are put together and show real-life applications
- 'Just Remember' section at the end of the chapters lists helpful hints and possible problem areas
- Guidelines for developing efficient C programs given in the last chapter, together with a list of some common mistakes that a less experienced C programmer could make
- Programming Projects discussed in the appendix show how to integrate the various features of C when handling large programs
- A CD along with the book provides all the programs in an executable format along with two programming projects: "Inventory" and "Record Entry"

URL: http://www.mhhe.com/balagurusamy/ansic5e

The McGraw·Hill Companies

Mc Graw Hill Higher Education

www.tatamcgrawhill.com

ISBN-13:978-0-07-068182-8
ISBN-10:0-07-068182-1

9 780070 681828

Copyrighted material