A TEXTBOOK OF SMALL ‘C’ PROGRAMS

by

Dr. Ravi P. Reddy, Ph.D.

NATIONAL INSTITUTE OF SCIENCE & TECHNOLOGY
PALUR HILLS, BERHAMPUR, ORISSA - 761 008, INDIA
Welcome to the exciting world of C programming! Hope you will enjoy using this beginner's textbook as you learn the nuances of the C language and begin to appreciate the power of this graceful programming language.

This book is intended for beginning and intermediate level programmers. No textbook can seriously claim to take you beyond that because the only way to get to a “masters” level is by practicing the skills learnt in textbooks by applying them to various real world problems.

This book evolved out of my lecture notes for a second semester course entitled "Programming in C" which I taught at our institute. There were several distinct characteristics of the students whom I was teaching.

- They were all students of engineering and, hence, they all had a solid mathematical background,
- They had all undergone a first course entitled "Fundamentals of Computing" which gave them a reasonably good understanding of the basic principles involved in the working of computers and the organization of the major hardware components of a computer.,
- They had had no formal course on computer algorithms except a brief introduction to flow charts in their first course.

The short length of each lesson in the book is intended to make it possible for the reader to focus on one small idea at a time. It also makes it possible to for a teacher to cover a lesson in about 2 to 4 hours of classroom time. Depending on the nature of the course being taught, an instructor may want to experiment with different sequences for these chapters but they are best handled in a sequential fashion.

I have tried to mention several features of the ANSI standard at various points in the book. For a more succinct and authoritative explanation, the reader is referred to Kernighan and Ritchie’s classic textbook, ”The C Programming Language”, 2nd edition. I will refer to this book simply as K & R throughout this book.
I have taken a conscious effort to integrate various stylistic conventions of C programs into the various chapters. I believe that it is best to learn a good programming style right at the beginning. Otherwise, it becomes difficult to break out of established set of bad programming habits.

All the programs were compiled and run on Borland C compiler v.2.0 running on MS-DOS v6.22. My practical experience has been Borland's compilers are perhaps the most popular C compilers used in India. At some places, I have mentioned how C programs run differently on UNIX platforms. This was based on my students' experience with running programs from K & R which have a distinct UNIX flavor. For example, a common problem arises when the program tries to check for EOF in interactive input. This works fine on a UNIX system but a MS-DOS based system simply hangs up!

I am reasonably confident that these programs should work on other compilers as well as I have consciously stuck to the ANSI standard libraries and conventions. However, I give no guarantees to this effect.

This book is meant for students who wish to learn through writing C programs and not as a leisure reading book. This book is definitely NOT meant to be a read like a novel over a cup of tea. It is meant to be a textbook cum workbook where the emphasis is on hands-on learning by executing the various programs given in the book. So, your first task is to get hold of a C compiler and

Suggestions for improvement are welcome from all users of this textbook! Have fun!
# TABLE OF CONTENTS

## GETTING STARTED
- **1.0** Lesson Goals
- **1.1** Compilers and Interpreters
- **1.2** The HELLO.C Program
- **1.3** Comments in C Programs
- **1.4** Bugs & Programs
- **1.5** CIRCLE.C - A Small C Program
- **1.6** Use of Whitespace in C
- **1.7** Constants in C
- **1.7** Points to Remember

## INTEGERS
- **2.0** Lesson Goals
- **2.1** Storage Requirements for an Integer
- **2.2** Integer Data Types in C
- **2.3** The LIMITS.H File
- **2.4** Integer Constants in C
- **2.5** Simple Input/Output of Integers
- **2.7** Points to Remember

## REAL NUMBERS
- **3.0** Lesson Goals
- **3.1** Storage of Real Numbers
- **3.2** Binary Storage of Real Numbers
- **3.3** Floating Point Data Types
- **3.4** The FLOAT.H File
- **3.5** Floating Point Constants in C
- **3.6** Simple Input/Output of Real Numbers
- **3.7** Points to Remember

## CHARACTERS
- **4.0** Lesson Goals
- **4.1** The ASCII Character Set
- **4.2** The IBM Character Set or Extended ASCII Code
- **4.3** Characters in ANSI C
- **4.4** Simple Input/Output of Characters
- **4.5** Points to Remember

## STRINGS
- **5.0** Lesson Goals
- **5.1** Strings in C
- **5.2** Declaring Character Arrays
- **5.3** Simple Input/Output of Strings
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.8</td>
<td>Mini-Problem 9 - Range of a Projectile</td>
</tr>
<tr>
<td>10.9</td>
<td>Mini-Problem 10 - Cartesian to Polar Transformation</td>
</tr>
<tr>
<td>10.9</td>
<td>Points to Remember</td>
</tr>
<tr>
<td>11.0</td>
<td>Lesson Goals</td>
</tr>
<tr>
<td>11.1</td>
<td>The printf Function</td>
</tr>
<tr>
<td>11.2</td>
<td>Fine Tuning the printf Function</td>
</tr>
<tr>
<td>11.3</td>
<td>The scanf Function</td>
</tr>
<tr>
<td>11.4</td>
<td>Assignment Suppression</td>
</tr>
<tr>
<td>11.5</td>
<td>Points to Remember</td>
</tr>
<tr>
<td>11.6</td>
<td>Lesson Goals</td>
</tr>
<tr>
<td>12.1</td>
<td>Decision Criteria</td>
</tr>
<tr>
<td>12.2</td>
<td>Relational Operators</td>
</tr>
<tr>
<td>12.3</td>
<td>Logical Operators</td>
</tr>
<tr>
<td>12.4</td>
<td>Some Boolean Identities</td>
</tr>
<tr>
<td>12.5</td>
<td>Storage of Boolean Values in C</td>
</tr>
<tr>
<td>12.6</td>
<td>Evaluation of Logical Expressions</td>
</tr>
<tr>
<td>12.7</td>
<td>Character Classification Functions</td>
</tr>
<tr>
<td>12.8</td>
<td>Comparing Strings</td>
</tr>
<tr>
<td>12.9</td>
<td>Conditional Operator</td>
</tr>
<tr>
<td>12.10</td>
<td>Points to Remember</td>
</tr>
<tr>
<td>13.0</td>
<td>Lesson Goals</td>
</tr>
<tr>
<td>13.1</td>
<td>Operator Precedence</td>
</tr>
<tr>
<td>13.2</td>
<td>The Comma Operator</td>
</tr>
<tr>
<td>13.3</td>
<td>Evaluation Sequence of Operands</td>
</tr>
<tr>
<td>13.4</td>
<td>Points to Remember</td>
</tr>
<tr>
<td>14.0</td>
<td>Lesson Goals</td>
</tr>
<tr>
<td>14.1</td>
<td>Control Flow</td>
</tr>
<tr>
<td>14.2</td>
<td>Statements and Blocks</td>
</tr>
<tr>
<td>14.3</td>
<td>If-Else Statement</td>
</tr>
<tr>
<td>14.4</td>
<td>Nested If-Else Statements</td>
</tr>
<tr>
<td>14.5</td>
<td>Switch Statement</td>
</tr>
<tr>
<td>14.6</td>
<td>Program Indentation</td>
</tr>
<tr>
<td>14.7</td>
<td>goto Statement</td>
</tr>
<tr>
<td>14.8</td>
<td>Points to Remember</td>
</tr>
<tr>
<td>15.0</td>
<td>Lesson Goals</td>
</tr>
<tr>
<td>15.1</td>
<td>while Loop</td>
</tr>
<tr>
<td>15.2</td>
<td>do-while Loop</td>
</tr>
<tr>
<td>15.3</td>
<td>for Loop</td>
</tr>
<tr>
<td>15.4</td>
<td>break and continue Statements</td>
</tr>
<tr>
<td>15.5</td>
<td>Null Statements in Loops</td>
</tr>
<tr>
<td>15.6</td>
<td>Infinite Loops</td>
</tr>
<tr>
<td>15.7</td>
<td>Points to Remember</td>
</tr>
</tbody>
</table>
PROGRAMS USING BRANCHING AND LOOPING

16.0 Lesson Goals
16.1 Finding Special Integers
16.2 Computing Square Roots
16.3 Sum of Integers
16.4 Product of Numbers
16.5 Fibonacci Numbers
16.6 Factorial Calculation
16.7 Sides of a Triangle
16.8 Reversing an Integer
16.9 Generating a Pyramid of Digits
16.10 Generating a Pattern of Characters
16.11 Computing Square Root by Iteration
16.12 Iterative Solution of Nonlinear Equations
16.13 A Guessing Game
16.14 Power Series Computation of sine(x)
16.15 Calculation of Equal Instalment Payments
16.16 Computation of GCD of Two Integers
16.17 Calculation of Perfect Numbers
16.18 Calculation of Pythagorean Triplets
16.19 Creating Fractional Approximations for PI
16.20 Points to Remember

FILE I/O OPERATIONS

17.0 Lesson Goals
17.1 Files and Streams
17.2 Opening a File
17.3 Input/Output from Files
17.4 Closing a File
17.5 Predefined Streams
17.6 Flushing of Stream Buffers
17.7 Error Handling for Files
17.8 Mini Problem - Changing to Upper Case
17.9 Mini Problem - Separating Odd and Even Integers
17.10 Mini Problem - Counting Characters, Words & Lines
17.11 Mini Problem – Shift Encoding of Strings
17.12 Points to Remember

FUNCTIONS

18.0 Lesson Goals
18.1 What is a Function?
18.2 Functions in C
18.3 Calling a Function
18.4 Developing Simple Functions
18.5 Function Prototypes
18.6 Call by Value
18.7 Mini Problem - Developing Character Type Functions
18.8 Old Style Functions
18.9 Using Non-ANSI Library Functions
TABLE OF CONTENTS

18.10 Points to Remember

STORAGE CLASSES
19.0 Lesson Goals
19.1 Automatic and External Variables
19.2 Static Variables
19.3 Pseudorandom Numbers
19.4 Points to Remember

RECURSION
20.0 Lesson Goals
20.1 What is Recursion?
20.2 Recursive Calculation of Factorials
20.3 Printing an Integer in Reverse
20.4 Guidelines for Developing Recursive Functions
20.5 Memory Considerations for Recursion
20.6 Printing a String in Reverse
20.7 The Towers of Hanoi
20.8 Generating Permutations
20.9 Alternatives to Recursion
20.10 Points to Remember

ONE DIMENSIONAL ARRAYS
21.0 Lesson Goals
21.1 The Need for Arrays
21.2 Defining Arrays
21.3 Accessing an Array Element
21.4 Initialization of Arrays
21.5 Character Arrays and Strings
21.6 Manipulating Arrays
21.7 Input/Output of Arrays
21.8 Passing Arrays to Functions
21.9 Call by Reference
21.10 Passing Strings to Functions
21.11 Points to Remember

PROGRAMS USING FILES, ARRAYS AND FUNCTIONS
22.0 Lesson Goals
22.1 Insertion Sorting
22.2 Selection Sorting
22.3 Bubble Sorting
22.4 Cyclic Permutation of an Array
22.5 Printing Pascal's Triangle
22.6 Calculating Mean and Standard Deviation
22.7 Fitting a Straight Line to Data
22.8 The Sieve of Eratosthenes
22.9 Computing Prime Numbers
22.10 Computing Factorials of Large Integers
22.11 Numerical Integration using Trapezoidal Rule
22.12 Plotting a Line Graph
22.13 Drawing a Histogram
22.14 Computing Harmonic Mean
22.15 A Hex Dump Program
22.* Points to Remember

POINTERS
23.0 Lesson Goals
23.1 The Address of a Variable
23.2 Pointer Variables
23.3 Pointers to void
23.4 Pointers to Pointers
23.5 Initialization of Pointer Variables
23.6 The Indirection Operator
23.7 Pointers and Functions
23.8 Address Arithmetic
23.9 Pointers and Arrays
23.10 Arguments for scanf
23.11 Legal Operations on Pointers
23.12 Pointers and Strings
23.13 Pointers to Streams
23.14 Dangling References
23.15 Arrays of Pointers
23.16 Pointers to Functions
23.17 Printing Trigonometric Tables
23.18 Numerical Integration
23.* Points to Remember

MULTIDIMENSIONAL ARRAYS
24.0 Lesson Goals
24.1 Two Dimensional Arrays
24.2 Storage of Two Dimensional Arrays
24.3 Pointers and Two Dimensional Arrays
24.4 Multidimensional Arrays
24.5 Passing Multidimensional Arrays to Functions
24.6 Functions for Square Matrices
24.7 Solution of Linear System of Equations
24.8 Points to Remember

DYNAMIC MEMORY ALLOCATION
25.0 Lesson Goals
25.1 Static Memory Allocation
25.2 Dynamic Memory Allocation
25.3 Points to Remember

THE C PREPROCESSOR
26.0 Lesson Goals
26.1 The Preprocessor
26.2 Symbolic Constants
26.3 Macros with Arguments
26.4 Macros versus Functions
26.5 Nested Macros
26.6 Other Preprocessor Directives
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Lesson Goals</th>
<th>Subtopics</th>
</tr>
</thead>
<tbody>
<tr>
<td>26.7</td>
<td>Points to Remember</td>
<td></td>
</tr>
<tr>
<td>27.0</td>
<td>Lesson Goals</td>
<td>27.1 Command Line Arguments</td>
</tr>
<tr>
<td>27.1</td>
<td>Command Line Arguments</td>
<td></td>
</tr>
<tr>
<td>27.2</td>
<td>Sum of Integers</td>
<td>27.3 Copying Files</td>
</tr>
<tr>
<td>27.3</td>
<td>Copying Files</td>
<td>27.4 Points to Remember</td>
</tr>
<tr>
<td>27.4</td>
<td>Points to Remember</td>
<td></td>
</tr>
<tr>
<td>28.0</td>
<td>Lesson Goals</td>
<td>28.1 Modularity</td>
</tr>
<tr>
<td>28.1</td>
<td>Modularity</td>
<td>28.2 Accessing External Variables &amp; Functions</td>
</tr>
<tr>
<td>28.2</td>
<td>Accessing External Variables &amp; Functions</td>
<td>28.3 Static Variables and Functions</td>
</tr>
<tr>
<td>28.3</td>
<td>Static Variables and Functions</td>
<td>28.4 User Defined Header Files</td>
</tr>
<tr>
<td>28.4</td>
<td>User Defined Header Files</td>
<td>28.5 Avoiding Repeated Inclusion of Header Files</td>
</tr>
<tr>
<td>28.5</td>
<td>Avoiding Repeated Inclusion of Header Files</td>
<td>28.6 Managing Multifile Programs using Projects</td>
</tr>
<tr>
<td>28.6</td>
<td>Managing Multifile Programs using Projects</td>
<td>28.7 Location of Files</td>
</tr>
<tr>
<td>28.7</td>
<td>Location of Files</td>
<td>28.8 Points to Remember</td>
</tr>
<tr>
<td>28.8</td>
<td>Points to Remember</td>
<td></td>
</tr>
<tr>
<td>29.0</td>
<td>Lesson Goals</td>
<td>29.1 Typedef</td>
</tr>
<tr>
<td>29.1</td>
<td>Typedef</td>
<td>29.2 Union</td>
</tr>
<tr>
<td>29.2</td>
<td>Union</td>
<td>29.3 Enumerated Constants</td>
</tr>
<tr>
<td>29.3</td>
<td>Enumerated Constants</td>
<td>29.4 Points to Remember</td>
</tr>
<tr>
<td>29.4</td>
<td>Points to Remember</td>
<td></td>
</tr>
<tr>
<td>30.0</td>
<td>Lesson Goals</td>
<td>30.1 Structures</td>
</tr>
<tr>
<td>30.1</td>
<td>Structures</td>
<td>30.2 Declaring Structures</td>
</tr>
<tr>
<td>30.2</td>
<td>Declaring Structures</td>
<td>30.3 Initializing Structures</td>
</tr>
<tr>
<td>30.3</td>
<td>Initializing Structures</td>
<td>30.4 Memory Allocation for Structures</td>
</tr>
<tr>
<td>30.4</td>
<td>Memory Allocation for Structures</td>
<td>30.5 Accessing Data in Structures</td>
</tr>
<tr>
<td>30.5</td>
<td>Accessing Data in Structures</td>
<td>30.6 Arrays of Structures</td>
</tr>
<tr>
<td>30.6</td>
<td>Arrays of Structures</td>
<td>30.7 Structures and Pointers</td>
</tr>
<tr>
<td>30.7</td>
<td>Structures and Pointers</td>
<td>30.8 Passing Structures to Functions</td>
</tr>
<tr>
<td>30.8</td>
<td>Passing Structures to Functions</td>
<td>30.9 Self Referential Structures</td>
</tr>
<tr>
<td>30.9</td>
<td>Self Referential Structures</td>
<td>30.10 Structures and Unions</td>
</tr>
<tr>
<td>30.10</td>
<td>Structures and Unions</td>
<td>30.11 Points to Remember</td>
</tr>
<tr>
<td>30.11</td>
<td>Points to Remember</td>
<td></td>
</tr>
<tr>
<td>31.0</td>
<td>Lesson Goals</td>
<td>31.1 Using Date and Time Functions</td>
</tr>
<tr>
<td>31.1</td>
<td>Using Date and Time Functions</td>
<td>31.2 Measuring Elapsed Time</td>
</tr>
<tr>
<td>31.2</td>
<td>Measuring Elapsed Time</td>
<td>31.3 Points to Remember</td>
</tr>
<tr>
<td>31.3</td>
<td>Points to Remember</td>
<td></td>
</tr>
<tr>
<td>32.0</td>
<td>Lesson Goals</td>
<td>32.1 Sorting using the qsort Function</td>
</tr>
<tr>
<td>32.1</td>
<td>Sorting using the qsort Function</td>
<td>32.2 Searching using bsearch</td>
</tr>
<tr>
<td>32.2</td>
<td>Searching using bsearch</td>
<td>32.3 Sorting and Searching in Structs</td>
</tr>
<tr>
<td>32.3</td>
<td>Sorting and Searching in Structs</td>
<td>32.4 Points to Remember</td>
</tr>
<tr>
<td>32.4</td>
<td>Points to Remember</td>
<td></td>
</tr>
</tbody>
</table>
ADVANCED FILE OPERATIONS
33.0 Lesson Goals
33.1 Direct Read/Write
33.2 Random Access Files
33.3 File Opening Modes
33.4 Points to Remember

TRAPPING AND HANDLING ERRORS
34.0 Lesson Goals
34.1 Input Validation
34.2 Using Non-Local Jumps
34.3 Using the assert Macro
34.4 Points to Remember

BIT LEVEL PROGRAMMING
35.0 Lesson Goals
35.1 The Bitwise Operators
35.2 Bitwise Shifting
35.3 Storage Optimization
35.4 Bit Fields
35.* Points to Remember

PROGRAMMING STYLE
36.0 Lesson Goals
36.1 Good Programming Style
36.2 Readability and Maintainability
36.3 Improving Portability
36.4 Usability
36.5 Efficiency Considerations
36.6 Points to Remember
Getting Started

1.0  Lesson Goals

- Understanding how a C program is written, compiled and executed.
- Learning about the use of comments and whitespace in a C program.
- Gain a broad understanding of the elements of a C program by going through a small program – CIRCLE.C.
- To familiarize yourself with the C programming environment on your computer.

1.1  Welcome to C

Welcome to the wonderful world of C programming! If you are a newcomer to high level programming and want to make C your first language in programming, I hope this book will help you get a thorough understanding of the various aspects of high level programming in C.

If you are moving to C from another language, I hope that this book will help you make a painless transition. I myself have moved to C after a long innings of programming in FORTRAN and a relatively shorter phase of programming Pascal. During these transitions, I enjoyed the fact that the fundamental programming techniques and algorithms do not have to be relearnt. I also remember the painful parts of the transition. In particular, getting used to the call-by-value mechanism for C functions after using FORTRAN’s call-by-reference mechanism for close to a decade was tough! Of course, I also had to learn to handle strange new data types like pointers.

I am sure you will enjoy the process of learning this nimble yet powerful language called C!

1.2  Compilers and Interpreters

In this chapter, we will try to get a preview of many features of the C programming language without getting too deep into the details. But before we start looking at the C language, we need to know a few things about how programs work on a computer.

A computer program written in a **high-level language** consists of commands for the computer in an English-like text. Examples of high level languages are C, C++, BASIC, PASCAL, FORTRAN, ADA, COBOL, etc. High level languages enable us to translate an **algorithm** for solving a problem into a computer program. An algorithm is a set of well defined steps designed to solve a problem and these are usually specified without reference to any particular computer language. We are free to
implement a given algorithm in any computer language of our choice. In this book, we will look at this task of translating algorithms into programs written in the C language.

The instructions written in the high level language are stored in a computer file (in a plain text file format) called the source code file or source file for short. For C language, it is customary to name source files with a .C extension. Before these instructions can be executed by a computer, these instructions must be converted to a low level (machine level) binary language consisting of ones and zeros. Compilers and interpreters represent two strategies for doing this job.

A compiler translates (compiles) the instructions in the source code file into an object code file. This task is done once and only once. The object code file usually has a file extension .OBJ or .O. The object code is written in the machine language of the particular computer on which you want to run your program. This implies that the object code for one machine will obviously not work on a computer having a different type of CPU. However, we can use the same source code on different computers. This portability between computers is the primary reason for using high level languages for writing all our programs. We gain the freedom of moving from one computer to another without a major programming effort.

Very often, a large program is broken up into several small source files in which case we need to generate an object code file for each of the source files individually.

Once the object code is available, the various object code modules are connected together and some additional information required to execute the program is added to the object code. This process is known as linking and the final result is an executable program (usually stored with a file extension .EXE). Only after linking is a program ready for execution or running.

The example shown in Fig.1.1 should clarify the various steps involved in the compilation and linking of a program containing three source files, MYMAIN.C, FOOBAR.C, and MYSUB.C.
Running a program involves loading the executable program into the volatile memory (RAM) of the computer and executing the commands in the program.

In an interpreter, no object code files are created. Every time the program needs to be executed, the source program is translated directly into machine instructions and these instructions are executed. A good example of a C interpreter is the QUINCY interpreter that accompanies [18]. But as a C programmer, more often than not, you will be working with a compiler and not an interpreter.

Now is the time for you to install whatever C development software you have and read its user manual. Look at the instructions on how to edit, compile and run a program.

1.3 The HELLO.C Program

Let us begin by looking at the smallest possible C program.

**Program 1.1 – MINIMAL.C** The minimal do-nothing program

```
main()
{
}
```

This is a perfectly valid program that will "run" (execute) on your computer (that is, once you have figured out how to use the C compiler on your computer!). Unfortunately, you will not see anything because it is a program that does not do anything!

Next, we look at the "Hello World" program immortalized by Kernighan and Ritchie's classic book "The C Programming Language" [7]. Enter the following program into a source file named HELLO.C and execute it on your computer.

**Program 1.2 – HELLO.C** The “Hello World” program (from Kernighan and Ritchie[7])

```
#include <stdio.h>
main()
{
    printf("Hello,world!\n");
}
```

Compile and run the above program on your computer. You may get a warning from the compiler saying "Function must return a value". For the time being, we will ignore the warning and focus on the output of the program which will be displayed on the monitor of your computer.

Hello, world!

Let us examine this program in some detail. **main** is the name of a function. Every C program consists of one or more functions which may have different names but it must have one **main** function. Further, there must be one and only one **main** function in any program. Programs always begin executing from the **main** function.
This calls for a brief explanation of what is a function. A function is a small module of code which performs a specialized task that is repeated very often. Think of a master chef who is preparing a grand feast. He has employed an assistant whose only job is to chop vegetables. The assistant chops onions, chops carrots, chops beans, . . . He does nothing but chop vegetables. This is "a specialized task that is repeated often." Any specialized task that is repeated often is put into a function.

Suppose that the master chef calls this assistant and shouts. "Chop!" The assistant shouts back. "Chop what?" The answer to this question make up the arguments to the function which are placed inside a set of parentheses after the name of the function. So, we could think of a function chop whose argument will be the name of a vegetable which can be used as follows

chop(onions);
chop(beans);

To use another analogy, the name of the function can be seen as a verb indicating the action and the arguments inside the parentheses act like nouns on whom the action takes place.

The function main in HELLO.C has no arguments as indicated by the blank set of parentheses. Next, we observe the body of the main function enclosed in a pair of braces {}. Every function begins with an opening brace '{' and ends with a closing brace '}'. Inside, we find a call to the function printf, i.e., the function main wants the function printf to perform a task. The job of the printf function is to print output which will appear on your computer monitor. What should it print? The arguments are given inside the parentheses following the name printf, in this case, the string of characters "Hello, world!
". Such a string of characters enclosed in double quotes is known as a character string or a string constant.

The newline character at the end of the string constant is a special escape sequence known as the newline character. Escape sequences provide a convenient way of representing various hard-to-type (i.e., difficult to type from a standard computer keyboard) or invisible characters. They are explained in more detail in Chapter 4. The newline character places the text cursor (the small blinking line of box on your computer's monitor is the cursor) at the beginning of the next line. We will learn about other escape sequences for cursor control in Chapter 4.

The next question that arises is "Where do we find the function printf?" It is definitely not present in our own program! The answer lies in the first line of the program

#include <stdio.h>

The symbol # (read "hash") tells the compiler that this line is a "preprocessor directive". This statement tells the preprocessor to include the contents of the header file (the .h extension indicates a header file) named stdio.h. We will learn more about the preprocessor and its functions at a later stage. For now, it suffices to say that this line gives information about the libraries required to execute this program. In particular, it points to the fact that the information about the printf function is to be found in the stdio library (standard input/output library). A library is a collection of functions in object code form stored in a single file. Libraries are usually stored with a special file extension like .LIB or .OLB. Standard libraries eliminate the need for us to write many of the functions needed for our programs by giving us ready-made functions which can do the job. In other
words, we do not have to continuously reinvent wheels. We can use the “wheels” developed by someone else and placed in the standard library for our convenience. Only when we are not satisfied with the performance (speed, accuracy, etc.) of a function from the standard library, or when a function meeting our specifications does not exist, should we write our own user-defined functions.

1.4 Comments in C Programs

Make a copy of the HELLO.C program and name it as HELLO2.C. Make the necessary modifications so that it looks as follows.

Program 1.3 – HELLO2.C An improved “Hello World” program

```c
#include <stdio.h>

void main() /* start of main */
{
    printf("Hello,world!\n");
    printf("Let's learn C!");
}       /* end of main */
```

The output from this program is shown below.

Hello,world!
Let's learn C!

This program demonstrates the use of comments to explain a program in plain English. A comment is any text enclosed between a starting /* and an ending */. A comment can go on for several lines as shown by the first six lines of this program. Comments are totally ignored by the C compiler. Then, why are they necessary? They explain the various facts about a program in plain English for the convenience of the programmer. This helps in ease of use when some one else has to use your program. This also helps you when you look at very old programs of your own making when you see them after a long time. Appropriate comments are part of good programming style because they improve the readability of your programs.

Notice that we have placed the word void in front of main() and this eliminates the warning "Function should return value." We will learn more about the return type of a function later in Chapter 18. For now, we will simply write all main functions as void main().

1.5 Bugs & Programs

A bug is an error in a program which prevents it from fulfilling the desired task of the program.

The simplest kinds of bugs are caused by errors in syntax. A violation of some of the rules of C syntax will usually generate an error message from the compiler. A program having errors will not execute. The compiler also generates a number of warnings which may be ignored, i.e., a program
will execute even when the compiler is giving warnings. However, a good programmer will not overlook these warnings as they often give valuable clues to some possible errors.

Once, the syntax errors have been removed, you have a working program which executes, i.e., you have a program which does something. But very often, like an untrained dog, what the program does and what you want it to do are very different things! This usually implies some fault in the logic of the algorithm or some other serious mistake in the design of the program. Programming experience is the only thing that can help you remove these logical bugs. The difference between the two kinds of errors can be illustrated by the sentences, "The cow has ate mountain" and "The cow ate the mountain". The first sentence is syntactically incorrect as it violates rules of the English grammar. The second sentence is syntactically perfect but poses serious logical problems. Note that the second sentence is illogical only if we know something about cows and mountains, i.e., it requires outside knowledge to determine that the second sentence is illogical. The rules of the English grammar alone will not help us in this task.

Every good compiler has a **symbolic debugger** which is a set of tools to help you to find bugs and fix them. Some of the features of every good symbolic debugger are as follows.

- Facility for executing a program in a line by line fashion.
- Facility for setting breakpoints at selected lines of the program. The program execution stops whenever a breakpoint is encountered and this allows the user to review the intermediate output from the program.
- Facility for setting a watch on some selected variables. This provides for a continuous display of the values of these variables. By tracking how they change, we can often catch the bugs.

### 1.6 CIRCLE.C - A Small C Program

In this section, we will look at a slightly larger C program containing more elements of the C language. The explanation of this program is only meant to give you a quick preview of the major elements of a C program. Just as a movie trailer does not tell the whole story of the movie, this program and its explanation is not meant to teach you C but to show you what a C program looks like. Therefore, do not be worried if you do not understand the details.

The program **CIRCLE.C** given below calculates the radius of a circle as mentioned in the comments at the beginning of the program.

The `# include <stdio.h>` statement is the preprocessor statement which says that we are going to use some functions from the standard I/O library. The next statement

```c
#define PI 3.141592
```

needs some explanation. This is yet another preprocessor statement (Remember that anything beginning with a `#` is a preprocessor statement) defining a **symbolic constant**.
Program 1.4 – CIRCLE.C Program to calculate the area of a circle
(adapted from Gottfried [3])

/ * ================== CIRCLE.C ====================
Progr am to calculate the area of a circle.
=================================================================* /
#include<stdio.h>
#define PI 3.141592

void main() /* start of main */
{
    float radius, area; /* variable declarations */
    float find_area(float); /* function prototype declaration */
    printf("Enter radius >> "); /* output statement */
    scanf("%f", &radius); /* input statement */
    printf("You have entered radius = %f ", radius); /* echo input */
    if(radius < 0) /* input validation */
    {
        /* action on error */
        printf("ERROR: Radius must be non-negative");
    }
    else
    {
        area = find_area(radius); /* call to function find_area*/
        printf("Area of circle with radius of %f = %f", radius, area);
    }
} /* end of main */

/* find_area: function to calculate area of circle given its radius as input */
float find_area(float r) /* start of find_area */
{
    float a;
    a = PI * r * r;
    return(a);
} /* end of find_area */

The preprocessor is a program that takes the C program and modifies it before it is compiled.

The result of preprocessing is a modified C program. For instance, the original CIRCLE.C program contains the statement

\[
a = PI \times r \times r;
\]
This gets modified into the following statement after preprocessing.

\[ a = 3.141592 \times r \times r; \]

What the preprocessor has done is to substitute 3.141592 for every instance of \( \pi \) found anywhere in the program. This is a simple text based operation and does not involve any computation or compilation into machine language. We will look at the preprocessor in more detail in Chapter 25.

Symbolic constants help to improve the readability and maintainability of C programs, both of which are important aspects of good programs.

Next, we have the main function as usual. The first part of any function is the variable declaration part which is followed by the executable statements.

```c
float radius, area; /* variable declaration */
```

The declaration of variables allocates memory space for the storage of various types of values during the execution of the program. The above declaration asks for two variables named `radius` and `area` of the type `float` to be allocated appropriately sized memory space. Use of an undeclared variable generates an error at the time of compilation and the program will not work. All variables must be declared before being used. All declarations must be completed before the first executable statement of the function.

The next statement

```c
float find_area(float); /* function prototype declaration */
```

is a function prototype declaration which tells the program what kind of work can be expected from the function named `find_area`. Let us go back to the example of our chef's assistant. Imagine the master chef giving a command to the assistant "Chop water!" You can imagine the confusion of the assistant because he expects to chop only some kind of a vegetable. A function prototype avoids such erroneous inputs from the master chef by telling him that the assistant chops only vegetables. Here, the main function is being told that the function `find_area` will accept only one argument of type `float` and nothing else.

After this, we call the `printf` and `scanf` functions from the standard I/O library to perform the following I/O operations:

- Prompt the user for input.
- Accept the value of the radius input by the user.
- Echo the value of radius to output to confirm that the correct input has been accepted by the computer. If there are errors (or bugs as they are called) in your program, it is possible that what you have input and what the computer accepts may be different. Therefore, it is good programming practice to echo (i.e., write back to monitor) the input values.

```c
printf("Enter radius >> "); /* output statement */
scanf("%f",&radius); /* input statement*/
printf("You have entered radius = %f ",radius); /* echo input*/
```
The above sequence of operations constitutes what is sometimes known as conversational programming. Next, we perform validation of the input indicating an error if the radius is negative. Input validation (or error trapping) is another important aspect of good programming style because "to err is human, to forgive takes a computer." (Source unknown)

We perform a branching using an if-else statement. If the radius is negative, one set of operations is performed, and if the radius is non-negative, another set of operations is performed. We will go into the details of branching in Chapter 14.

```c
if(radius < 0)
{
    Execute this block if radius < 0.
}
else
{
    Otherwise execute this block.
}
```

The function find_area is called with one argument, radius, and the value returned by the function is stored in the variable area.

Next, we have the user-defined function find_area which has its own variable declaration part. A variable a is declared in this function. The variable r is also a valid variable for the function as it forms the argument to the function. Notice that the function does not use any variables except a and r. The variables a and r belong exclusively to the function find_area, i.e., the scope of the variables a and r is limited to the function find_area. We will learn more about scope rules in Chapter 19.

Next, the area is calculated in the function find_area and sent back to the calling program by the return statement.

```c
float find_area(float r)
{
    float a;
    a = PI * r * r;
    return(a);
}
```

Notice the text indentation style used in the program CIRCLE.C. Indentation is not necessary for the program to work but it is important because it makes the program more readable. Another useful element in making programs more readable is the judicious use of whitespace in the program.

### 1.7 Use of Whitespace in C

Spaces, tabs, blank lines, formfeed characters (special characters which tell printers to move to the next sheet), and carriage returns are known as whitespace characters. The presence of these characters in a C source code file is ignored by the C compiler. For example, the HELLO.C program could be written in the following manner without encountering any problems.
#include <stdio.h> main() {printf("Hello,world!\n");}

The only problem is that it is not very readable from a programmer's point of view, i.e., the various structural units of the program are not easy to find. Such programs indicate a poor style of programming.

Whitespace is used as a separator between names. For example, my show is considered as two names (or identifiers as we will learn to call them later) but myshow is considered as one single name.

Whitespace in a string constant is a significant part of the starting constant and is NOT ignored, e.g., the string constant "Hello,world!" is different from " Hello, world! ."

### 1.8 Constants in C

There are a various types of constants in any high level programming language. Constants in the C language can be broadly classified as follows:

- **NUMERIC CONSTANTS**
  - Integer Constants (e.g., 45, -567)
  - Real or Floating Point Constants (e.g., 456.78, -4.e35, -66.0E-98)
- **CHARACTER CONSTANTS**
  - Single Character Constants (e.g., 'K', 'a', '1', '0')
  - String Constants (e.g., "Hello,world", "1", "")

Each kind of constant is related to a specific data type and has specific storage requirements. These data types will be explored in detail in the next four chapters.

### 1.9 Points to Remember

- Compilers and interpreters are two ways to execute a program.
- A C program needs to compiled and linked before it can be executed (run).
- Every C program consists of a number of functions, each of them carrying out a specialized task.
- A C program must have one and only one function named main.
- Whitespace is ignored by the C compiler except when it occurs within a string constant.
- Whitespace should be used judiciously to make a program more "readable".

### Study Projects

1. Study the historical development of the C programming language.
2. Make a list of at least 10 high level computer languages. What are the special features of these languages?
3. Find out how computer languages are classified into various generations (first generation languages to fifth generation languages). To which generation does C belong to?

4. Find out the details of the C compiler/interpreter which you are using, i.e., the software developer, company name, etc. From the software manual or advertisements, find out the special features of your software as compared to other compilers/interpreters.

5. Find out the origins of the word "bug" as applied to an error in a computer program.

6. What are the features of the symbolic debugger of your compiler?

Review Quiz

1. Define the following terms in less than 5 lines each:
   (a) High level language
   (b) Algorithm
   (c) Compilation
   (d) Interpreter
   (e) Source code
   (f) Object code
   (g) Function

2. Why do we write programs in high level languages?

3. Can we write a comment inside another comment as shown below?

   ```c
   /* Outer level comment
   /* This is the inner comment */
   */
   ```

4. Are the following two programs equivalent from the C compiler's viewpoint?

   ```c
   #include <stdio.h>
   void main(){printf("Hello, world!\n");
   printf("Let's learn C!\n");
   }
   ```

5. State why each of the following strings are different from the string "My World".

   "my world"  "My World"  "My World"
Programming Exercises

1. The best way to learn programming is to take an existing program and make modifications to it. As you "play" with the program, you will encounter various errors and warnings, you will discover various new aspects of the language and you will learn about the finer points of the language. Run the HELLO.C program on your computer and, following that, perform the following exercises.

(a) Introduce various typing errors in the program and observe the various warnings and errors generated by the C compiler.

(b) Delete the first line of the program and observe the various warnings and errors generated by your compiler.

(c) Introduce a carriage return after the '!' character in the program as shown below and observe the warnings and errors.

```
printf("Hello,world!");
```

2. Run the HELLO2.C program on your computer and, following that, perform the following exercises.

(a) Delete the newline character from the first printf call and observe the effect on the output.

(b) Replace the \n with \n\n\n\n and observe the effect on the output.

(c) Delete the opening /* of a comment and observe the various warnings and errors generated by the C compiler.

3. Run the CIRCLE.C program on your computer and, following that, perform the following exercises.

(a) Delete the line containing the variable declarations in main and observe the warnings and errors generated by the compiler.

(b) Delete the line containing the function prototype declaration in main and observe the warnings and errors generated by the compiler.

(c) Move the variable declaration statement to the middle of main and observe the errors and warnings generated.

(d) Change the function call to the following and observe the errors and warnings generated.

```
area = find_area(radius,3.45); /* call to function */
```
2.0 Lesson Goals

- To understand how integers are stored in a computer.
- To learn about the different data types available in the C language for storing integers.
- To learn how to perform simple input and output of integers.

2.1 Storage Requirements for an Integer

Just like everything else on a computer, an integer needs to be represented as a sequence of binary digits. A bit stores one binary digit and eight bits make one byte of computer storage. We can allocate any number of bytes for an integer. To begin our study, let us consider a one byte representation of an integer.

If we need to store only non-negative integers, then it is possible to store integers ranging from 0 (corresponding to the binary value 00000000 and it's equivalent hexadecimal value of 00) to 255 (corresponding to the binary value 11111111 and it's equivalent hexadecimal value of FF). Therefore, there are 256 different integers which can be stored in this one byte unsigned integer representation which correspond to the 256 different bit patterns that can be generated using 8 bits (256 = 2^8).

But, if we want to store both negative and positive integers, the sign itself represents one bit of binary information (0 for positive and 1 for negative) and, therefore, we have only 7 bits left for representing the integer part. The maximum positive integer that can be represented using 7 bits is 127 (corresponding to the binary value 01111111 and it's equivalent hexadecimal value of 7F). The negative integers are stored using a 2's complement representation and the minimum negative integer works out to be -128 (corresponding to the binary value 10000000 and it's equivalent hexadecimal value of 80). The correspondence of the various bit patterns with the unsigned and signed integers is represented in Table 1.1.

Notice the unusual correspondence between the signed and the unsigned integers. The unsigned integer 128 corresponds to the signed integer value of -128 and the unsigned integer 255 corresponds to the signed integer value of -1.
# Integers

## Table 1.1 Correspondence between signed and unsigned integers

for a one byte integer

<table>
<thead>
<tr>
<th>Bit Pattern (Binary)</th>
<th>Hexadecimal (Unsigned)</th>
<th>Decimal (Unsigned)</th>
<th>Decimal (Signed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000000</td>
<td>00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>000000001</td>
<td>01</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>01111111</td>
<td>7F</td>
<td>127</td>
<td>127</td>
</tr>
<tr>
<td>10000000</td>
<td>80</td>
<td>128</td>
<td>-128</td>
</tr>
<tr>
<td>10000001</td>
<td>81</td>
<td>129</td>
<td>-127</td>
</tr>
<tr>
<td>11111110</td>
<td>FE</td>
<td>254</td>
<td>-2</td>
</tr>
<tr>
<td>11111111</td>
<td>FF</td>
<td>255</td>
<td>-1</td>
</tr>
</tbody>
</table>

**NOTE:** To find the 2's complement value for a given bit pattern, you can follow this example. To convert 8E₁₆, we note that its binary representation is given by 10001110, the one in the MSB (Most significant bit) indicates that this is a negative integer. Taking the 2’s complement bit by bit gives us 01110001. Adding 1 to this value results in 01110010. 

\[
(01110010)₂ = (72)₁₆ = 7*16 + 2 = (114)₁₀
\]

Therefore, as a signed integer, \((8E)₁₆ = (-114)₁₀\) in 2's complement signed representation.

As an unsigned integer, \((8E)₁₆ = (142)₁₀\)

Based on the above study, we can make the following generalizations for an n-bit integer representation.

<table>
<thead>
<tr>
<th>n-BIT INTEGER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum signed value = (-2^n)</td>
</tr>
<tr>
<td>Maximum signed value = (2^n - 1)</td>
</tr>
<tr>
<td>Minimum unsigned value = 0</td>
</tr>
<tr>
<td>Maximum unsigned value = (2^n - 1)</td>
</tr>
</tbody>
</table>

## 2.2 Integer Data Types in C

The ANSI standard specifies three kinds of integers, namely, short integers (**short int** or **short**), integers (**int**), and long integers (**long int** or **long**). Further, each of these data types can be specified to be **signed** or **unsigned**. This makes for a total of six kinds of integer data types.
The ANSI standard specifies that

- the number of bytes for a `short int` must be greater than or equal to 2 bytes,
- the number of bytes for an `int` must be greater than or equal to 2 bytes,
- the number of bytes for a `long int` must be greater than or equal to 4 bytes,
- the number of bytes for an `int` must be greater than or equal to that of a `short int`, and
- the number of bytes for a `long int` must be greater than or equal to that of an `int`.

However, the exact size for these data types in terms of bytes is not specified in the ANSI standard. **The exact size of the various integer types is implementation dependent**, i.e., it depends on the the specific C compiler. In turn, the developer of the compiler has to work within the constraints imposed by the hardware of the particular computer. A typical personal computer might allocate 2 bytes for a `short int`, 2 bytes for an `int` and 4 bytes for a `long int`, and by doing so, it fulfills the minimum requirements of the ANSI standard. Let us calculate the minimum and the maximum values for 2 byte and 4 byte integers.

### 2 BYTE INTEGER (32 bits)

<table>
<thead>
<tr>
<th>Minimum signed value</th>
<th>+32767</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum signed value</td>
<td>+32767</td>
</tr>
<tr>
<td>Minimum unsigned value</td>
<td>0</td>
</tr>
<tr>
<td>Maximum unsigned value</td>
<td>+65535</td>
</tr>
</tbody>
</table>

### 4 BYTE INTEGER (64 bits)

<table>
<thead>
<tr>
<th>Minimum signed value</th>
<th>+2147483647</th>
<th>≈ -2.14 x 10^9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum signed value</td>
<td>+2147483648</td>
<td>≈ +2.14 x 10^9</td>
</tr>
<tr>
<td>Minimum unsigned value</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Maximum unsigned value</td>
<td>+4294967295</td>
<td>≈ +4.29 x 10^9</td>
</tr>
</tbody>
</table>

### 2.3 The LIMITS.H File

Where do we find the implementation dependent details about the integer data types? These are supplied in a file named `LIMITS.H` (the `.H` extension represents a header file) which comes along with every ANSI standard C compiler. Shown below is a portion of a typical `LIMITS.H` file containing various symbolic constants defining the limits of the various integer data types.

```c
#define SHRT_MAX            0x7FFF
#define SHRT_MIN            ((int)0x8000)
#define USHRT_MAX           0xFFFFU
#define INT_MAX             0x7FFF
#define INT_MIN             ((int)0x8000)
#define UINT_MAX            0xFFFFU
#define LONG_MAX            0x7FFFFFFL
#define LONG_MIN            ((long)0x80000000L)
#define ULONG_MAX           0xFFFFFFFFUL
```
Given below is a small C program, INTLIMS.C, which makes use of these symbolic constants to print out the various integer limits in both decimal and hexadecimal forms.

Program 2.1 – INTLIMS.C  
Printing the integer limits of your computer

```c
/*  ================= INTLIMS.C ==============================  
This program prints out the limits of the various  
types of integers used on a particular compiler.  
The various symbolic constants used for these limits  
are defined in the header file LIMITS.H.  
================================================================*/
#include <stdio.h>  
#include <limits.h>  

void main()  
{  
    printf("INTEGER LIMITS \n");  
    printf("SHRT_MAX = %d \t= %X", SHRT_MAX, SHRT_MAX);  
    printf("USHRT_MAX = %u \t= %X", USHRT_MAX, USHRT_MAX);  
    printf("INT_MAX = %d \t= %X", INT_MAX, INT_MAX);  
    printf("INT_MIN = %d \t= %X", INT_MIN, INT_MIN);  
    printf("UINT_MAX = %u \t= %X", UINT_MAX, UINT_MAX);  
    printf("LONG_MAX = %ld \t= %lX", LONG_MAX, LONG_MAX);  
    printf("LONG_MIN = %ld \t= %lX", LONG_MIN, LONG_MIN);  
    printf("ULONG_MAX = %lu \t= %lX", ULONG_MAX, ULONG_MAX);  
}
```

The output from this program might look like the following listing but the exact values may be different on your compiler.

```
INTEGER LIMITS  
SHRT_MAX = 32767 = 7FFF  
USHRT_MAX = 65535 = FFFF  
INT_MAX = 32767 = 7FFF  
INT_MIN = -32768 = 8000  
UINT_MAX = 65535 = FFFF  
LONG_MAX = 2147483647 = 7FFFFFFF  
LONG_MIN = -2147483648 = 80000000  
ULONG_MAX = 4294967295 = FFFFFFFF
```

Knowledge of the minimum and maximum limits of each of these data types is essential for designing a robust program using these data types.

### 2.4 Integer Constants

The most common integer constants are ordinary decimal integers, e.g., 143, -7893, etc. An octal integer is specified by prefixing it with a 0 (zero). For instance, 077, 065, and 01111 are all valid octal integers. Hexadecimal integers are represented using a leading 0x or 0X, e.g., 0x73f, 0xFF0A, etc.
A suffix \texttt{u} or \texttt{U} indicates an unsigned integer. A suffix \texttt{L} or \texttt{l} indicates a long integer value. Therefore, \texttt{22UL} represents a long unsigned integer value of 22 and \texttt{343u} represents an unsigned integer value of 343.

What is the data type of \texttt{1234}? For decimal integers having no suffixes (\texttt{u}, \texttt{U}, \texttt{l} or \texttt{L}), the compiler tries the possibility of storing it using the data type \texttt{int}. If that is not possible, the compiler will try \texttt{long int} for storage. If the number is too large for a \texttt{long int} and is not negative, then the compiler tries to use the data type \texttt{unsigned long int}. Similar rules apply to the octal and hexadecimal integers also. The following extract from Kernighan and Ritchie\cite{7} explains these rules (italicized text inserted).


cite{7}

If it \texttt{[integer constant]} is unsuffixed and decimal, it has the first of these types in which its value can be represented: \texttt{int, long int, unsigned long int}.
If it is unsuffixed octal or hexadecimal, it has the first possible of these types: \texttt{int, unsigned int, long int, unsigned long int}.
If it is suffixed by \texttt{u} or \texttt{U}, then \texttt{unsigned int, unsigned long int}.
If it is suffixed by \texttt{l} or \texttt{L}, then \texttt{long int, unsigned long int}.

### 2.5 Simple Input/Output of Integers

Let us look at a simple program that performs the following tasks.
- Outputs a message asking for input.
- Accepts input of an integer from the user.
- Stores the value in a variable named \texttt{mynum}.
- Outputs the value to the monitor.

**Program 2.2 – IOINTEGR.C**  
**Input and output of integers**

```c
/* ==============================================================
 * IOINTEGR.C  
 * Program to input and output an integer.  
 * ==============================================================*/
#include <stdio.h>
void main()
{
    int mynum;
    printf("Enter an integer value: ");
    scanf("%d", &mynum);
    printf("The integer you have entered is %d", mynum);
}
```

The first step is to declare the variable \texttt{mynum} of type \texttt{int}. Next, we use the \texttt{printf} function for output and \texttt{scanf} function for input. Notice the prefix \& placed before \texttt{mynum} in the \texttt{scanf} function call. The symbol \& translates to "address of" and we will look at this in more detail when we learn about pointers in Chapter 23. For the present, use the following rule-of-thumb blindly.
When calling the `scanf` function, place an ampersand (&) before the variable name which is being input (for all integers, floating point numbers, and characters).

In the `scanf` statement and in the last `printf` statement, the %d is replaced by the value of the variable `mynum` in the output. The conversion specifier %d indicates to the computer that the value of `mynum` must be printed as an integer.

For a variable of type `short int`, use the conversion specifier %hd instead of %d.

For a variable of type `long int`, use the conversion specifier %ld instead of %d.

### 2.6 Points to Remember

- Negative integers are stored using a two’s complement scheme.
- The minimum and maximum integer values can be calculated from the number of bytes allocated to the integer data type and our choice of a signed or unsigned representation system.
- The same bit pattern can represent two different integer values in signed and unsigned data types.
- There are three integer data types in C, i.e., `short int` (or `short`), `int` and `long int` (or `long`). Each of these data types may be `signed` or `unsigned` (the default is `signed`).
- The number of bytes allocated to each of these data types is implementation dependent (i.e., depends on your compiler and your computer). Therefore, to ensure portability from one implementation to another, your program should make no assumptions regarding the actual size of any integer data type.

### Review Quiz

1. Calculate the minimum and maximum limits for (a) a 3 byte signed integer data type and (b) a 3 byte unsigned integer data type.

2. Convert the following from the given base to the other specified base. If you are having difficulty with these base conversions, refer to any standard textbook which teaches fundamentals of computers.
   
   (a) \((11000111)_2 = (?)_8\)
   (b) \((10110111)_2 = (?)_{16}\)
   (c) \((975)_8 = (?)_2\)
   (d) \((673)_2 = (?)_{16}\)
   (e) \((127)_{10} = (?)_2\)
   (f) \((255)_{10} = (?)_{16}\)
   (g) \((FEED)_{16} = (?)_8\)
   (h) \((FF)_{16} = (?)_2\)

3. Which decimal integers do the following bit patterns represent in a signed integer representation? What do they represent as unsigned integers?

   \((11111111)_2\)
   \((00001111)_2\)
   \((10000000)_2\)
   \((01111111)_2\)

4. If a new compiler uses 5 bytes for the `short`, the `int` and the `long` data types, does it conform to ANSI standards?
5. If a compiler uses 2 bytes for the \texttt{short} and the \texttt{int} data types and 4 bytes for the \texttt{long} data type, what will be the data type for each of the following constants?

(a) $-378$  
(b) $44444$  
(c) $-33333$  
(d) $3FFFFu$  
(e) $FFF1$

\begin{center}
\textbf{Programming Exercises}
\end{center}

1. Modify IOINTEGR.C for the input/output a short integer.
2. Modify IOINTEGR.C for the input/output a long integer.
3. Modify IOINTEGR.C for the input/output an unsigned long integer.
3.0 Lesson Goals

- To understand how real numbers are stored in a computer.
- To understand the implications of a finite precision representation for real numbers.
- To learn about the different data types available in the C language for storing real numbers.
- To learn how to perform simple input and output of real numbers.

3.1 A Hypothetical Decimal Computer

A real number (usually referred to as a floating point number) is represented by two integers in a computer, the first integer representing a normalized mantissa and the second integer representing an exponent. A normalized mantissa has an absolute value less than 1.0 and greater than or equal to 0.1.

\[
\text{real number} = M \times b^E
\]

where

- \(b\) is the base of the representation system,
- \(E\) is the exponent, and
- \(M\) is a normalized mantissa

such that

\[
(1/b) \leq |M| < 1.0
\]

(|M| is the absolute value of M).

Before we examine these issues in the binary system, it is easier to visualize the issues in the decimal system because most of us tend to think in decimal in our daily life.

For this purpose, let us take a hypothetical decimal computer in which one decimal digit can be stored in each "decimal bit". Let us assume that the normalized mantissa is stored in 4 "decimal bits" plus one sign bit and the exponent is stored in two "decimal bits" plus one sign bit. The representation of some numbers on this hypothetical decimal computer is shown in Table 3.1. On our hypothetical decimal computer, we seek a representation of the form

\[
\text{real number} = M \times 10^E
\]

Since, we are using base 10, the normalized mantissa must satisfy the condition

\[
0.1 \leq |M| < 1.0
\]
Table 3.1 Representation on a Hypothetical Decimal Computer

<table>
<thead>
<tr>
<th>Real Number</th>
<th>Normalized Real Number</th>
<th>Mantissa</th>
<th>Exponent</th>
</tr>
</thead>
<tbody>
<tr>
<td>33.45</td>
<td>+0.3345 x 10^2</td>
<td>3345</td>
<td>2</td>
</tr>
<tr>
<td>0.02345 x 10^{-3}</td>
<td>+0.2345 x 10^{-4}</td>
<td>2345</td>
<td>-4</td>
</tr>
<tr>
<td>-340</td>
<td>-0.3400 x 10^3</td>
<td>-3400</td>
<td>3</td>
</tr>
<tr>
<td>99.99 x 10^{97}</td>
<td>+0.9999 x 10^{99}</td>
<td>9999</td>
<td>99</td>
</tr>
</tbody>
</table>

Now, let us look at some important issues connected with this computer.

Can we represent the the value 0.23452 on this computer? It cannot be represented because we can only store 4 digits of the normalized mantissa but here we need 5 digits. The best we can do is to store the mantissa as 2345 and the exponent as 0, leading to a loss of precision. We see that the number of digits in the representation sets the limit for the precision with which real numbers can be stored in a computer. As computer programmers, we have to learn to live in this world of **finite precision** real numbers.

What happens when we try to compute 0.001 + 1.0 on this computer? To add two real numbers, we must first equalize their exponents as shown.

\[
0.1 \times 10^1 + 0.1 \times 10^{-2} = 0.1 \times 10^1 + 0.0001 \times 10^1
\]

To equalize the two exponents, we perform a right shift of all the digits of the second number by 3 places and increase the exponent to +1.

<table>
<thead>
<tr>
<th>Second number = 0.1 \times 10^{-2}</th>
<th>mantissa = 1000 exponent = -2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Second number (after right shift by 3 places)</td>
<td>mantissa = 0001 exponent = +1</td>
</tr>
<tr>
<td>First number = 0.1 \times 10^1</td>
<td>mantissa = 1000 exponent = +1</td>
</tr>
<tr>
<td>SUM</td>
<td>mantissa = 1001 exponent = +1</td>
</tr>
</tbody>
</table>

We obtain the expected result of 0.1001 x 10^1 or 1.001. Next, let us try to add 0.0001 and 1.0 on this computer.

<table>
<thead>
<tr>
<th>Second number = 0.1 \times 10^{-3}</th>
<th>mantissa = 1000 exponent = -3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Second number (after right shift by 3 places)</td>
<td>mantissa = 0000 exponent = +1</td>
</tr>
<tr>
<td>First number = 0.1 \times 10^1</td>
<td>mantissa = 1000 exponent = +1</td>
</tr>
<tr>
<td>SUM</td>
<td>mantissa = 1000 exponent = +1</td>
</tr>
</tbody>
</table>

We got 1.0 + 0.0001 = 1.0 which is definitely NOT what we expected! The significant digit 1 in 0.0001 was lost during the right shifting of the digits performed for exponent equalization. Such a loss
of significant digits occurs in almost every arithmetic operation that uses a finite precision representation for real numbers.

For any finite precision representation of floating point numbers, the **machine accuracy** is the smallest number \( \varepsilon \) such that

\[
1 + \varepsilon \neq 1.
\]

On our hypothetical computer, we see that the machine accuracy is 0.001.

Next, let us examine how many real numbers have an exponent of 1. The smallest mantissa is 1000 and the largest is 9999. This gives us 9000 real numbers that lie between 1 and 10. We see that, because of finite precision, there are only a small finite number of real numbers between any two real numbers!

Next, let us how many numbers lie between 10 and 100. The answer is once again 9000. In a similar fashion, there are 9000 real numbers that can be represented with values between \( 10^{99} \) and \( 10^{99} \). There are also just 9000 real numbers that can be represented with values between \( 10^{-99} \) and \( 10^{-99} \)! It is obvious that these numbers are not evenly distributed along the number but tend to be more densely populated near zero. This is another consequence of the finite precision of our representation of real numbers.

Let us summarize what we have learnt about our hypothetical decimal computer.

- Only a finite number of significant digits can be stored.
- There may be a loss of significant digits during arithmetic operations like addition. A lower machine accuracy guarantees that more significant digits will be preserved.
- Given any two real numbers stored in a finite precision representation, there are a finite number of storable real numbers in between these two numbers.
- The distribution of the storable real numbers is extremely uneven with a large concentration near zero.

### 3.2 Binary Storage of Real Numbers

Coming back to the real world binary computers, let us assume that we have a 5 bit binary representation (excluding sign bit) for a binary mantissa. With this, let us look at the simple task of converting a decimal fraction, 0.75, to its binary form.

\[
\begin{align*}
0.75 \times 2 &= 1.5 & \text{integer part} &= 1 & \text{fraction part} &= 0.5 \\
0.5 \times 2 &= 1 & \text{integer part} &= 1 & \text{fraction part} &= 0.0
\end{align*}
\]

Therefore,

\[
(0.75)_{10} = (0.11)_2
\]

Next, let us try to convert 0.2.

\[
\begin{align*}
0.2 \times 2 &= 0.4 & \text{integer part} &= 0 & \text{fraction part} &= 0.4 \\
0.4 \times 2 &= 0.8 & \text{integer part} &= 0 & \text{fraction part} &= 0.8 \\
0.8 \times 2 &= 1.6 & \text{integer part} &= 1 & \text{fraction part} &= 0.6 \\
0.6 \times 2 &= 1.2 & \text{integer part} &= 1 & \text{fraction part} &= 0.2
\end{align*}
\]
\[ 0.2 \times 2 = 0.4 \quad \text{integer part} = 0 \quad \text{fraction part} = 0.4 \]
\[ 0.4 \times 2 = 0.8 \quad \text{integer part} = 0 \quad \text{fraction part} = 0.8 \]

Therefore, \((0.2)_10 = (0.001100110011\ldots)_2\).

A terminating decimal number can become a nonterminating binary number! With 5 binary digits, we can only store the mantissa as \((0.00110)_2\) which when converted back to decimal results in a value of 0.1875. Increasing the size of the mantissa to 8 bits will allow us to store 0.2 as \((0.00110011)_2\) which when converted back to decimal is 0.1992. This is known as roundoff error. So, we see that the very process of converting a decimal fraction to binary can lead to a loss of accuracy.

The representation of some numbers using a binary mantissa of 8 bits is shown in Table 3.2.

<table>
<thead>
<tr>
<th>Real Number (Decimal)</th>
<th>Normalized Binary Mantissa</th>
<th>Exponent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10000000</td>
<td>(\times 2^1)</td>
</tr>
<tr>
<td>3</td>
<td>11000000</td>
<td>(\times 2^2)</td>
</tr>
<tr>
<td>0.25</td>
<td>10000000</td>
<td>(\times 2^{-1})</td>
</tr>
<tr>
<td>10(^{-7})</td>
<td>11010110</td>
<td>(\times 2^{-31})</td>
</tr>
</tbody>
</table>

We see that in base 2, the normalized mantissa must satisfy the condition

\[(0.1)_2 \leq |M| < 1.0 \quad \text{(as 1/2 in base 10 equals 0.1 in base 2)}\]

Let us calculate the machine accuracy for this representation.

\[\varepsilon = 2^{-7} = 7.8125 \times 10^{-3}\]

To calculate the significant digits of accuracy (in base 10) that we can expect, we take the logarithm of \(\varepsilon\) to base 10.

Number of significant digits = \(- \log_{10}(\varepsilon) = 2.1072 \approx 2\)

This implies that, speaking on the average, we can expect about 2 decimal digits of accuracy with an 8 bit binary mantissa. Note that this is only a rough estimate of the average accuracy because in some cases the accuracy will significantly lower as we had seen earlier in the case of numbers like 0.2.

Another interesting fact that can be seen from Table 3.2 is that the MSB of the normalized mantissa is always 1. By the very definition of a normalized mantissa, the MSB cannot be 0 and in the binary system, if the MSB is not 0, it must be 1. Note that on our hypothetical decimal computer of Section 3.1 the most significant digit may be any nonzero digit, i.e., any digit between 1 to 9. On a binary computer, the only nonzero digit is 1 and, therefore, the MSB of the normalized mantissa must be 1.
Using this fact, we can choose NOT to store this bit and yet assume its presence. In such cases, it is known as a **ghost bit**. Using a ghost bit, only 7 bits are necessary to store an 8 bit mantissa. We gain the accuracy given by an extra bit without wasting any physical storage for this bit. But the ghost bit creates other problems during computations and, for this reason, is not used on all machines (see Press et al.[14] for more details).

Let us look at a more complete example of a floating point representation. Let us look at a 64 bit representation organized in the following fashion.

| Bits for mantissa | = 52 + 1 ghost bit = 53 effective bits |
| Sign bit for mantissa | = 1 |
| Sign bit for exponent | = 1 |
| Bits for exponent | = 10 |
| TOTAL | = 64 |

In this representation,

- **Machine accuracy** = \( \varepsilon = 2^{-52} = 2.220446 \times 10^{-16} \)
- **Number of significant digits** = \(-\log_{10}(\varepsilon) = 15.653 \approx 15\)

The largest positive value that can be stored depends on the maximum exponent (because the largest normalized mantissa is 0.1111111111111... \( \approx 1 \)) which is \(2^{10} = 1024\).

\[
\text{Largest possible value} = 2^{1024} \approx 1.8 \times 10^{308}
\]

Any larger value cannot be stored in this form and will result in an **overflow error**.

### 3.3 Floating Point Data Types

There are three floating point data types in the C language for representing real numbers, namely, float, double (standing for double precision), and long double. All floating point data type are signed, i.e., you cannot use the unsigned modifier for any of these data types. The precision (i.e., number of significant digits) and range of exponents is implementation dependent for all of these data types. The only conditions to be met are that the type long double must be at least as precise as the type double and the type double must be at least as precise as float.

### 3.4 The FLOAT.H File

Where do we find the implementation dependent details about the floating point data types? These are supplied in a file named FLOAT.H, an include file which is a part of every ANSI C compiler. As an alternative, you can execute the program FLOATLIM.C given below.
Program 3.1 – FLOATLIM.C  Printing floating point limits of your compiler

/*  ============================================================== FLOATLIM.C ==============================================================
Program prints the limits of the floating point variables
based on symbolic constants given in the header file FLOAT.H.
================================================================================================*/
#include <stdio.h>
#include <float.h>

void main()
{
    printf("\nFLOAT data type\n===============\nN-bits in mantissa   = %u",FLT_MANT_DIG);
    printf("\nN-significant digits = %u",FLT_DIG);
    printf("\nepsilon        = %e",FLT_EPSILON);
    printf("\nSmallest value = %e",FLT_MIN);
    printf("\nLargest value  = %e",FLT_MAX);

    printf("\nDOUBLE data type\n================\nN-bits in mantissa   = %u",DBL_MANT_DIG);
    printf("\nN-significant digits = %u",DBL_DIG);
    printf("\nepsilon        = %e",DBL_EPSILON);
    printf("\nSmallest value = %le",DBL_MIN);
    printf("\nLargest value  = %le",DBL_MAX);

    printf("\nLONG DOUBLE data type\n====================\nN-bits in mantissa   = %u",LDBL_MANT_DIG);
    printf("\nN-significant digits = %u",LDBL_DIG);
    printf("\nepsilon        = %e",LDBL_EPSILON);
    printf("\nSmallest value = %Le",LDBL_MIN);
    printf("\nLargest value  = %Le",LDBL_MAX);
}

A sample output is shown below.

FLOAT data type
===============
N-bits in mantissa   = 24  
N-significant digits = 6
epsilon        = 1.192093e-07
Smallest value = 1.175494e-38
Largest value  = 3.402823e+38

DOUBLE data type
================
N-bits in mantissa   = 53  
N-significant digits = 15
epsilon        = 2.220446e-16
Smallest value = 2.225074e-308
Largest value  = 1.797693e+308

LONG DOUBLE data type
=====================
N-bits in mantissa   = 64  
N-significant digits = 19
epsilon        = 1.084202e-19
Smallest value = 3.362103e-4932
Largest value  = 1.189731e+4932
3.5 Floating Point Constants in C

A floating point constant contains either a decimal point or an integral exponent (given after and e or E). Therefore, 123.45, 34.4E-76, 5e78, and 1. are all valid floating point constants. The default type is assumed to be double. A suffix f or F specifies a float value. A suffix l or L specifies a long double value. Therefore, 4.6f is considered a float type value and 4.6L is considered a long double type of value.

3.6 Simple Input/Output of Real Numbers

Let us look at a simple program that performs the following tasks

- Outputs a message asking for input.
- Accepts input of a real number from the user.
- Stores the value in a variable named rnum.
- Outputs the value to the monitor.

Program 3.2 – IOREALS.C

```c
/* ============================  IOREALS.C ==============================
 Program to input and output a real number (type - float).
 ========================================================================*/
#include <stdio.h>
void main()
{
    float rnum;
    printf("Enter a real number : ");
    scanf("%f",&rnum);
    printf("The integer you have entered is %f",rnum);
}
```

In the `scanf` statement and in the last `printf` statement, the `%f` is replaced by the value of the variable `rnum` in the output. The specifier `%f` indicates to the computer that the value of `rnum` must be printed as a float. Optionally, you can use `%e` or `%g` instead of `%f`.

For a variable of type `double`, you can use `%lf`, `%le`, or `%lg`.

For a variable of type `long double`, you can use `%Lf`, `%Le`, or `%Lg`.

Let us look at simple example to understand the significance of precision. The output from the statement

```c
printf("%f,%lf",123.45f,123.45);
```

is

```
123.449997,123.450000
```
on my computer (it may be different on another implementation). Here we see the effect of changing a
decimal fraction to a finite precision binary floating point representation.

3.7 Points to Remember

• Real numbers are stored using a floating point representation in which the integral mantissa and
  the integral exponent are stored.
• Only a finite number of significant digits can be stored. This finite precision representation leads
to many unusual characteristics of floating point data types.
• The accuracy of a floating point data type depends on the number of bits used to store the
  mantissa.
• There is a loss of accuracy in almost every computation involving floating point data types
  because of the finite precision representation.
• There are three floating point data types in C for representing real numbers, namely, float, double, and long double. The range and accuracy of each of these data types is
  implementation dependent.

Review Quiz

1. Given an 80 bit floating point representation organized in the following fashion:
   
   | Bits for mantissa | = 63 |
   | Sign bit for mantissa | = 1 |
   | Sign bit for exponent | = 1 |
   | Bits for exponent | = 15 |
   
   Assuming that there is no ghost bit, calculate the following:
   (a) Accuracy
   (b) Number of significant digits
   (c) Smallest representable real number
   (d) Largest representable real number

Programming Exercises

1. Modify IOREALS.C to perform input/output for a double variable.
2. Modify IOREALS.C to perform input/output for a long double variable.
4.0 Lesson Goals

- To understand how characters are represented in C.
- To learn about the various character sets used in computers.
- To learn how to perform simple input and output of characters.

4.1 The ASCII Character Set

Before we study how characters are represented in C, we will look at the related issue of character sets. Like everything else on a computer, a character must be represented by a binary string of 1s and 0s on a computer. We need to agree upon a standard mapping of a character to a bit pattern. In the absence of such a standard, there would be utter confusion. For example, if I use the bit pattern 00000001 to represent the character 'A' and you decide to use the same bit pattern for '?', it would become impossible for us to exchange information. This is where the standard character sets step in and prevent the resulting chaos! A standard character set represents a mapping of bit patterns to characters agreeable to a large section of the programming community.

The most popular way of representing characters is using the ASCII (pronounced as as-kee and standing for American Standard Code for Information Interchange) code which uses 7 bits for representing a character. Historically speaking the ASCII code was developed for use in teleprinters where the eighth bit was reserved as a parity checking bit (used for checking the integrity of the transmitted data). Let us take a more detailed look at the 128 characters (numbered from 0 to 127) of the ASCII character set shown in Table 4.1. The common names of the graphic characters are shown in Table 4.2.

The ASCII characters from 0 to 31 and 127 are called control characters. These are non-printing characters and most of them are used to control the operation of some hardware device.

- The character 7 (BEL=bell) when sent to a monitor produces a short beep sound on the in-built speaker.
- The character 9 (HT=horizontal tab) inserts a tab in the text which moves the cursor to the next tab position.
- The character 10 (LF=linefeed or newline) moves the cursor to the start of the next line.
- The character 11 (VT=vertical tab) produces a vertical tab.
- The character 12 (FF=formfeed) instructs a printer to eject the current sheet being printed.
- The character 13 (CR=carriage return) moves the cursor to the start of the current line.
- The character 27 (ESC=escape) is used as the beginning character of various hardware control sequences (e.g., for printer setup etc.).

Table 4.1 The ASCII Character Set

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NUL</td>
<td>32</td>
<td>space</td>
<td>64</td>
<td>@</td>
</tr>
<tr>
<td>1</td>
<td>SOH</td>
<td>33</td>
<td>!</td>
<td>65</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>STX</td>
<td>34</td>
<td>&quot;</td>
<td>66</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>ETX</td>
<td>35</td>
<td>#</td>
<td>67</td>
<td>C</td>
</tr>
<tr>
<td>4</td>
<td>EOT</td>
<td>36</td>
<td>$</td>
<td>68</td>
<td>D</td>
</tr>
<tr>
<td>5</td>
<td>ENQ</td>
<td>37</td>
<td>%</td>
<td>69</td>
<td>E</td>
</tr>
<tr>
<td>6</td>
<td>ACK</td>
<td>38</td>
<td>&amp;</td>
<td>70</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>BEL</td>
<td>39</td>
<td></td>
<td>71</td>
<td>G</td>
</tr>
<tr>
<td>8</td>
<td>BS</td>
<td>40</td>
<td>(</td>
<td>72</td>
<td>H</td>
</tr>
<tr>
<td>9</td>
<td>HT</td>
<td>41</td>
<td>)</td>
<td>73</td>
<td>I</td>
</tr>
<tr>
<td>10</td>
<td>LF</td>
<td>42</td>
<td>*</td>
<td>74</td>
<td>J</td>
</tr>
<tr>
<td>11</td>
<td>VT</td>
<td>43</td>
<td>+</td>
<td>75</td>
<td>K</td>
</tr>
<tr>
<td>12</td>
<td>FF</td>
<td>44</td>
<td>,</td>
<td>76</td>
<td>L</td>
</tr>
<tr>
<td>13</td>
<td>CR</td>
<td>45</td>
<td>-</td>
<td>77</td>
<td>M</td>
</tr>
<tr>
<td>14</td>
<td>SO</td>
<td>46</td>
<td>.</td>
<td>78</td>
<td>N</td>
</tr>
<tr>
<td>15</td>
<td>SI</td>
<td>47</td>
<td>/</td>
<td>79</td>
<td>O</td>
</tr>
<tr>
<td>16</td>
<td>DLE</td>
<td>48</td>
<td>0</td>
<td>80</td>
<td>P</td>
</tr>
<tr>
<td>17</td>
<td>DC1</td>
<td>49</td>
<td>1</td>
<td>81</td>
<td>Q</td>
</tr>
<tr>
<td>18</td>
<td>DC2</td>
<td>50</td>
<td>2</td>
<td>82</td>
<td>R</td>
</tr>
<tr>
<td>19</td>
<td>DC3</td>
<td>51</td>
<td>3</td>
<td>83</td>
<td>S</td>
</tr>
<tr>
<td>20</td>
<td>DC4</td>
<td>52</td>
<td>4</td>
<td>84</td>
<td>T</td>
</tr>
<tr>
<td>21</td>
<td>NAK</td>
<td>53</td>
<td>5</td>
<td>85</td>
<td>U</td>
</tr>
<tr>
<td>22</td>
<td>SYM</td>
<td>54</td>
<td>6</td>
<td>86</td>
<td>V</td>
</tr>
<tr>
<td>23</td>
<td>ETB</td>
<td>55</td>
<td>7</td>
<td>87</td>
<td>W</td>
</tr>
<tr>
<td>24</td>
<td>CAN</td>
<td>56</td>
<td>8</td>
<td>88</td>
<td>X</td>
</tr>
<tr>
<td>25</td>
<td>EM</td>
<td>57</td>
<td>9</td>
<td>89</td>
<td>Y</td>
</tr>
<tr>
<td>26</td>
<td>SUB</td>
<td>58</td>
<td>:</td>
<td>90</td>
<td>Z</td>
</tr>
<tr>
<td>27</td>
<td>ESC</td>
<td>59</td>
<td>;</td>
<td>91</td>
<td>[</td>
</tr>
<tr>
<td>28</td>
<td>FS</td>
<td>60</td>
<td>&lt;</td>
<td>92</td>
<td>\</td>
</tr>
<tr>
<td>29</td>
<td>GS</td>
<td>61</td>
<td>=</td>
<td>93</td>
<td>]</td>
</tr>
<tr>
<td>30</td>
<td>RS</td>
<td>62</td>
<td>&gt;</td>
<td>94</td>
<td>^</td>
</tr>
<tr>
<td>31</td>
<td>US</td>
<td>63</td>
<td>?</td>
<td>95</td>
<td>_</td>
</tr>
<tr>
<td>96</td>
<td>`</td>
<td>127</td>
<td>DEL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 4.2 Names of the Graphic Characters

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 to 9</td>
<td>Decimal digits</td>
</tr>
<tr>
<td>A to Z</td>
<td>Uppercase or Capital letters</td>
</tr>
<tr>
<td>a to z</td>
<td>Lowercase or Small letters</td>
</tr>
<tr>
<td>!</td>
<td>Bang, Exclamation mark</td>
</tr>
<tr>
<td>#</td>
<td>Hash, Number/pound sign</td>
</tr>
<tr>
<td>$</td>
<td>Dollar sign</td>
</tr>
<tr>
<td>%</td>
<td>Percent sign</td>
</tr>
<tr>
<td>&amp;</td>
<td>And sign, ampersand</td>
</tr>
<tr>
<td>&quot;</td>
<td>Double quotes</td>
</tr>
<tr>
<td>'</td>
<td>Single quote</td>
</tr>
<tr>
<td>&quot;</td>
<td>Back quote, Accent grave</td>
</tr>
<tr>
<td>*</td>
<td>Star, Asterisk</td>
</tr>
<tr>
<td>+</td>
<td>Plus</td>
</tr>
<tr>
<td>-</td>
<td>Minus/Hyphen</td>
</tr>
<tr>
<td>/</td>
<td>Slash</td>
</tr>
<tr>
<td>\</td>
<td>Backslash</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>Left parenthesis</td>
</tr>
<tr>
<td>)</td>
<td>Right parenthesis</td>
</tr>
<tr>
<td>[</td>
<td>Left bracket</td>
</tr>
<tr>
<td>]</td>
<td>Right bracket</td>
</tr>
<tr>
<td>{</td>
<td>Left brace</td>
</tr>
<tr>
<td>}</td>
<td>Right brace</td>
</tr>
<tr>
<td>_</td>
<td>Underscore</td>
</tr>
<tr>
<td>?</td>
<td>Question mark</td>
</tr>
<tr>
<td>@</td>
<td>&quot;At&quot; sign or Rate symbol</td>
</tr>
<tr>
<td>^</td>
<td>Caret, circumflex</td>
</tr>
<tr>
<td>~</td>
<td>Tilde</td>
</tr>
<tr>
<td>:</td>
<td>Colon</td>
</tr>
<tr>
<td>;</td>
<td>Semicolon</td>
</tr>
<tr>
<td>.</td>
<td>Period</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>=</td>
<td>Equal to</td>
</tr>
</tbody>
</table>

The character 0 (NUL=null) is known as the null character and plays an important role in the storage of string constants in the C language.

The character 32 is the space character (entered using the spacebar on the keyboard). The space character along with the horizontal tab, newline, vertical tab, formfeed and carriage return characters are together known as whitespace characters. We had seen in Chapter 1 that whitespace characters are ignored by the C compiler.

Characters 32 through 126 are known as printing characters (even though the space character does not print anything, it does create a blank space). These include the upper case letters (from 65 through 90), the lower case letters (from 97 through 122), and the ten decimal digits (from 48 through 57). All printing characters except the letters, digits and the space are considered punctuation characters in C terminology. We will reexamine these issues once again when we look at the standard character functions in Chapter 10.

The program PRNASCII.C given below displays the printable ASCII characters on the monitor.
Program 4.1 – PRNASCII.C

/* ================================================== PRNASCII.C ==============================================================
 Program to print all printable ASCII characters to the screen along with their ASCII codes
 ==============================================================*/
#include <stdio.h>

void main()
{
    unsigned char j;
    printf("\n LIST OF PRINTABLE ASCII CHARACTERS\n");
    printf("\n\n\n");
    for(j=33;j<127;j++)
    {
        printf(" %3d  %c%c",j,j,j%8?' ':'
);
    }
}

4.2 The IBM Character Set or Extended ASCII Code

Since, modern computers do not need parity checking, it is quite safe to use the eighth bit also for storage of additional characters. This can be done in two ways.

(1) by adding the codes -128 through -1, creating a signed character representation, OR
(2) by adding codes 128 through 255 creating an unsigned character representation.

Since, every unsigned bit pattern has its equivalent signed value, we will refer only to the unsigned representation in the following discussion.

Such a character set that has codes from 0 through 255 is known as an extended character set and the exact symbols stored in an extended character set may differ from computer to computer.

The characters 128 through 255 on IBM compatible personal computers have been standardized into the IBM character set. The IBM character, given in Appendix A, also substitutes the control characters 0 through 31 with printable symbols. These additional symbols are useful in creating neat looking user interfaces and/or adding an artistic touch to your outputs which are usually very dull and prosaic.

Program 4.2 – IBMCHARS.C

/* ============================================================== IBMCHARS.C ==============================================================
 Program to print all printable characters of IBM extended character set to the screen along with their ASCII codes
 ==============================================================*/
#include <stdio.h>

void main()
{ unsigned char a;
    printf("\n LIST OF IBM-PC EXTENDED CHARACTERS\n");
    printf("\n\n\n");
    for(a=128;a<255;a++)
    {
        printf(" %3d %c%c",a,a,((a+1)%8)?' ':'\n');
    }
}
The following program, DRAWBOX.C, when executed on an IBM compatible PC will produce a box surrounding the text.

```
/* ==============================================================
 * DRAWBOX.C ==============================================================
This program draws a box uses IBM PC character set
 ==============================================================*/
#include <stdio.h>
void main()
{
    unsigned char top_left=201,top_right=187;
    unsigned char bot_left=200,bot_right=188;
    unsigned char dash = 205,bar=186;
    printf("\t%c%c%c%c\n",top_left,dash,dash,dash,top_right);
    printf("\t%c   %c\n",bar,bar);
    printf("\t%cBOX%c\n",bar,bar);
    printf("\t%c   %c\n",bar,bar);
    printf("\t%c%c%c%c%c\n",bot_left,dash,dash,dash,bot_right);
}
```

4.3 Characters in ANSI C

Characters in C are stored as one byte integers. Therefore, all the calculations that we performed for understanding the limits of one byte integers in Chapter 2 are valid for characters. Furthermore, all characters are valid integer constants in the C language and can be used wherever an integer value is expected. This dual nature of characters, i.e., the fact that they represent both a character and an integer at the same time is a powerful feature of the C language and provides a convenient solution to many problems. However, it is also the source of much confusion for novice programmers!

The ANSI standard provides for a data type char for representing characters. Two modifications of this type are signed char and unsigned char. Whether char is signed or unsigned by default is implementation dependent. This creates no problems as long as we are using the standard ASCII character set (0 through 127). But to ensure portability of any program that uses an extended character set, it is a good practice to specify signed char or unsigned char instead of char.

Character constants are represented by enclosing them in single quotes (apostrophes). For example, 'A', 'g', '%', etc. Note that '0' is the digit zero with an ASCII code of 48 and should not be confused with the NULL character having ASCII code 0 and represented by the escape sequence '\0'.

Escape sequences provide a convenient method for representing non-printing characters using printable characters. An escape sequence consists of a backslash (\) followed by one or more special characters. Note that, even though an escape sequence consists of more than one character, it is internally stored as a single character corresponding to the escape sequence. A list of escape sequences is given in Table 4.3.
Table 4.3 Escape Sequences

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>newline</td>
<td>NL</td>
</tr>
<tr>
<td>horizontal tab</td>
<td>HT</td>
</tr>
<tr>
<td>vertical tab</td>
<td>VT</td>
</tr>
<tr>
<td>backspace</td>
<td>BS</td>
</tr>
<tr>
<td>carriage return</td>
<td>CR</td>
</tr>
<tr>
<td>formfeed</td>
<td>FF</td>
</tr>
<tr>
<td>alert (bell)</td>
<td>BEL</td>
</tr>
<tr>
<td>backslash</td>
<td>\</td>
</tr>
<tr>
<td>question mark</td>
<td>?</td>
</tr>
<tr>
<td>single quote</td>
<td>'</td>
</tr>
<tr>
<td>double quote</td>
<td>&quot;</td>
</tr>
<tr>
<td>octal number</td>
<td>\ooo</td>
</tr>
<tr>
<td>hex number</td>
<td>\xhh</td>
</tr>
</tbody>
</table>

Note the escape sequence ‘\\’. We need a special escape for the backslash itself because, otherwise, the character ‘\’ might be misinterpreted as an incomplete escape sequence. Similarly, the single quote (‘‘), the double quote ("_) and the question mark (?) have a special meaning in the C language. Therefore, escape sequences are necessary to strip them of this special meaning and indicate the plain character.

The escape sequence ‘\0’ represents the null character (ASCII code 0) and should never be confused with the digit ‘0’ (ASCII code 48).

Characters can also be entered as an octal escape sequences consisting of three octal digits written as ‘ooo’. For example, ‘\013’ represents the vertical tab character and ‘\007’ the bell character.

Characters can also be entered as a hexadecimal escape sequences consisting of one or more hexadecimal digits (0...9, a...f, A...F) written as ‘\xhh’. For example, ‘\xb’ represents the vertical tab character and ‘\x7’ the bell character.

4.4 Simple Input/Output of Characters

Let us look at a simple program that performs the following tasks

- Outputs a message asking for input.
- Accepts input of a character from the user.
- Stores the value in a char variable named mynum.
- Outputs the value to the monitor as a character.
- Outputs the value to the monitor as an integer.

The last step will demonstrate the integer nature of a character variable.
Program 4.4 – IOCHAR.C  

/* ================================================================  IOCHAR.C ================================================================  
 * Program to input and output an character.  
 * ============================================================== */ 

#include <stdio.h> 

void main() 
{
  int mychar;
  printf("Enter a character : ");
  scanf("%c", &mychar);
  printf("The character you have entered is %c", mychar);
  printf("The ASCII code of character is %d", mychar);
}

In the printf statements, the %c is replaced by the value of the variable mychar. The specifier %c indicates to the computer that the value of mychar must be printed as a character. Printing the same character with a %d conversion specification prints out the ASCII value of the character. To obtain the unsigned ASCII value of a character, use %u in the last printf call.

4.5 Trigraphs and Wide Characters

Apart from the ASCII character set, the only other character set in significant use is the EBCDIC code (Extended Binary Coded Decimal Interchange Code) widely used on some of the older IBM computers. Appendix B lists the EBCDIC character set.

On some computer keyboards, particularly European language keyboards, a few of the characters used in the syntax of the C language are absent. To enable such users, a set of trigraph sequences have been defined. Each of this set is replaced with the equivalent character during the preprocessing stage.

Table 4.4 Trigraph sequences

<table>
<thead>
<tr>
<th>Trigraph</th>
<th>????</th>
<th>???/</th>
<th>???'</th>
<th>???(</th>
<th>???)</th>
<th>???!</th>
<th>???&lt;</th>
<th>???&gt;</th>
<th>???-</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replaced with</td>
<td>#</td>
<td>\</td>
<td>^</td>
<td>[</td>
<td>]</td>
<td></td>
<td></td>
<td></td>
<td>~</td>
</tr>
</tbody>
</table>

Another issue concerning the international use of the C language has been the difficulty of mapping various foreign languages to the 7-bit ASCII or an 8-bit extended ASCII character set. Some of the Asian languages have thousands of character symbols which just cannot be accommodated into an 8-bit representation. For such languages, a new data type named wchar_t (wide character type) has been included in the C language. Wide character constants are prefixed with an L, e.g., L'x'. The wide character type is an integral data type defined in the header file stddef.h.
4.6 Points to Remember

- The standard ASCII character set is a 7-bit code. Extended 8-bit ASCII character sets are usually not portable from one platform to another and should be avoided.
- C provides a one byte integer data type named char for storing characters.
- char may be signed char or unsigned char by default depending on the implementation.
- Escape sequences provide a convenient way to represent non-printable characters and special characters.
- Do not confuse the character NULL (\'0\' having ASCII code 0) with the character ZERO (\'0\' having ASCII code 48).

Review Quiz

1. What is the relationship between the ASCII code of an uppercase letter and its corresponding lowercase letter?
2. Which of the following are legal character constants?
   '0'  '\'  '''  '"'  '&'  'F'  'ag'  '\n'  '\z'

Programming Exercises

1. If you have access to a compiler working on an "IBM compatible", modify DRAWBOX.C to display various kinds of boxes. Try drawing double lined borders.
9

Strings

5.0 Lesson Goals

- To understand how strings are represented in C.
- To understand the relation between strings and the characters that make a string.

5.1 Strings in C

In C, a string constant is represented by a set of characters enclosed within double quotes (quotation marks), e.g., "Hello", "123", and "C Language" are all valid string constants. A string constant may contain one or more escape sequences (which are nothing but special characters), e.g., "\nHello", "\t\a\bFAT", and "c:\\temp" are all valid string constants. An empty string constant can be written as "".

The storage and internal representation of character strings differs widely from one computer language to another. When creating a program out of modules written in different high level languages, it is relatively easy to exchange information about integers, floating point numbers and characters between the different modules. But strings pose a considerable level of difficulty because of the different methods adopted for their storage. In C, a character string is a null terminated array of characters.

Let us look at this using some examples. First, we will look at how the string constant "Hello" is stored. It is stored in an array of 6 characters - five for each character of the string and the last character containing a null character ("\0") as shown in Figure 5.1.

Note how we number the characters as 0, 1, 2, 3, 4, and 5 and NOT as 1, 2, 3, 4, 5, and 6. As a good C programmer, it is imperative that you start counting from 0 and not from 1. Similarly, the string "c:\\temp" will be stored in 8 bytes as shown in Figure 5.1 - the escape sequence "\\" counting as only one character and an additional byte for the null character at the end.

How many bytes are required to store the empty string ""? One byte which is used to store the null character as shown in Figure 5.1.
5.2 Declaring Character Arrays

We will study arrays in Chapter 21 but right now we will pick up enough knowledge about arrays to be able to use strings. A character array is declared and a string constant is stored in it as follows:

```c
char string_name[n_chars] = string constant;
```

where `n_chars` is the number of characters that we need to store in the string constant. It is perfectly safe to ask for many more bytes than what is really required. On the other hand, the number of characters declared must NOT be less than the number required to store the string. For example,

```c
char greet[6] = "Hello";  // 6 chars stored, 6 bytes declared
char myname[25] = "John Doe";  // 9 chars stored, 25 declared
```

In the second example shown in Figure 5.2, all characters stored beyond byte number 8 are simply ignored. The string terminates (ends) at the first null character encountered in the character array. Everything after that is ignored. For example, look at the second character sequence shown in Figure 5.2. What is the string constant stored in this array? It is "John" and NOT "John Doe" because byte number 4 contains the null terminator.

```c
char greet[6] = "Hello";  // 6 chars stored, 6 bytes declared
char myname[25] = "John Doe";  // 9 chars stored, 25 declared
```

The character array must be large enough to hold the required number of characters plus one null termination character.
Note that all character arrays need not be valid strings. An array of characters may contain a set of characters without having a null character anywhere in it. Such a character array is not a valid string.

5.3 Simple Input/Output of Strings

Let us look at a simple program that performs the following tasks

- Outputs a message asking for input.
- Accepts input of a string from the user.
- Stores the value in a character array named mystr.
- Outputs the string to the monitor.

Program 5.1 – IOSTR1.C

```c
/* ============================  IOSTR1.C ==============================
Program to input and output a string.
=======================================================================*/
#include <stdio.h>
void main()
{
    char mystr[100];
    printf("Enter a string :");
    gets(mystr);
    printf("The string you have entered is \n%s",mystr);
}
```

Try to enter the following sentence as your input

India is a great country.

The first step is to declare a character array of 100 characters. Here, we are assuming that our string will not exceed 99 characters as we need to reserve one character for the null terminator. Notice that we use a new function this time, i.e., `gets`, to perform the input. Also, note that there is no ampersand (&) before the string name in the `gets` statement. Until we learn more about arrays and pointers, just remember the thumb rule that an & should NOT be placed in front of a character string name for input operations.

In the last `printf` statement, the %s is replaced by the value of the string `mystr` in the output.

If we replace the call to the `gets` function in IOSTR1.C with the following statement

```c
scanf("%s",mystr);
```

and execute the modified program, the program accepts only the first word (ending with a whitespace character) as the input string. If you use the example input given above, the string accepted is only "India". Remember this important difference between the input of a string using the `gets` function and the `scanf` function using a %s specification. We will look into this again in Chapter 11.
When you have to print only one string, you have two equivalent options which are shown below.

```c
printf("%s", "Hello");
printf("Hello");
```

The first style prints the string using a conversion specifier `%s`. The second uses the fact that the `printf` function expects a string as its first argument.

How can we print out the individual characters belonging to a string? For this, we need to remember that a string is simply a character array and then use the standard notation for an element of an array. An element of an array is specified by the name of the array and an index (also known as a subscript), indicating the position of the particular element, given inside brackets `[ ]`. Remember that, for an array of `n` elements, the valid indices range from `0` to `(n-1)`. ALWAYS BEGIN COUNTING FROM 0. This is shown in the program `INDCHAR.C` given below.

**Program 5.2 – INDCHAR.C**

```c
/* ========================== INDCHAR.C ========================
Printing individual characters of a string.
================================================================* /
#include <stdio.h>
int main(void)
{
    char mystr[40] = "Programming in the C Language";
    printf("\nCharacter  0 is %c", mystr[0]);
    printf("\nCharacter 1 is %c", mystr[1]);
    printf("\nCharacter 19 is %c", mystr[19]);
    return 0;
}
```

The output from this program is as follows.

Character 0 is P
Character 1 is r
Character 19 is C

### 5.4 Automatic Concatenation of Strings

Adjacent string constants are automatically concatenated and treated as a single string. This is demonstrated by the small program `ADJSTR.C` given below.
Program 5.3 – ADJSTR.C

Concatenation of adjacent strings

/* =============================== ADJSTR.C =============================
Program showing concatenation of adjacent strings constants.
====================================================================*/
#include <stdio.h>

int main(void)
{
    char mystr[60] = "One " "World!
";
    printf("%s", mystr);
    printf("Hello", " World"");
    printf("Hello, " "Again!");
    return 0;
}

The output from this program is shown below.

One World!
Hello, World
Hello, Again!

In the second printf, notice that even though we have whitespace in between the two strings, these are treated as adjacent strings because C ignores whitespace.

5.5 Points to Remember

• Strings are represented using null terminated arrays of characters, i.e., the null character ('\0' or ASCII code 0) is used to indicate the end of the string in an array of characters.

• Adjacent string constants are automatically concatenated.

Review Quiz

1. How many bytes will be required to store each of the following string constants?
   (a) "small"
   (b) "\apple"
   (c) "\near"
   (d) "\n\a\t"
   (e) "\xaf"
   (f) "\o123"
   (g) "\\\t"
   (h) "\\"

2. What will be the output from the following statement?
   printf("one\n%s\nthree","four");

Programming Exercises

**** to be added
Identifiers & Symbolic Constants

6.0 Lesson Goals

• To understand the rules for creating valid identifiers in C.
• To understand the use and benefits of symbolic constants.

6.1 Identifiers

Identifiers are the names given to various program elements like variables, functions, arrays, structures, etc. Every high level language has its own set of rules for the creation of valid identifiers. The identifiers in the C language must adhere to the following rules.

• An identifier can be made of letters, digits, and the underscore character '_'. The underscore character should not be confused with the dash character '-' which is NOT a valid character. This problem is confounded by the fact that both these characters are usually assigned to the same key on a standard computer keyboard.
• The first character can be a letter or the underscore character. It must NOT be a digit.
• It is good programming style not to begin identifier names with an underscore ('_') because names beginning with an underscore are usually reserved for system defined names. By not using the underscore character in the beginning of your identifiers, you will avoid any possible conflict with system defined names.
• Identifiers can be arbitrarily long but only 31 characters are significant on most machines.
• The case of the letters is significant. Therefore, the identifiers, STUDENT, student, and Student are all considered to be different identifiers.
• No whitespace is allowed inside an identifier.
• The identifier must not be a keyword used (reserved word) in the C language, i.e., it should not be any of the words shown in Table 6.1.
Table 6.1 List of Keywords in C

<table>
<thead>
<tr>
<th>auto</th>
<th>double</th>
<th>int</th>
<th>struct</th>
</tr>
</thead>
<tbody>
<tr>
<td>break</td>
<td>else</td>
<td>long</td>
<td>switch</td>
</tr>
<tr>
<td>case</td>
<td>enum</td>
<td>register</td>
<td>typedef</td>
</tr>
<tr>
<td>char</td>
<td>extern</td>
<td>return</td>
<td>union</td>
</tr>
<tr>
<td>const</td>
<td>float</td>
<td>short</td>
<td>unsigned</td>
</tr>
<tr>
<td>continue</td>
<td>for</td>
<td>signed</td>
<td>void</td>
</tr>
<tr>
<td>default</td>
<td>goto</td>
<td>sizeof</td>
<td>volatile</td>
</tr>
<tr>
<td>do</td>
<td>if</td>
<td>static</td>
<td>while</td>
</tr>
<tr>
<td>Possible implementation defined keywords (see your compiler manual for actual list)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ada</td>
<td>far</td>
<td>huge</td>
<td>pascal</td>
</tr>
<tr>
<td>asm</td>
<td>fortran</td>
<td>near</td>
<td></td>
</tr>
</tbody>
</table>

Let us take a look at some invalid identifiers and see why they are invalid.

1stname begins with a digit
last-name contains invalid character '-' (dash)
tax$ contains invalid character $
union reserved keyword

6.2 Naming, Declaring and Initializing Variables

Every function in C has two parts. There is a set of declarations followed by the executable statements. A major part of the declarations consists of declaration of variables. Variables are declared as follows

data type list of variable names;

The semi-colon at the end must be present to indicate the end of the declaration statement. For example, the following declaration declares four int variables named x, y, k, and count.

```c
int x, y, k, count;
```

The above declaration may be broken up into a set of equivalent declarations as shown below

```c
int x;
int y;
int k, count;
```

There can be any number of declaration statements in a function.

An small but important aspect of designing good C programs is the choice of names. **Names of variables should be self-explanatory** as far as possible. A good choice of variable names goes a long way in improving the readability of a program. For example, let us say that we have been asked to write a program to calculate the force applied using Newton's law, i.e., force equals mass times acceleration. Then, we could declare the following variables.

```c
float F, m, a; /* F is force, m is mass, a is acceleration due to gravity*/
```
\[ F = m \times a; \]

\( F, m, \) and \( a \) are perfectly valid identifiers according to the rules of forming identifiers but the above choice does not lead to a very readable program. It is customary to reserve uppercase identifiers for things like symbolic constants and structures. The name \( F \) does not follow this convention. Longer, self-explanatory, lowercase names are preferable. A better choice of names is shown below

```c
float force, mass, accel_gravity;
accel_gravity = 9.81;
force = mass \times \text{accel} \_ \text{gravity};
```

This choice of names could make it unnecessary to add any explanatory comments to the program! For you as the programmer, longer names means more typing but the extra effort pays off in the long run by adding to the quality of your C code.

While naming your variables, avoid use of the abbreviation "no" to indicate "number". For example, a variable named \( \text{no} \_ \text{students} \) can easily be misinterpreted to mean that "there are no students in . . .". A name like \( \text{num} \_ \text{students} \) is much better and less likely to be misinterpreted.

When a variable is declared, its value is unknown. The memory allocated to this variable will contain some unknown binary pattern and using this in some calculations, even by mistake, could lead to unpredictable and nonrepeatable results. Therefore, it is a good idea to assign an initial value to the variables right at the time of declaration. For example,

```c
float force=0.0, mass=0.0, accel_gravity =9.81;
force = mass \times \text{accel} \_ \text{gravity};
```

Initialization during declaration can sometimes eliminate an assignment later in the program as shown in the above example for the variable \( \text{accel} \_ \text{gravity} \). Initialization during declaration is not compulsory but simply a part of good programming practice. But if the program is very small in size or the variables are initialized at a later part in the program, you may choose not to initialize the variables at the time of declaration.

### 6.3 Symbolic Constants

A symbolic constant is a name that is used in place of a constant (numeric, character, or string). It is usually defined at the beginning of a program using a pre-processor directive as shown below

```c
#define name text
```

For example,

```c
#define PI 3.141592
#define CITY "NEW DELHI"
#define TRUE 1
```
Any valid identifier can be used as a symbolic constant. However, \textbf{it is customary to use uppercase names for symbolic constants.} Let us look at a few lines from a program containing the above symbolic constants.

\begin{verbatim}
printf("\nThe value of PI is %f",PI);
printf("\nThe CITY is %s",CITY);
printf("\nThe value is %d",TRUE);
\end{verbatim}

At the end of preprocessing, every occurrence of the name is replaced by the text \textit{except when the name of the symbolic constant appears inside a string constant}. This exception must be kept in mind when using symbolic constants. In effect, what the computer executes is the following code.

\begin{verbatim}
printf("\nThe value of PI is %f",3.141592);
printf("\nThe CITY is %s","NEW DELHI");
printf("\nThe value is %d",1);
\end{verbatim}

Notice that PI remains unchanged inside the string in the first line and the word CITY remains unchanged inside the string in the second line.

If the preprocessor simply changes a symbolic constant to its equivalent text, why use a symbolic constant at all?
\begin{itemize}
  \item The use of symbolic constants makes the program more readable. For instance, PI is definitely more readable and a more intuitive symbol than 3.141592.
  \item The use of symbolic constants makes the program easier to maintain, i.e., makes it easier to make changes correctly.
\end{itemize}

Let us look at the maintenance of a small program that contains the following lines (this is not a complete program).

\begin{verbatim}
int  physmark[100], chemmark[100], mathmark[100];
float avgmarks[100];
...
for(j=0; j<100; j++)
  printf("For Physics, the average marks of 100 students is %f",pavg);
\end{verbatim}

This program is designed to process the marks of 100 students. What happens when the number of students increases to 150? We will have to edit the program and change all occurrences of 100 (shown in bold) to 150. That works out to 6 changes with the accompanying danger that, if we forget to change even one of them, the program will stop functioning correctly. The same could be written using a symbolic constant \texttt{NSTUD} as follows.

\begin{verbatim}
#define NSTUD 100
int  physmark[NSTUD], chemmark[NSTUD], mathmark[NSTUD];
float avgmarks[NSTUD];
for(j=0; j< NSTUD; j++)
  printf("For Physics, the average marks of %s students is %f", NSTUD,pavg);
\end{verbatim}
Now, every time the number of students changes, we need to make only one change in the program, i.e., change the text for the symbolic constant from 100 to whatever the new number may be. We no longer have to worry about losing the integrity of the program due to some omitted change. In this way, symbolic constants add to the **maintainability** of a C program.

You must always keep in mind the "text-substitution" aspects of symbolic constants. For example, look at the following lines

```c
#define PI 3.141592;
area = PI * r * r;
```

After preprocessing, this becomes the invalid statement shown below and generates error mesages.

```c
area = 3.141592; * r * r;
```

The bug is caused by the semi-colon placed at the end of the symbolic constant. We will consider more examples of such harmful side effects of preprocessor statements in Chapter 25.

**Whitespace before text value of the symbolic constant and after it are both ignored.** A symbolic constant basically represents a piece of text and, therefore, it has no data type associated with it.

### 6.4 Points to Remember

- A good naming style is an essential part of good programming style.
- Symbolic constants can be used to improve the maintainability and readability of programs.

---

Review Quiz

Programming Exercises
Integer Arithmetic

7.0 Lesson Goals

- To learn about the assignment operator
- To learn about integer arithmetic using the binary addition, subtraction, multiplication, division and remainder operators.
- To understand how integer arithmetic is different from the usual arithmetic.
- To learn about the increment and decrement operators.
- To learn the use of shorthand assignment operators.

7.1 The Assignment Operator

In this chapter, we will look at integer arithmetic in the C language but before doing that, we need to look at the assignment operator represented by the symbol \( = \). This symbol which denotes equality in arithmetic has a very different meaning in the C language.

To begin with, \( = \) is an operator, i.e., it does something. What does it do? An assignment statement in C is shown below.

\[
\text{variable} = \text{expression}
\]

which can be read as "assign the value of the expression to the variable". The assignment operator takes the value of the expression and puts it into the memory location of the variable. Let us look at some simple examples.

\[
x = 4
\]

Here the old value of \( x \) is replaced by a value of 4. The old value, whatever it might have been, is erased. This is shown in Figure 7.1.
Let us consider the other example shown in Figure 7.1,

\[ y = x \]

The existing value of \( y \) is replaced with the existing value of \( x \). The old value of \( y \) is erased.

A small but extremely important algorithm is based on the behavior of the assignment operator. This is the algorithm involved in swapping two numbers. Let us consider two integers \( x \) having a value of 7 and \( y \) having a value of 11. We want to interchange the two values such that at the end of the swapping, \( x \) must have a value of 11 and \( y \) must have a value of 7. Will the following operations do the job?

\[
\begin{align*}
  y &= x \\
  x &= y \\
  y &= \text{temp}
\end{align*}
\]

No! At the end of the first assignment statement, both variables have a value of 7 and this is not changed by the second assignment operation. How do we get this right? Think of a cup X containing water and a cup Y containing milk. How can we interchange the contents of the two? We will need a third cup. First, we pour the water into the third cup. Next, we pour the milk from cup Y into cup X and, finally, pour the water in the third cup into cup Y. In a similar fashion, we need a temporary variable to hold the contents of one of the variable, say \( \text{temp} \). Now, we can write

\[
\begin{align*}
  \text{temp} &= x \\
  x &= y \\
  y &= \text{temp}
\end{align*}
\]
Values can be assigned to a character variable using either the character constant or the equivalent ASCII code. For example, if `mychar` is a `char` variable, then

```c
mychar = 'P';  // more readable and portable version

and

mychar = 80;
```

are both equivalent statements. However, from the readability viewpoint, the first statement is **definitely preferred** as the user does not have to search for a table of ASCII codes. The first statement is also more portable because, even if we port our program to a program that uses a non-ASCII code (e.g., EBCDIC character set) for character representation, this statement will continue to work fine. The character 'P' will be changed to whatever integer is used for the character which may not be 80. Another example is shown below

```c
mychar = 0;
```

is equivalent to

```c
mychar = '\0';
```

Owing to the special nature of strings in C, **IT IS ILLEGAL TO ASSIGN A STRING CONSTANT TO A STRING VARIABLE.**

```c
mystr = "hello";  // ILLEGAL STATEMENT
```

We will gain a better understanding of the reasons behind this when we learn about pointers in Chapter 23.
7.2 Arithmetic Operators

The arithmetic binary operators are as shown below.

+  addition operator
-  subtraction operator
*  multiplication operator
/  division operator
%  remainder or modulus operator

The first three operators, as applied to integers, need no special explanation as they have the usual mathematical significance. They all give exact integer results as long as the bounds of the particular data type are not exceeded. The following program takes two integers as input from the user and prints their sum, difference, and products.

Program 7.1 – ARITH1.C Simple integer arithmetic

```c
/* ============================  ARITH1.C ==============================
Program to try out simple integer arithmetic operations.
=======================================================================*/
#include <stdio.h>
void main()
{
    int x,y;
    printf("Enter the first integer : ");
    scanf("%d",&x);
    printf("Enter the second integer : ");
    scanf("%d",&y);
    printf("The two integers are %d and %d
",x,y);
    printf("x+y = %d",x+y);
    printf("x-y = %d",x-y);
    printf("x*y = %d",x*y);
}
```

If the result of an arithmetic operation exceeds the limits of that particular unsigned data type, no error is reported and the most significant bits of the result are lost. For example, if an unsigned int which is assigned a storage of 2 bytes in a particular implementation, then the largest storable integer value is 65535 (=FFFF in hex). The operation 65535 + 1 should yield the result (10000)16. However, since there are only two bytes for the storage, the result become (0000)16!

The result of a similar problem on signed integral constants is implementation dependent. You can run the following program, CHEKOVFL.C to determine the effect of such an operation.
Program 7.2 – CHEKOVFL.C  Overflow behavior of Integer data types

/* ================================== CHEKOVFL.C ==================================
checking overflow behavior of integer data types.*/
#include <stdio.h>
#include <limits.h>
int main(void)
{
    printf("Common to all implementations\n");
    printf("USHRT_MAX=%hd, USHRT_MAX+1=%hd",USHRT_MAX,USHRT_MAX+1);
    printf("UINT_MAX=%d, UINT_MAX+1=%d",UINT_MAX,UINT_MAX+1);
    printf("Implementation dependent behavior\n");
    printf("SHRT_MAX=%hd, SHRT_MAX+1=%hd",SHRT_MAX,SHRT_MAX+1);
    printf("INT_MAX=%d, INT_MAX+1=%d",INT_MAX,INT_MAX+1);
    return 0;
}

The output from this program is shown below.

Common to all implementations
USHRT_MAX=-1, USHRT_MAX+1=0
UINT_MAX=-1, UINT_MAX+1=0

Implementation dependent behavior
SHRT_MAX=32767, SHRT_MAX+1=-32768
INT_MAX=32767, INT_MAX+1=-32768

Owing to this dangerous habit of ignoring integer overflow seen in most implementations, we must be exceptionally careful in making sure that the result of any arithmetic operation does not exceed the limits of the particular data type in use.

7.3 Integer Division

Integer division in a C program is very different from the usual mathematical operation. When an integer is divided by another, any fractional part of the resultant quotient is truncated (i.e., chopped off) to yield an integer result. For example,

6/4 results in 1 (1.6666... after truncation)
6/3 results in 2 (no fractional part in result)
7/10 results in 0 (0.7 after truncation)

Owing to this unusual effect of integer division, the mathematical identity, \((a/b) \times b = 1\), is seldom true in integer calculations!

There are other pitfalls in integer division. When the division involves one negative integer and one positive integer, ANSI C leaves the result to be implementation dependent. For example, \(-5/3\)
may yield a result of $-1$ on some computers and $-2$ on some computers! This can pose a serious portability problem for programs that perform division operations involving one negative integer and one positive integer.

The **modulus or the remainder operator**, as the name suggests, yields the integer remainder left after division. \( a \% b \) gives the remainder obtained after dividing \( a \) with \( b \). For example,

\[
\begin{align*}
6 \% 4 & \quad \text{results in 2} \\
6 \% 3 & \quad \text{results in 0} \\
7 \% 10 & \quad \text{results in 7}
\end{align*}
\]

The operation \( x \mod y \) is read as "\( x \mod y \)."

Just like division, **when the remainder operation involves one negative integer and one positive integer, ANSI C leaves the result to be implementation dependent.** For example, \(-5 \% 3\) may yield a result of $-2$ on some computers and $+1$ on some computers. In spite of this implementation dependency of the division and the remainder operations, the following identity is always satisfied.

\[
a - ((a/b)\times b) \quad \text{is equal to} \quad (a \% b)
\]

For example, if \( a \) is $-5$ and \( b \) is $+3$, then \((a\%b)\) is either

\[
-5 - (-1 \times 3) \quad \text{which yields} \quad -2
\]

or

\[
-5 - (-2 \times 3) \quad \text{which yields} \quad +1,
\]

depending on whether \((a/b)\) equals $-1$ or $-2$. Type and execute the following program, ARITH2.C, with various values of \( x \) and \( y \) to learn more about the division and the modulus operators.

**Program 7.3 – ARITH2.C**

```c
/*
 * ARITH2.C
 *
 * Program to try out integer division and remainder operations.
 */
#include <stdio.h>
void main()
{
    int x,y;
    printf("\n Enter the first integer : ");
    scanf("%d",&x);
    printf("\n Enter the second integer : ");
    scanf("%d",&y);
    printf("\n The two integers are %d and %d\n",x,y);
    printf("\n x/y = %d",x/y);
    printf("\n x\%y = %d",x\%y);
    printf("\n x - ((x/y)\*y) = %d",x -((x/y)*y));
}
```

A sample output from this program is shown below.
Enter the first integer: 45
Enter the second integer: 10
The two integers are 45 and 10
\[ \frac{x}{y} = 4 \]
\[ x \% y = 5 \]
\[ x - \left( \frac{x}{y} \right) y = 5 \]

Notice the use of \( x \% y \) (shown in bold) to print \( x \% y \) in the output. Usually, in a printf function, the \( \% \) symbol followed by a character indicates a conversion specification (e.g., \( \%d \), \( \%c \), etc.). To remove this special meaning for the \( \% \) symbol, we need to write \( \%\% \).

### 7.4 Shorthand Assignment Operators

An expression like,

\[ x = x \times 2 \]

in which a variable \( x \) appears in the right hand expression as well as being the assignment variable, is very common in computer programs. The C language provides for a special shorthand notation that eliminates the need to use the variable name twice in this expression. It can be written as

\[ x *= 2 \]

with no whitespace between the * and the =. Similarly, other operators exist for the remaining arithmetic operators. For example,

\[ x += a \quad \text{is equivalent to} \quad x = x + a, \]
\[ x -= 2 * p \quad \text{is equivalent to} \quad x = x - 2 * p, \]
\[ x /= (m+2) \quad \text{is equivalent to} \quad x = x / (m+2), \]
\[ x %= 2 \quad \text{is equivalent to} \quad x = x \% 2. \]

The shorthand operators lead to faster and more readable code.

### 7.5 The Increment and Decrement Operators

Very often, while writing a program, we need to increase or decrease an integer variable by 1. For example, if we are processing students records, we need a variable which can be used as a counter to keep track of the number of records already processed. After processing the records of the current student, this counter needs to be incremented by one. The **increment operator** ++ and the **decrement operator** -- (no whitespace is allowed between the two characters), allow us to perform such operations elegantly. They are both examples of **unary operators**, i.e., operators which have only one operand.

The increment operator increases the value of the variable by 1 (e.g., \( k++ \) increase the value of \( k \) by 1) and the decrement operator decreases the value of the variable by 1. Each of these can be used in two different ways, i.e., as a prefix operator or as a postfix operator. Let us look at some examples shown in Table 7.1.
Table 7.1  Some examples of the increment operator
(Assume \( j = 3 \) at the beginning of each example)

<table>
<thead>
<tr>
<th>Statements</th>
<th>Equivalent Version</th>
<th>Final Value of ( j )</th>
<th>Final Value of ( k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ( k = ++j; ) ( j = j+1; ) ( k = j; )</td>
<td>( j = j+1; ) ( k = j; )</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>2. ( k = j++; ) ( k = j; ) ( j = j+1; )</td>
<td>( j = j+1; ) ( k = j; )</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3. ( k = j+1; ) ( k = j+1; ) ( j = j+1; )</td>
<td>( j = j+1; ) ( k = j; )</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4. ( ++j; ) ( k = j; ) ( j = j+1; ) ( k = j; )</td>
<td>( j = j+1; ) ( k = j; )</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5. ( j++; ) ( k = j; ) ( j = j+1; ) ( k = j; )</td>
<td>( j = j+1; ) ( k = j; )</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

In Example 1 given in Table 7.1, the prefix ++ indicates a pre-increment operation. Therefore, the value of \( j \) is incremented by 1 and the resulting value of 4 is assigned to \( k \).

In Example 2, the postfix ++ indicates a post-increment operation. Therefore, the value of \( j \) is assigned to \( k \) and after this statement has been executed, \( j \) is incremented by 1.

In Example 3, there is no increment operation on the variable \( j \) and, therefore, it remains unchanged. Notice, how this is different from Example 1 where the value of \( j \) is changed.

In Examples, 4 and 5 given in Table 7.1, the incrementing is performed as a separate statement and, therefore, there is no special significance for the placement of the increment operator. Let us look at a more complex example.

\[
j = 3;
\]
\[
m = 7;
\]
\[
k = (++j) + (m++);
\]

The last statement (shown in bold) is equivalent to the following three statements

\[
j = j+1;
k = j + m;
m = m+1;
\]

and, therefore, the final values of \( j \), \( k \), and \( m \), are 4, 11, and 8, respectively. The use of the increment and decrement operators within complex expressions is difficult to understand at first glance and, therefore, in the interest of creating more readable programs, it is better to write them as separate statements. The set of statements

\[
++j;
k = j + m;
m++;
\]

should be preferred to the previously shown cryptic form.
Notice that when used on a single variable in a separate statement, the post and pre operators are exactly equivalent, i.e., the statement

```c
++p;
```

is equivalent to

```c
p++;
```

and

```c
--p;
```

is equivalent to

```c
p--;
```

### 7.6 Arithmetic with Characters

In Chapter 4, we have seen the inherent integer nature of characters. This integer nature allows us to perform arithmetic operations on characters and provides for great elegance in the manipulation of characters. For example, we can subtract a character from another character.

```c
'a' - 'A' yields a value of 32
```

because \((\text{ASCII code of } 'a') - (\text{ASCII code of } 'A')\) equals \((97 - 65)\)

To convert an uppercase character to lowercase, we simply add 32 to the character

```c
'Q' + 32 yields the character 'q'
```

ASCII code of \('Q' + 32\) which is \(81 + 32\) equals 113, i.e., ASCII code of \('q'\).

The following program, CHARMATH.C, illustrates some of these features of performing integer arithmetic operations on character variables.
/* Program 7.4 – CHARMATH.C Integer arithmetic on characters */
#include <stdio.h>
void main()
{
    char mychar='A';
    printf("Enter an uppercase character : ");
    scanf("%c",&mychar);
    printf(" The character you have entered is %c",mychar);
    printf(" mychar-'A' = %d",mychar-'A');
    printf(" %c is character %d of the uppercase alphabet", mychar,mychar-'A'+1);
    printf(" lowercase equivalent is mychar + 32= %c",mychar+32);
    printf(" next character is mychar + 1 = %c",mychar+1);
}

7.7 Points to Remember

• Most compilers do not give any warning about overflow occurring during integer arithmetic.
• Any fractional remainder is truncated off during integer division.
• Division and remainder operation between one positive operand and one negative operand will give implementation dependent results.
• The post- and pre-increment operators have very different behavior when used as a part of a statement.
• Shorthand assignment operators should be used wherever possible.

Review Quiz

What will be the output of the following program:
main()
{
    int a,b,c;
    a = 100;
    b = ++a;
    c = b++;
    printf("a=%d\n",a);
    printf("b=%d\n",b);
    printf("c=%d\n",c);
}

- 2 * - 3 / 4 % 5 - - 6 + 4
Ans 11

int i,j,k;
i = j = k = 1;
Integer Arithmetic

\[ i -= -j-- - --k; \]
\[ y = y + 1; \]
\[ z = x + y; \]
\[ x = x + 1; \] as one statement

**Programming Exercises**
More About Arithmetic

8.0 Lesson Goals

8.1 Floating Point Arithmetic

The addition, subtraction, multiplication, and division operators have their usual mathematical functionality with respect to floating point numbers also. We must not, however, forget the implications of the finite precision representation scheme which were discussed in Chapter **. For example, even with floating point data types, it would be unrealistic to expect that \((a/b)\times b\) will yield 1 because of roundoff error.

The modulus operator, \%, CANNOT be used for floating point data types. The increment and decrement operators can be applied to the floating point data types also.

Many high level languages have an exponentiation operator. Unlike many high level languages, C does not have an exponentiation operator. You need to use the function pow from the standard math library to perform exponentiation. This function pow takes two arguments of type double. For example, to find 2.3 raised to the exponent 3.6, we can use the following lines in the program

```c
#include <math.h>

double result;
result = pow(2.3, 3.6);
```

or alternatively,

```c
#include <math.h>

double x=2.3, y=3.6, result;
result = pow(x, y);
```

To specify the use of the standard mathematical function library, we must include the math.h header file. The use of another mathematical function, the sqrt function which takes one argument of the type double and calculates the square root, is illustrated below.
More About Arithmetic

#include <math.h>
double y = 2.0, s_root_y;
s_root_y = sqrt(y);

Execute the following program, ARITH3.C, to look at the execution of common arithmetic operators.

Program 8.1 – ARITH3.C Floating point arithmetic

/* ============================ ARITH3.C ==============================
* Program to try out simple floating arithmetic operations.
=====================================================================*/
#include <stdio.h>
void main()
{
    float x, y;
    printf("Enter the first real number : ");
    scanf("%g", &x);
    printf("Enter the second real number : ");
    scanf("%g", &y);
    printf("The two numbers are %g and %g\n", x, y);
    printf("x+y = %g", x + y);
    printf("x-y = %g", x - y);
    printf("x*y = %g", x * y);
    printf("x/y = %g", x / y);
}

8.2 Arithmetic Conversions

Before a binary operator can perform its operation, it is required that the two operands be of the same data type. If the two operands are not of the same data type, one of them is converted according to a set of rules called the usual arithmetic conversions. For more details on these rules, refer to K & R.

In a nutshell, what these rules specify is that the operand having lower precision or range of values is converted to the data type of the other operand, i.e., the lower type is promoted to the higher type. The long double is considered the highest type and the char is considered to be the lowest type as shown in Figure 8.1.
For example, if an addition involves a float value and a double value, the float value is first converted to double. If a multiplication involves one float and one int value, the int value is first promoted to a float. Let us look at the evaluation of an arithmetic expression

\[(\text{double}) + (\text{float}) \times (\text{int})\]

The multiplication operator promotes the int value to a float and the result of the multiplication is a float type value. Next, when this float type value has to be added to the double type value, the float value is promoted to double before the addition is performed.

Owing to these rules of type conversion, various arithmetic operations may not exhibit their usual properties like associativity. For example,

\[15.0 \times (2/3) \text{ yields a result of 0.0}\]

whereas

\[(15.0 \times 2)/5 \text{ yields a result of 6.0.}\]

In the second case, 2 is converted to the type double before the multiplication is performed.
8.3 Conversion during Assignment

Another important kind of type conversion is associated with the assignment operator. When the data type of the expression on the right hand side of an assignment is not the same as the data type of the variable to which it is being assigned to, the value of the expression is forcibly converted to the data type of the variable. For example, when an integer type variable is assigned a floating point value, the floating point value is truncated. Care must be taken to ensure that the resultant value after such a truncation can be accommodated in the integer type. For example,

```c
int k;
k = 34.56e23; /* error : overflow will occur here */
```

will result in an error because the maximum value of the int type will be exceeded when the value 34.56e23 is converted to an integer.

What is the final result of an assignment expression, i.e., what is the value of \(( p = q)\)? The value of an assignment expression is the value of the variable on the left hand side of the assignment expression. This allows us to write assignment statements like the following

```c
a = b = c = d = 8;
```

as the equivalent of the four following assignments

```c
d = 8;
c = d;
b = c;
a = b;
```

8.4 Rounding Off

Rounding off a floating point value to its nearest integer is a pretty common task and here we shall look at this problem in some detail. For the time being, we will consider only positive values. To round off a positive fraction, we need to

- find the whole number part of the real number
- find the fractional part
- add one to the whole number part if the fractional part is more than 0.5
- leave the whole number part unchanged is it is less than or equal to 0.5.

This can easily be achieved by the steps shown in the following program segment.

```c
float x;
int rounded;
rounded = x + 0.5;
```
For example, if \( x \) is 4.3, then \( x + 0.5 \) yields 4.8 which when truncated yields 4. If \( x \) is 4.65, then \( x + 0.5 \) yields 5.15 which when truncated yields 5. This is a clever use of the truncation effect of converting a floating point value to an integer.

How do we round off to the nearest thousand? For example, 4356 should yield 4000 after such a rounding off process and 4578 should yield a value of 5000. We can use the following steps.

```c
float x;
int rndto1000, temp;
... temp = (x/1000) + 0.5;
rndto1000 = temp * 1000;
```

The variable `temp` is used to store the rounded off number of thousands.

### 8.5 Typecasting

Occasionally, we need to forcibly change the data type of an expression. For this, we can use the typecast operator as shown below.

```
(data type) expression
```

The type cast operator is a unary operator as it requires only one operand. For example, if we need to find the square root of an integer value using the square root function, we can typecast the integer into a `double`.

```c
int k=5;
double rootval;
rootval = sqrt( (double) k);
```

The typecast operator (double) converts the integer value of `k` into the type double. Note that the data type of `k` remains unchanged as `int`. Let us look at another example,

```c
float x = 7.8;
int krem;
krem = ( (int) x) %2;
```

Since, we cannot use the remainder operator with a `float` value, we type cast the value into an `int`. Note that the value of `x` remains unchanged as 7.8 by the type cast operation. The type cast operator yields an `int` value of 7 which leads to a value of 1 for `krem`.

### 8.6 The Unary Minus Operator

A minus sign placed before a numerical constant, variable or expression changes the algebraic sign of the operand. Note that, even though they share the same symbol, i.e., `-`, the minus operation is distinctly different from the subtraction operation which is a binary operation requiring two operands. Some examples of the use of the unary minus are shown below

\[-x \quad -(x+a) \quad -345\]
8.7 The sizeof Operator

The sizeof operator is used to determine the implementation dependent storage requirements of the various data types. In Chapter 2 and 3, we had seen how to find this information using the symbolic constants in the header files LIMITS.H and FLOAT.H. The sizeof operator is a more elegant method to find the exact number of bytes required to store a variable of a particular data type. The sizeof operator preceded either a variable name OR a type cast. For example, if x has been declared to be a float variable, then both the expressions given below will yield the number of bytes required to store a float variable.

\[
\text{sizeof } x \quad \text{using sizeof with a variable name} \\
\text{sizeof } (\text{float}) \quad \text{using sizeof with a typecast}
\]

The program, SIZES.C given below will provide the information about the storage of various data types on your C compiler.

**Program 8.2 – SIZES.C Using sizeof operator to find sizes of data types**

```c
/* ============================  SIZES.C ============================== 
   * Using the sizeof operator.                                      
   * =================================================================*/
#include <stdio.h>
void main()
{ 
    printf("\nOn this C compiler,\n\n    for long double is %d",sizeof(long double)); 
    printf("\n    for double      is %d",sizeof(double)); 
    printf("\n    for float       is %d",sizeof(float)); 
    printf("\n    for long int    is %d",sizeof(long)); 
    printf("\n    for int         is %d",sizeof(int)); 
    printf("\n    for short int   is %d",sizeof(short)); 
    printf("\n    for char        is %d",sizeof(char)); 
}
```

8.8 Points to Remember

**Review Quiz**

**Programming Exercises**
13

Writing a Complete C Program

9.0 Lesson Goals

9.1 Elements of a Simple C Program

From the C programs that were used in the last few chapters, we can summarize the following essential elements of a C program in the following skeletal form.

```
#include header files for necessary libraries
void main()
{
    declarations of variables
    executable statements
}
```

Note that all variables must be declared before the first executable statement of the C program. In this chapter, we will look at some simple programming problems. In each case, we will look at the steps involved in designing a solution to the problem using the C language. The steps common to all these programs will be as follows.

- Identify the variables required for the program. Choose an appropriate data type for each variable. Create a good name for the variable.
- Design the input/output required by the problem. We will try to create a friendly sequence of input/output statements for the ease of the user.
- Translate the task into simple statements in C language using what we have learnt so far.
- Look at methods to improve the efficiency of the program.

9.2 Mini-Problem 1 – Fancy Text Output

Problem Statement: Write a simple C program to write the following message on the screen.
Let us begin by choosing a good name for the program. A good name should indicate the function of the program in a shorthand fashion. We will call it SAYBYE.C.

What do we see in the given output? It has 7 lines of text to be printed. We can use one call to the printf function for each line. Therefore, we must include the stdio.h header file.

Do we need any variables? No, because there is nothing that requires storing in the computer memory. The program SAYBYE.C is given below.

Program 9.1 – SAYBYE.C Printing simple patterns using printf
/* ============================  SAYBYE.C ============================
Program to print BYE on the monitor screen in large size.
===================================================================*/
#include <stdio.h>
void main()
{
    printf("\n *****   *   *  ******");
    printf("\n *    *  *   *  *     ");
    printf("\n *    *   * *   *     ");
    printf("\n *****     *    ****  ");
    printf("\n *    *    *    *     ");
    printf("\n *    *    *    *     ");
    printf("\n *****     *    ******");
}

9.3 Mini-Problem 2 - Adding Two Fractions
Problem Statement: Write a program that will add two positive fractions, a/b and c/d, and print the result as a proper fraction. Output from this program must resemble the one shown below.
The sum of 2/3 and 5/4 equals 23/12.

First, we decide to call our program FRACTADD.C.

Next, we note that we need two integer variables for each fraction, one to store the numerator and another to store the denominator. Therefore, we need six integer type variables for this problem. Since, we have not been informed about the range of the integer values that will be input, we will use the data type int.

We need to read in the numerators and denominators of the two fractions and the final result should look as shown above.
Writing a Complete C Program

The major steps in this C program are as follows:

1. Declare six int variables - num1, den1, num2, den2, numsum, densum
2. Read in numerator of first fraction (i.e., we need to include stdio.h header file)
3. Read in denominator of first fraction
4. Read in numerator of second fraction
5. Read in denominator of second fraction
6. Calculate the numerator of the result as (num1*den2 + num2*den1)
7. Calculate the denominator of the result as (den1*den2)
8. Write the result in the format shown above.

The above outline represents a pseudocode version of our program, i.e., it is not a real program but gives detailed guidelines for the program.

Program 9.2 – FRACTADD.C
Addition of two fractions
/*
 * FRACTADD.C
 * Program to add two positive fractions.
 */
#include <stdio.h>

void main()
{
    /*declare variables*/
    int num1, num2, numsum, den1, den2, densum;

    /*Read in first fraction*/
    printf("Enter numerator of first fraction : ");
    scanf("%d",&num1);
    printf("Enter denominator of first fraction : ");
    scanf("%d",&den1);
    printf("The first fraction is %d/%d\n",num1,den1);

    /*Read in second fraction*/
    printf("Enter numerator of second fraction : ");
    scanf("%d",&num2);
    printf("Enter denominator of second fraction : ");
    scanf("%d",&den2);
    printf("The second fraction is %d/%d\n",num2,den2);

    /*Calculate resulting fraction*/
    numsum = num1*den2 + num2*den1;
    densum = den1 * den2;

    /*Write the result*/
    printf("The sum of %d/%d and %d/%d equals %d/%d.\n", num1,den1,num2,den2,numsum,densum);
}

You will notice that this program does not reduce the result to the lowest terms. We will save this modification for a later chapter.

9.4 Mini-Problem 3 - Converting a Length
Problem Statement: Given a length in a whole number of inches, write a program to convert this length into yards, feet, and inches. For example, given a length of 102 inches, we must get

102 inches equals 2 yards, 2 feet, and 6 inches.

We choose to call the program LENCONV.C.

We will need one variable to store the length in inches input by the user. Since, the length will be in a "whole number of inches", we can use an int type variable. We will also need 3 more int variables to store the number of yards, number of feet, and number of inches. We will use the following names: len, n_yards, n_feet, n_inch.

Next, how do we convert a length of 102 inches into yards, feet and inches. We first need to know that 1 yard = 3 feet, 1 foot = 12 inches, and 1 yard = 36 inches. Therefore, dividing the len by 36 should yield the number of yards because division of two integers truncates the decimal part automatically. Get a calculator in hand and begin calculating.

\[ \frac{102}{36} = 2.8333 \text{ which is 2 after truncation.} \]

The remainder after subtracting 2 yards needs to be converted to feet. How do we find the remainder? We could calculate it as

\[ \text{remainder} = 102 - (36 \times 2) = 30 \]

but a more elegant method is to use the remainder operator.

\[ \text{remainder} = 102 \mod 36 = 30 \]

We will need an additional int variable to store this remainder. We will use the name remainder for this variable. How many feet are there in 30 inches? Repeating the above operations again, we get

\[ \text{feet} = \text{remainder} / 12 \]
\[ \text{inches} = \text{remainder} \mod 12 \]

The major steps in this C program are as follows:

1. Declare 5 int variables - len, n_yards, n_feet, n_inch, remainder
2. Read in value of len (Therefore, we need to include stdio.h header file)
3. Compute n_yards = len / 36
4. remainder = len % 36
5. n_feet = remainder / 12
6. n_inch = remainder % 12
7. Write the output in desired format

Program 9.3 – LENCONV.C

Unit conversion of length

/* .................................................. LENCONV.C ..................................................
Program to convert a length in inches into yards, feet & inches */
```c
#include <stdio.h>

void main()
{
    /* declare variables */
    int len, n_yards, n_feet, n_inch, remainder;

    /* read in value of len */
    printf("Enter len in inches (whole number) ":);
    scanf("%d", &len);
    printf("The length you have entered is %d inches", len);

    /* Perform the calculations */
    n_yards = len / 36;
    remainder = len % 36;
    n_feet = remainder / 12;
    n_inch = remainder % 12;

    /* Write the output */
    printf("%d inches equals %d yards, %d feet & %d inches.", len, n_yards, n_feet, n_inch);
}
```

You will notice that our program is liable to give ungrammatical output at times, e.g.,

49 inches equals 1 yards, 1 feet, and 1 inches.

### 9.5 Mini-Problem 4 - Temperature Conversion

Problem Statement: Given a temperature in degrees Fahrenheit, write a C program to convert it into degrees Celsius using the formula given below.

\[ C = \frac{5}{9} \times (F - 32) \]

We will call this program FTOC.C.

What are the variables that we need for this program? We will need one variable to store the temperature in Fahrenheit input by the user and another to store the converted temperature in Celsius. The presence of fractions in the equations suggests that we will need a floating point data type for these variables. It is also obvious that a very high precision is not necessary for this problem. Therefore, we choose `float` type variables.

Next, we need to examine the formula for conversion. If you use the formula as it is given above, the answer will always be 0 degrees Celsius. Why? Because \(\frac{5}{9}\) equals 0. There are several ways to rewrite the above equation such that all calculations are performed on floating point type data. For example,

\[
\frac{5.0}{9} \times (F - 32) \\
\frac{5}{9.0} \times (F - 32) \\
\frac{5 \times (F-32)}{9} \\
\frac{(float)\frac{5}{9}}{1.8} \\
(F-32)/1.8
\]

(5.0/9) will yield a double value

(float - int) and (float * int) yield a float value

typecast 5 into a float

perform the division 9/5 to yield 1.8
The last one is the most efficient because it eliminates one multiplication operation. *Whenever possible, try to reduce the number of arithmetic operations to a minimum by proper rewriting of the expressions.* This leads to more efficient programs.

The major steps in this C program are as follows:

1. Declare 2 float variables: tempfahr, tempcels
2. Read in value of tempfahr (include stdio.h header file)
3. Compute tempcels = (tempfahr - 32)/1.8
4. Write the output value of tempcels in a good readable format

---

**Program 9.4 – FTOC.C**  
**Converting a temperature from Fahrenheit to Celsius scale**

```c
/* Program to convert a temperature in Fahrenheit to Celsius.*/
#include <stdio.h>
void main()
{
    /* declare variables */
    float tempfahr, tempcels;
    /* read in value of tempfahr */
    printf("\nEnter temperature in Fahrenheit :");
    scanf("%f", &tempfahr);
    printf("\nThe temperature is %f degrees Fahrenheit", tempfahr);
    /* Perform the calculations */
    tempcels = (tempfahr-32)/1.8;
    /* Write the output */
    printf("\nwhich is equivalent to %f degrees Celsius", tempcels);
}
```

---

9.6  
**Mini-Problem 5 - Area and Volume of a Sphere**

Problem Statement: Write a C program to calculate the surface area and the volume of a sphere given its radius. Use the following formulas.

Volume of a Sphere = \(4\pi r^3/3\)
Surface Area of a Sphere = \(4\pi r^2\)

We will call this program SPHERE.C.

What are the variables that we need for this program? We will need one variable to store the radius input by the user. Then, we need 2 more variables to store the surface area and the volume of the sphere. The presence of fractions in the equations suggests that we will need a floating point data type for these variables. It is also obvious that a very high precision is not necessary for this problem. Therefore, we choose *float* type variables, sphervol, sphervarea.
We could also use another variable for storage of PI but, keeping in mind that it is a constant, we choose to define it as a symbolic constant.

Next, we need to examine the formulas for calculation. Since, there is no exponentiation operator in C, we need to write the powers of x as repeated multiplication.

\[
\text{Volume} = 4\pi r^3/3 \\
\text{Area} = 4\pi r^2
\]

Can we perform these calculations more efficiently? Notice that the formula for the volume can be rewritten as

\[
\text{Volume} = \text{Area} \times r/3
\]

and by doing this we eliminate 3 multiplication operations!

The major steps in this C program are as follows:

1. Define a symbolic constant PI with value 3.141592
2. Declare 3 float variables: radius, spher_vol, spher_area.
3. Read in value of radius (include stdio.h header file)
4. Compute spher_area = 4 * PI * radius * radius
5. Compute spher_vol = spher_area * radius/3
6. Write the values of spher_vol and spher_area in a good readable format

Program 9.5 – SPHERE.C  
Area and Volume of a Sphere

```c
/* ============================================================== 
** SPHERE.C ============================================================== 
** Program to calculate volume and surface area of a sphere. 
** ==============================================================*/
#include <stdio.h>
#define PI 3.141592

void main()
{
    /*declare variables */
    float radius, spher_vol, spher_area;
    /* read in value of radius */
    printf("\n Enter radius : ");
    scanf("%f",&radius);
    printf("\n The radius of the sphere is %f",radius);
    /* Perform the calculations */
    spher_area = 4 * PI * radius * radius;
    spher_vol = spher_area * radius/3;
    /* write the output */
    printf("\n\n Volume of sphere = %g",spher_vol);
    printf("\n Surface Area of sphere = %g",spher_area);
}
```

9.7 Mini-Problem 6 - Scientific Calculation

Problem Statement: Write a C program to calculate the effective mass, \( m_e \) (in kg), of a particle given its velocity, \( v \) (in m/sec), and its rest mass, \( m_0 \), using the relativistic relation

\[
m_e = \frac{m_0}{\sqrt{1 - \frac{v^2}{c^2}}}
\]

where \( c \) is the speed of light in vacuum = \( 3.1 \times 10^8 \) m/sec.

We will call this program EFFMASS.C

We will declare the speed of light as a symbolic constant LIGHTSPEED with a value of 3.1e8. Then, we need 2 variables for storing the rest mass and the velocity. What kind of floating point data type is most suitable for these calculations? Notice that we will have to use the \texttt{sqrt} function to compute a square root. If you are going to use any of the standard mathematical functions, it is best to use double type variables because the double type is what all math functions expect as their arguments. We will also need a variable to store the effective mass.

Let us now look at the calculations. We could perform the entire calculation in a single expression but it is often more efficient to perform the calculations in steps. For example,

\[
v^2 / c^2 = (v*v) / (c*c)
\]

but if we create an intermediate variable temp and calculate as follows

\[
temp = v/c
\]
\[
v^2 / c^2 = temp * temp
\]

we eliminate one multiplication operation and introduce an additional assignment operation.

Assignments, additions, and subtractions take very little time as compared to multiplications and divisions. Therefore, reducing multiplications and divisions (even at the cost of introducing a few extra assignments, additions, or subtractions) will improve a program's execution speed.

In addition, the intermediate variable, temp, may itself convey some useful information, in this case, the ratio of the velocity to the velocity of light.

The major steps in this C program are as follows:

1. Define a symbolic constant LIGHTSPEED with a value 3.1E8
2. Declare double variables: restmass, velocity, effmass, vbyc
3. Read in value of restmass (include \texttt{stdio.h} header file)
4. Read in value of velocity.
5. Computer vbyc = velocity/LIGHTSPEED
6. Write value of vbyc
7. Compute effmass = restmass / sqrt(1.0 - vbyc * vbyc) (need to include math.h header file)
8. Write the values of **********in a good readable format

Program 9.6 – EFFMASS.C Calculation of effective relativistic mass

```c
#include <stdio.h>
#include <math.h>
#define LIGHTSPEED 3.1E8

void main()
{
    /*declare variables */
    double restmass, velocity, effmass, vbyc;

    /*read in restmass and velocity*/
    printf("n Enter rest mass : ");
    scanf("%lg",&restmass);
    printf("n Enter velocity : ");
    scanf("%lg",&velocity);
    printf("n Rest mass = %lg, velocity = %lg\n",restmass,velocity);

    vbyc = velocity/LIGHTSPEED;
    printf("n Velocity/Speed of Light = %lg",vbyc);

    effmass = restmass/(sqrt(1.0 - vbyc*vbyc));
    /* write the output */
    printf("n\n\n The effective mass = %lg",effmass);
}
```

9.8 Mini-Problem 7 - Evaluating a Polynomial

Problem Statement: Write a program to evaluate the following polynomial given the values of the coefficients a, b, c, and d as inputs.

\[ y = a + bx + cx^2 + dx^3 \]

We will call this program EVALPOLY.C

We will use double values for the calculations. We will need 5 variables to store the input values of the four coefficients and the value of x. One more variable will be required to store the resultant value of y.

Let us now look at the calculations. We could perform the entire calculation as shown below
\[ y = a + b \cdot x + c \cdot x^2 + d \cdot x^3 \]

which involves 6 multiplications and 3 additions. Let us rewrite the polynomial in a **nested polynomial form** as follows

\[ y = a + (b + (c + d \cdot x) \cdot x) \cdot x \]

This is exactly the same polynomial but now it involves only 3 multiplications and is a more efficient expression to carry out the calculations.

The major steps in this C program are as follows:

1. Declare double variables: yval, xval, cofa, cofb, cofc, cofd
2. Read in values of cofa, cofb, cofc, cofd (include `stdio.h` header file)
3. Read in value of xval.
4. Compute \[ yval = cofa + (cofb + (cofc + cofd \cdot xval) \cdot xval) \cdot xval \]
5. Write the value of yval in a good readable format.

**Program 9.7 – EVALPOLY.C** Evaluation of a Polynomial using Horner’s Rule

```c
/* ================================================================ EVALPOLY.C ================================================================
 Program to evaluate a cubic polynomial. 
==================================================================*/
#include <stdio.h>

void main()
{
    /*declare variables */
    double yval, xval;
    double cofa, cofb, cofc, cofd;

    /* read in x-value*/
    printf("\n Enter x : ");
    scanf("%lg",&xval);
    printf("\n x = %lg",xval);

    /* read in coefficients */
    printf("\n Enter a,b,c,d (separated by space) : ");
    scanf("%lg%lg%lg%lg",&cofa,&cofb,&cofc,&cofd);
    printf("\n The coefficients are a=%lg, b=%lg, c=%lg, d=%lg",cofa,cofb,cofc,cofd);

    yval = cofa+(cofb+(cofc+cofd*xval)*xval)*xval;

    /* write y-value */
    printf("\n\n y = %lg",yval);
}
```

### 9.9 Points to Remember
**EXERCISE**

Problem Statement: Given the initial velocity, $u$ (m/sec), of a particle, uniform acceleration, $a$ (m/sec$^2$), and the time elapsed, $t$ (sec.), write a program to calculate the distance travelled, $s$ (m), using the formula

$$s = ut + \frac{1}{2} at^2$$

Write a program DISTCALC.C which will use this expression to calculate the distance travelled.

We will use double variables: `init_vel`, `time`, `accel`, `distance`. The unnecessary calculation of 1/2 is avoided by writing it as 0.5. The expression is further modified and written in nested polynomial form as follows

$$distance = (init\_vel + 0.5 \times accel \times time) \times time$$

2. Intersection of lines

$$a_1x + b_1y + c_1 = 0$$

$$D = a_1b_2 - a_2b_1 \text{ not equal to zero}$$

$$x = (b_2c_1 - b_1c_2)/D, \quad y = a_1c_2 - a_2c_1)/D$$
14

Standard Library Functions

10.0 Lesson Goals

10.1 The ANSI C Standard Library

The standard library contains a number of functions what are commonly required by all programmers. The library of functions is not an intrinsic part of the language and is only peripheral to the language itself. But the existence of the standard library eliminates the need for every programmer to "reinvent the wheel". We could develop our own function to find the square root of a floating point value but it is more convenient to use a readymade function. The function prototypes for these standard library functions are provided in a number of header files which are standardized. You will find that your own C compiler will come with many more header files which contain information about the implementation specific function libraries. Here, we will look at the various libraries specified by the ANSI standard which are not implementation dependent. Using these functions guarantees a very high degree of portability for your C programs. The following is a summary of some of the header files associated with the functions of the standard libraries.

- float.h contains the definitions of the implementation dependent constants related to the various floating point data types (see Chapter 3 for more details).
- limits.h contains the definitions of the implementation dependent constants related to the various integer data types (see Chapter 2 for more details).
- stdio.h contains function prototypes for various functions related to input and output (see Chapter 11 for more details).
- math.h contains function prototypes for various mathematical functions.
- ctype.h contains function prototypes for various functions related to characters.
- string.h contains function prototypes for various functions related to strings.
- stdlib.h contains a mixed bag of utility functions for performing various tasks.
- time.h contains the types and functions for manipulating time and date information (see Chapter 30 for more details).
10.2 Interpreting a Function Prototype

To be able to call a function from your program, you need to be able to interpret a function prototype. For example, let us examine the following

\[
\text{double sqrt(double x)}
\]

The first thing to note is the data type of the value returned by the function which in this case is \textit{double}. Every function can return one or zero values as the end result of calling the function.

- When one value is returned, the data type of the returned value must be specified before the name of the function.
- When the function does not return any value, the returned data type is specified as \textit{void}.
- If no return type is specified before a function name, the returned data type is assumed to be \textit{int} by default.

Therefore, the function declared as

\[
\text{foobar2(double x)}
\]

is assumed to return a value of type \textit{int} whereas a function declared as

\[
\text{void foobar3(double x)}
\]

will not return anything.

The value returned by a function can then be used in any regular expression, for instance, it can be assigned to a variable or it can even be a part of the call to another function. For example,

\[
\begin{align*}
x &= \text{sqrt}(p); \\
y &= \text{sqrt}(\text{sqrt}(q));
\end{align*}
\]

The second example assigns the fourth root (square root of the square root) of \( q \) to \( y \). It is obvious that we cannot assign the return value of a \textit{void} type function to anything because it does not return anything.

In some cases, the return value may be ignored as shown in the following example,

\[
\text{sqrt(x)};
\]
We calculate the square root of x but do not make any use of the result! However illogical it may seem, this is a perfectly valid way of doing things in C. Sometimes, this feature can be put to use in a clever fashion. For example, we can use the getchar function (which reads one character entered by the user) in the following manner

```c
getchar();
p = getchar();
```

Here, the first character entered by the user is ignored and the second character input by the user is assigned to the variable `p`.

Next, we need to examine the list of arguments needed by the function. In the above example, the function `sqrt` requires only one argument of type double. Let us look at another example

```c
void foobar(double x, int k)
```

This function `foobar` does not return any value. It has two arguments, the first of type `double` and the second argument of the type `int`. The values used in the call to the function must match these in both type and number, i.e., in any call to the function `foobar`,

- there must be two arguments
- the first argument must be of type `double`
- the second must be of the type `int`

Many errors in using the standard library functions can be traced to a mismatch in either the number or the type of the arguments.

### 10.3 Standard Input/Output Functions

We must include the `<stdio.h>` to use the following functions:

**getchar(void)**

returns the next unsigned character input from the keyboard (standard input device).

**putchar(char c)**

writes the character `c` to the monitor (standard output device). The character `c` is returned.

`getchar` and `putchar` are the simplest functions for the input and output of a character as shown in the following program.

```c
char cone = 'A', ctwo;
printf("Enter a character : ");
getchar(ctwo);
printf("\n Your have entered : ");
putchar(ctwo);
putchar('
');
putchar('a');
putchar(cone);
```
gets(s)
reads the next input line of text from the keyboard (standard input device) into the string
named s.

puts(s)
writes the string s on the monitor (standard output device)

gets and puts are the simplest functions for the input and output of a string as shown in the
following program segment.

```c
char mystr[]="Hello";
char nstr[80];
puts(mystr);
printf("Enter a line of text : \n");
gets(nstr);
printf("\n You have entered the string : \n");
puts(nstr);
```

There are two other functions, printf and scanf, which provide a powerful range of facilities for
input and output of various types of variables. We will look at these functions in Chapter **.

### 10.4 Mathematical Functions

In the following definitions of the functions, assume that x and y are type double and n is an int.

All mathematical functions return a double value. All the trigonometric functions accept
angles in radians and NOT in degrees. Therefore, any angle in degrees must be converted to the
equivalent value in radians before any of these functions is called.

We must include the `<math.h>` to use the following functions:

**Trigonometric Functions**

- sin(x)
- cos(x)
- tan(x)
- asin(x) = sin⁻¹(x) in the interval [-π/2, π/2]
- acos(x) = cos⁻¹(x) in the interval [0, π]
- atan(x) = tan⁻¹(x) in the interval [-π/2, π/2]
- atan2(y, x) = tan⁻¹(y/x) in the interval [-π, π]

**Hyperbolic Functions**

- sinh(x)
- cosh(x)
- tanh(x)

**Exponentiation Functions**

- pow(x, y) = x^y
- sqrt(x) = \( \sqrt{x} \)
- exp(x) = e^x

**Logarithmic Functions**

- log(x) natural logarithm
**log10(x)** base 10 logarithm

**Other Functions**

- **fabs(x)**
  \[= |x|\]
  - largest integer less than or equal to x
- **floor(x)**
  \[= \text{smallest integer greater than or equal to } x\]
- **ceil(x)**
- **ldexp**
- **frexp**
- **modf**
- **fmod**

Note that the function **fabs** should be used for a **double** type variable. The **abs** and **labs** functions given in `<stdlib.h>` are the functions to be used for finding absolute values of **int** type values and **long** values, respectively.

The floor and ceil functions should be understood very carefully.

- **floor(4.3)** will yield **4.0** and **ceil(4.3)** will yield **5.0**.
- **floor(-4.3)** will yield **-5.0** and **ceil(-4.3)** will yield **-4.0**.

## 10.5 Character Functions

We must include the `<ctype.h>` to use the following functions:

- **tolower(char c)** returns the lowercase character corresponding to the character c
- **toupper(char c)** returns the uppercase character corresponding to the character c

The following program segment demonstrates the use of these functions.

```c
char clower='p',cupper;
putchar(clower);
cupper = toupper(clower);
putchar(cupper);
putchar(tolower(cupper));
```

`<ctype.h>` also contains another useful set of functions relating to character type testing which we will study in Chapter 12.

## 10.6 String Functions

We must include the `<string.h>` to use the following functions:

- **strlen(s)** returns the number of characters (EXCLUDING the null terminator) in s as an int.
- **strcpy(t,s)** copies the contents of string s to string t.
- **strncpy(t,s,n)**
copies at most \( n \) characters of string \( s \) to string \( t \). \( n \) is an \texttt{int}.

\texttt{strcat(s,cs)}

concatenates the string \( cs \) to the end of string \( s \).

\texttt{strncat(s,cs,n)}

concatenates at most \( n \) characters of the string \( cs \) to the end of string \( s \).

The following C program, STRFUNS.C, illustrates the use of these functions.

**Program 10.1 – STRFUNS.C String functions in the standard function library**

```c
/* ==----------------------------- STRFUNS.C -----------------------------==
 Program to try out the string functions.
==================================================================*/
#include <stdio.h>
#include <string.h>

void main()
{
    /* declare variables */
    char first[60], last[60], namecopy[80];
    int fnum, lnum;

    printf("\n Enter your first name : ");
    gets(first);
    printf("\n Enter your last name  : ");
    gets(last);

    fnum = strlen(first);
    lnum = strlen(last);
    printf("\n Your first name has %d characters",fnum);
    printf("\n Your last name has  %d characters",lnum);

    strcat(first,last);
    printf("\n Your full name is : ");
    puts(first);

    strcpy(namecopy, first);
    printf("\n A copy of your name is : ");
    puts(namecopy);
}
```

The string library contains many other functions which we will examine in Chapter 22.

### 10.7 Utility Functions

We must include the \texttt{<stdlib.h>} to use the following functions:

\texttt{int abs(int n)}

returns the absolute value of an \texttt{int} value

\texttt{long labs (long n)}

returns the absolute value of a \texttt{long} value

\texttt{double atof(s)}

\texttt{int atoi(s)}

\texttt{long atol(s)}
all these 3 functions convert a string $s$ into a numeric value of the type given by their respective prototypes.

```c
int rand(void)
returns a pseudorandom integer in the range of 0 to RAND_MAX (predefined symbolic constant).

void srand(unsigned int n)
sets the seed value used for the rand function.

void exit(int s)
causes the program to terminate. A value of 0 for s usually indicates a normal termination. Other integer values are used to indicate abnormal termination. The integer value s is made available to the operating system for further processing.
```

```c
strtol **strtol**
strtol
```

There are many other useful functions defined in `<stdlib.h>` which we will examine in later chapters.

### 10.8 Mini-Problem 9 - Range of a Projectile

Problem: Write a program to calculate the range of a projectile, given its initial launch speed, $v$, and its angle of launch, $\theta$, in degrees, using the following relation

$$d = \frac{v^2 \sin(2\theta)}{2g}$$

where $g$ is the acceleration due to gravity ($=9.81 \text{ m/sec}^2$)

We will call this program RANGECAL.C.

We will need 4 `double` variables for the value of pi, velocity, angle and range. In this program we will compute the value of $\pi$ using $\cos^{-1}(-1)$ instead of using a symbolic constant.

We must remember to convert the angle from degrees to radians using the relation

$$(\text{angle in radians}) = (\text{angle in degrees}) \times \frac{\pi}{180}$$

We will use a symbolic constant GACCEL for $g$.

**Program 10.2 – RANGECAL.C Calculation of the range of a projectile**

```c
/* ------------------------------- RANGECAL.C -------------------------------
 Program to calculate range of a projectile.
 */
```
```c
#include <stdio.h>
#include <math.h>
#define GACCEL 9.81

void main()
{
    /*declare variables */
    double vel, angle, range, pi;
    /* Compute value of pi*/
    pi = acos(-1.0);
    /*read in velocity and angle*/
    printf("n Enter velocity (in m/sec): ");
    scanf("%lg",&vel);
    printf("n Enter angle (in degrees) : ");
    scanf("%lg",&angle);
    printf("n Velocity = %lg, angle = %lg degrees",vel,angle);
    /* change angle to radians*/
    angle *= pi/180.0;
    /* compute range */
    range = vel*vel*sin(2.0*angle)/(2.0*GACCEL);
    /* write the output */
    printf("n"n The range of the projectile = %lg m.",range);
}
```

10.9 Mini-Problem 10 - Cartesian to Polar Transformation

Problem: Given the Cartesian coordinates x and y or a point in 2-dimensional space, convert them into the equivalent polar coordinates, r and \( \theta \) (in radians).

We will call the program CARTOPOL.C.

We need 4 double variables - xcoord, ycoord, rad, theta.

To calculate, r from x and y, we can use the Pythagorean relation

\[
r = \sqrt{x^2 + y^2}
\]

How do we calculate the angle \( \theta \)? Taking \( \tan^{-1}(y/x) \) will never yield an angle in the third quadrant. The function \( \text{atan2}(y,x) \) comes in very useful here as it yields a value in the range \([-\pi, \pi] \) which is just what we need here.

Program 1.0.3 – CARTOPOL.C Converting coordinates from Cartesian to Polar

```c
/* ============================================================== CARTOPOL.C ==============================================================
Program to convert from Cartesian to Polar coordinates.
 ============================================================== Cartopol.C ==============================================================*/
#include <stdio.h>
#include <math.h>
```
```c
void main()
{
    /* declare variables */
    double xcoord, ycoord, rad, angle, pi;
    /* compute value of pi */
    pi = acos(-1.0);
    /* read in the 2 coordinates */
    printf("Enter xcoord : ");
    scanf("%lg",&xcoord);
    printf("Enter ycoord : ");
    scanf("%lg",&ycoord);
    /* compute radius and angle */
    rad = sqrt(xcoord*xcoord + ycoord*ycoord);
    angle = atan2(ycoord,xcoord);
    /* write the output */
    printf("radius = %lg\n",rad);
    printf("angle = %lg radians = %lg degrees",angle,angle*180.0/pi);
}
```

10.9 Points to Remember

Review Quiz

Programming Exercises
Formatted Input & Output

11.0 Lesson Goals

11.1 The printf Function

The printf function is the simplest function for obtaining formatted output. Its prototype is defined in <stdio.h>. The formal prototype is

\[
\text{int printf(const char * format, . . .)}
\]

In a simpler notation, we could write it as

\[
\text{int printf(format string, arg1, arg2, arg3, . . . , argN)}
\]

The function returns the total number of characters printed to the screen (or standard output device) but this value is rarely used.

Unlike most functions that we have met so far, the printf function does not have a fixed number of arguments. It can take a variable number of arguments as indicated by the ellipsis ( . . . ) in the function prototype. At the least, it might have only one argument, namely, the format string. In addition, it can have any number of arguments following the format string. The output from the printf function consists of the values of these arguments, converted according to the conversion specifications given in the format string.

The format string contains two things:

- ordinary characters which are written out without change
- conversion specifications of the form %z where the character z specifies the type of conversion desired as shown in Table 11.1.
Table 11.1 Conversion Specifications for printf function

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Conversion Spec.</th>
<th>Output Format</th>
<th>Example Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>%d</td>
<td>signed decimal</td>
<td>-3456</td>
</tr>
<tr>
<td></td>
<td>%i</td>
<td>signed decimal</td>
<td>-3456</td>
</tr>
<tr>
<td></td>
<td>%o</td>
<td>unsigned octal (no leading 0)</td>
<td>345</td>
</tr>
<tr>
<td></td>
<td>%x</td>
<td>unsigned hexadecimal using abcdedef as hex digits</td>
<td>f7ef</td>
</tr>
<tr>
<td></td>
<td>%X</td>
<td>unsigned hexadecimal using ABCDEF as hex digits</td>
<td>F7EF</td>
</tr>
<tr>
<td></td>
<td>%u</td>
<td>unsigned decimal</td>
<td>3456</td>
</tr>
<tr>
<td>double, float</td>
<td>%F</td>
<td>decimal notation (without exponent)</td>
<td>-65.78</td>
</tr>
<tr>
<td></td>
<td>%e</td>
<td>decimal notation (with exponent as e±xx)</td>
<td>-0.6578e11</td>
</tr>
<tr>
<td></td>
<td>%E</td>
<td>decimal notation (with exponent as e±xx)</td>
<td>-0.6578E11</td>
</tr>
<tr>
<td></td>
<td>%g</td>
<td>either as %f or as %e</td>
<td></td>
</tr>
<tr>
<td></td>
<td>%G</td>
<td>either as %f or as %E</td>
<td></td>
</tr>
<tr>
<td>char</td>
<td>%c</td>
<td>unsigned character</td>
<td>Q</td>
</tr>
<tr>
<td>String</td>
<td>%s</td>
<td>characters from string (excluding null terminator)</td>
<td>Hello</td>
</tr>
<tr>
<td>Pointer</td>
<td>%p</td>
<td>print as a pointer (implementation dependent)</td>
<td></td>
</tr>
<tr>
<td>Write into int type argument</td>
<td>%n</td>
<td>number of characters written so far.</td>
<td></td>
</tr>
<tr>
<td>No argument</td>
<td>%</td>
<td>the character %’</td>
<td>%</td>
</tr>
</tbody>
</table>

A prefix of ‘h’ before an integer specification (e.g., %hd) is used for a short type.
A prefix of ‘l’ before an integer specification (e.g., %ld) is used for long type.
A prefix of ‘L’ before a double specification (e.g., %Lg) is used for long double type.

There must be one conversion specification for each argument given after the format string. The conversion specifications are matched on a one-to-one basis with the arguments to produce the output as shown in Figure 11.1.

```c
printf("\n Value of x is %d, y = %d and the letter is %c", 44, y, 'G');
```

**OUTPUT:** Value of x is 44, y = -6357 and the letter is G
Figure 11.1 The correspondence between the arguments & conversion specifications

Notice that the arguments can be either constants or variables as shown in the example of Figure 11.1. 44 and 'Q' are constants whereas y is an int variable whose value is -6357.

Also notice that, apart from the conversion specifications, everything is reproduced exactly as it appears in the format string. The program, SIMPRINT.C, given below demonstrates some simple uses of the printf function.

Program 11.1 – SIMPRINT.C Simple conversion specifications for the printf function

```c
/*
 * fixed conversion specifications for the printf function.
 */
#include <stdio.h>

int main()
{
    char cc = 'Q';
    char name[80] = "Hello, World!"
    short j = -123;
    int k = 3456;
    long m = -98765L;
    float x = -3456.7;
    double y = 333.77E33;
    long double z = 444.55E66;

    printf("\n cc = %c, ASCII code of cc = %d", cc, cc);
    printf("\n name = %s", name);
    printf("\n  j = %d", j);
    printf("\n  k = %d", k);
    printf("\n  m = %ld", m);
    printf("\n  x = %g", x);
    printf("\n  y = %g", y);
    printf("\n  z = %Lg", z);

    return 0;
}
```

Output:

c = Q, ASCII code of cc = 81
name = Hello, World!
j = -123
k = 3456
m = -98765
x = -3456.7
y = 3.3377e+33
z = 4.4455e+68

11.2 Fine Tuning the printf Function

In the last section, we have learnt about the simple formats for getting output. Very often, we would like to fine tune the appearance of the output. This is achieved by inserting the some additional information between the % symbol and the conversion specification character z. This might consist of:

- Flags
- Field width specifications
Precision specifications

The following flags are available:

- forces left justification of the value
+ forces a + sign to be printed in front of positive numbers
space if the first character is not a sign, this puts a space character
0 for numbers, pads leading space with zeros
# specifies a modified output depending on the basic conversion specification character
  for o, the first digit is zero
  for x, 0x will be prefixed to non-zero value
  for X, 0X will be prefixed to non-zero value
  for e, f, and E, decimal point will always be shown
  for g and G, decimal point and trailing zeros will be shown

The minimum field width is specified by an integer value, N. The argument will be allocated at least N characters in the output and more if necessary. The unused characters are filled with space characters (or with '0' if 0 flag has been used). This is known as padding.

The precision is specified by writing a decimal point followed by a number P. For a string, P is the maximum number of characters to be printed. For floating point values, it is the minimum number of digits to be printed after the decimal point. For integer values, it is the minimum number of digits to be printed, padded with leading zeros if required.

The following program, FINPRINT.C, demonstrates various ways in which the output from the printf function can be modified. This program includes a user-defined function, count_line, which outputs a line of digits, making it easier to count characters in the output.

Program 11.3 – FINPRINT.C  Flags and modifiers for printf conversion specifications

```c
/*  Fine-tuning the printf function. */
#include <stdio.h>
void count_line(void);
void main()
{
    char name[80]="Hello, World!";
    int j = -3456, k = 783;
    double x = -3456.487, y = 783.23;
    count_line();
    printf("\n01|%s|",name);
    printf("\n02|%30s|",name);
    printf("\n03|%6s|",name);
    printf("\n04|%30.5s|",name);
...
1.3 The scanf Function

The `scanf` function is the simplest function for obtaining formatted input. Its prototype is defined in `<stdio.h>`. The formal prototype is
int scanf(const char * format, ...) 

In a simpler notation, we could write it as

int scanf(format string, parg1, parg2, parg3, ..., pargN)

The function returns the total number of input values converted and assigned to the respective arguments. The input is usually from the keyboard (or standard input device).

Just like the printf function, the scanf function can also take a variable number of arguments as indicated by the ellipsis (...) in the function prototype. At the least, it might have only one argument, namely, the format string. In addition, it can have any number of arguments following the format string. All the arguments must be pointers. Since, we haven't yet met pointers, we will use the following rule-of-thumb for the arguments in the scanf function.

If it is a string, use the name of the string.
If it is any other type of variable, prefix the '&' symbol to the variable name.

EXAMPLE:
char name[70], first;
int k;
scanf("%d%s%c", &k, name, &first);

Blanks and tabs in the format string are ignored. All other characters (except '%') must match the next non-whitespace character in the input. Whitespace characters in the input are ignored.

Each input field in the input is converted and assigned to an argument. An input field is a string of non-whitespace characters in the input. The only exception to this rule is when characters are being read in using a %c conversion specification. In the case of %c, whitespace characters are not skipped over as they are valid character constants.

Conversion specifications (shown in Table 11.2) for the scanf function are slightly different from those of the printf function.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Conversion Spec.</th>
<th>Output Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>%d</td>
<td>decimal integer</td>
</tr>
<tr>
<td></td>
<td>%i</td>
<td>any integer (decimal or octal with 0 prefix or hexadecimal with 0x prefix)</td>
</tr>
<tr>
<td></td>
<td>%o</td>
<td>octal integer</td>
</tr>
<tr>
<td></td>
<td>%x</td>
<td>hexadecimal integer</td>
</tr>
<tr>
<td></td>
<td>%u</td>
<td>unsigned decimal integer</td>
</tr>
<tr>
<td>float</td>
<td>%f</td>
<td>float type value</td>
</tr>
<tr>
<td></td>
<td>%e</td>
<td>float type value</td>
</tr>
<tr>
<td><strong>char</strong></td>
<td><strong>%g</strong></td>
<td>float type value</td>
</tr>
<tr>
<td>-------------</td>
<td>--------</td>
<td>------------------</td>
</tr>
<tr>
<td><strong>%c</strong></td>
<td></td>
<td>unsigned character</td>
</tr>
<tr>
<td><strong>String</strong></td>
<td><strong>%s</strong></td>
<td>string of non-whitespace characters ('\0' added)</td>
</tr>
<tr>
<td></td>
<td>[.....]</td>
<td>set of valid characters for string</td>
</tr>
<tr>
<td></td>
<td>[^.....]</td>
<td>set of invalid characters for string</td>
</tr>
<tr>
<td><strong>Pointer</strong></td>
<td><strong>%p</strong></td>
<td>a pointer value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(implementation dependent)</td>
</tr>
<tr>
<td><strong>Write into int type argument</strong></td>
<td><strong>%n</strong></td>
<td>number of characters read so far by the scanf function.</td>
</tr>
<tr>
<td><strong>No argument</strong></td>
<td><strong>%%</strong></td>
<td>the character '%'</td>
</tr>
</tbody>
</table>

A prefix of 'h' before an integer specification (e.g., %hd) is used for a **short integer** type.
A prefix of 'l' before an integer specification (e.g., %1d) is used for **long integer** type.
A prefix of 'l' before a float specification (e.g., %lg) is used for **double** type.
A prefix of 'L' before a float specification (e.g., %Le) is used for **long double** type.

The following program, SIMSCAN.C, and the sample inputs shown below it demonstrate some of the features of the **scanf** function.

**Program 11.3 – SIMSCAN.C**  Simple conversion specifications for the scanf function

```c
#include <stdio.h>

void main()
{
    char cc;
    char name1[80]="", name2[80]="";
    short j;
    int k;
    long m;
    float x;
    double y;
    long double z;

    printf("\n Enter a character :" );
    scanf("%c",&cc);
    printf("\n cc = %c, ASCII code of cc = %hd",cc,cc);
    fflush(stdin);

    printf("\n Enter a string (at least two words) \n" );
    scanf("%s %s",name1,name2);
    printf("\n name1 = %s, name2 = %s",name1,name2);
    fflush(stdin);

    printf("\n Enter a short int value :" );
    scanf("%hd",&j);
    printf("\n j = %hd",j);
    fflush(stdin);
```
printf("\n Enter an integer value :");
scanf("%d",&k);
printf("    k = %d",k);
fflush(stdin);

printf("\n Enter a long int value :");
scanf("%ld",&m);
printf("    m = %ld",m);
fflush(stdin);

printf("\n Enter a float value :");
scanf("%g",&x);
printf("    x = %g",x);
fflush(stdin);

printf("\n Enter a double value :");
scanf("%lg",&y);
printf("    y = %g",y);
fflush(stdin);

printf("\n Enter a long double value :");
scanf("%Lg",&z);
printf("    z = %Lg",z);

11.4 Assignment Suppression

Putting a prefix of '"' before the conversion specification character leads to assignment suppression. Assignment suppression means that the value is read from the input and is subsequently discarded without any assignment to an argument. Let us look at a program segment to understand this better.

```c
int k=10,m=20;
printf("\n Enter values of k1 and k2 : ");
scanf("%d%d%d",&k,&m);
printf("\n k = %d, m = %d",k,m);
```

If the user input is as follows (values separated by spaces because spaces are ignored by the scanf function).

40 60 80

The value of 40 is read in as an integer and is discarded because we have specified assignment suppression. The value of 60 is assigned to k. The value of 80 is assigned to m.

11.5 Points to Remember

Review Quiz
Programming Exercises
Logical Expressions

Two roads diverged in a yellow wood,
And sorry I could not travel both
And be one traveler, long I stood

from "The Road Not Taken" by Robert Frost

12.0 Lesson Goals

12.1 Decision Criteria

The programs that we have seen so far are pretty much linear in structure. But very often we need to perform some calculations based on some criterion. For example, if an integer \( n \) is even, we need to perform one calculation and if it is odd, then a different calculation needs to be performed. Like the traveller of Robert Frost, we need to take decisions about which path to take. For taking such decisions, we pose questions to the computer. To keep the questions simple, we keep in mind the binary nature of all computers and ask questions that can be answered with a simple TRUE or FALSE (or equivalently, YES or NO).

For instance, we cannot pose the question "Is the integer \( k \) odd?" because the computer has no inherent knowledge of "odd" and "even". We have to use our knowledge of arithmetic and rephrase the question as "Does the expression \( k \% 2 \) yield a value of 1?" The answer is a YES or a NO.

12.2 Relational Operators

Relational operators are used to create **logical expressions** (Boolean expressions). A logical expression is an expression that can be evaluated by the computer to yield a result of either TRUE or FALSE. The relational operators can be of either inequality or equality type.

**INEQUALITY OPERATORS**

-  \(<\)  less than
-  \(\leq\)  less than or equal to
-  \(>\)  greater than
-  \(\geq\)  greater than or equal to
**EQUALITY OPERATORS**

`==`  
equal to

`!=`  
not equal to

Writing `=` when you mean `==` is probably the most common error in C programming! And this error cannot be caught by your compiler because both are valid operators. Let us look at an example. With `m` having a value of 4,

\[(m=4)\] evaluates to TRUE

while

\[(m==4)\] evaluates to FALSE!

The second case is obvious as it checks for the equality of `m` and 4. In the first case, `m` is assigned a value of 4. The result of an assignment is the value of the variable on the left hand side. Therefore, \[(m=4)\] evaluates to 4 and 4 being a nonzero integer, the expression evaluates to TRUE.

---

### 12.3 Logical Operators

Logical operators are also known as logical connectives. They are used to combine simple logical expressions into one large logical expression.

- `&&` AND (Binary operator)
- `||` OR (Binary operator)
- `!` NOT (Unary operator)

\[\text{logexpr1} \&\& \text{logexpr2}\] is TRUE only if both \text{logexpr1} and \text{logexpr2} are TRUE, otherwise it is FALSE.

\[\text{logexpr1} \mid\mid \text{logexpr2}\] is TRUE if either \text{logexpr1} or \text{logexpr2} or both are TRUE, otherwise it is FALSE, i.e., it is FALSE only if both \text{logexpr1} or \text{logexpr2} are FALSE

\[!\text{logexpr}\] is TRUE if \text{logexpr} is FALSE, otherwise it is TRUE.

The examples shown below illustrate the use of relational and logical operators to construct logical expressions. The truth value of each expression is also shown.

```c
int j=3, k=4, m=6;
char p = 'a', q = 'A';
j < k            TRUE
(j+k) < m        FALSE
(m-1) == (k-1)   TRUE
(j<k) && (m > k+1) TRUE
!(j<k)           FALSE
!(j<k) || (j<m)  TRUE
p > q            TRUE
```

107
Notice how we can use characters to create logical expressions. This is possible because characters are stored as 1 byte integers. Therefore, comparing characters involves comparing their equivalent ASCII codes.

### 12.4 Some Boolean Identities

If A, B, and C are logical expressions, then the following identities are valid.

\[
\begin{align*}
A \land B \land C & \Leftrightarrow (A \land B) \land C \Leftrightarrow A \land (B \land C) \\
A \lor B \lor C & \Leftrightarrow (A \lor B) \lor C \Leftrightarrow A \lor (B \lor C) \\
A \land B & \Leftrightarrow B \land A \\
A \land A & \Leftrightarrow A \\
A \land \text{TRUE} & \Leftrightarrow A \\
A \land \text{FALSE} & \Leftrightarrow \text{FALSE} \\
A \lor A & \Leftrightarrow \text{TRUE} \\
A \lor \text{FALSE} & \Leftrightarrow A \\
(A \land B) \lor (A \land C) & \Leftrightarrow A \land (B \lor C) \\
!A & \land A \\
!A \land A & \Leftrightarrow \text{FALSE} \\
!(A \land B \land C) & \Leftrightarrow (!A) \lor (!B) \lor (!C) \\
!(A \lor B \lor C) & \Leftrightarrow (!A) \land (!B) \land (!C)
\end{align*}
\]

The following identities are known as De Morgan's complimentary relations

\[
\begin{align*}
!(A \land B \land C) & \Leftrightarrow (!A) \land (\neg B) \land (\neg C) \\
!(A \lor B \lor C) & \Leftrightarrow (!A) \land (\neg B) \land (\neg C)
\end{align*}
\]

The following identity is valid from Boolean algebra

\[
A \lor B \Leftrightarrow B \lor A \quad \text{(WARNING: may not work in C)}
\]

but in the C language it may result in slightly different operations being performed. Therefore, the equivalency shown in this identity must be taken with a pinch of salt as we will see in Section 12.6.

### 12.5 Storage of Boolean Values in C

Most high level languages (e.g., FORTRAN) have a special data type to store Boolean value of TRUE and FALSE. In this respect, C is an exception as it has no inbuilt data type for storing TRUE and FALSE. Instead, we get the immense flexibility of using any integer data type as a Boolean type. An integer value of 0 is considered FALSE. Any non-zero integer is considered TRUE. The internal representation of TRUE uses a value of 1 as shown by the output from the following program segment.

```c
printf("\nInternal value of TRUE is %d",4>3);
printf("\nInternal value of FALSE is %d",4<3);
```
This can be used to create some unusual expressions. For instance,

\[
\begin{align*}
m &= (4 > 3); \\
n &= (4 < 3);
\end{align*}
\]

assigns a value of 1 to \( m \) and a value of 0 to \( n \).

Even though, internally, TRUE may be stored as 1, you are free to use any nonzero integer to represent TRUE. **Any integer expression is a valid logical expression.** Since characters are also integers in the C language, **any character is a valid logical expression.**

The examples shown below illustrate the use of integers as logical expressions.

```c
long int j = 0;
int k = -4, m = 6;
char p = '\0', q = 'F';
j FALSE
k TRUE
m-8 TRUE
m-6 FALSE
m-2 TRUE
p FALSE
q TRUE
q - 'F' FALSE
q - 'A' TRUE
```

Due to this equivalency of integer expressions with logical expressions, the expression \((m==0)\) is equivalent to \(!m\). The choice between the two expressions depends on the programmer's judgment about the readability of the resultant expression. For instance,

\[
\begin{align*}
!passed \quad & \text{is more readable than} \quad passed == 0 \\
k\%2 == 0 \quad & \text{is more readable than} \quad !(k\%2)
\end{align*}
\]

**A very common error results from writing**

\((k=0)\)

**when one intends to write**

\((k==0)\)

The first expression changes the value of \( k \) to 0 and the final value of the expression is 0, i.e., FALSE. The second expression checks for the logical equality of \( k \) with 0.
In C, the evaluation of a logical expression is terminated as soon as its truth value has been ascertained. This can occasionally lead to some unusual side effects. Let us consider some logical expressions assuming \( m=3 \), \( n=2 \), and \( p=1 \) before these expressions are evaluated.

**LOGEXPR1** \((m++ == 3) \lor (n++ == 2) \lor (p++ == 1)\)

then \( m=4 \), \( n=2 \), and \( p=1 \) after this expression has been evaluated.

While evaluating the logical expression from left to right, \( m \) is compared against 3 and found equal to 3 yielding a TRUE value for the entire logical expression. Since, a TRUE value ORed with anything else is TRUE, further evaluation of the remaining expression is not necessary. The post increment is performed on \( m \) resulting in a value of 4 for \( m \).

**LOGEXPR2** \((m++ == 5) \lor (n++ == 2) \lor (p++ == 1)\)

then \( m=4 \), \( n=3 \), and \( p=1 \) after this expression has been evaluated.

While evaluating the logical expression from left to right, \( m \) is compared against 3 and found not equal to 3. Since, this is FALSE, the post increment is performed on \( m \) resulting in a value of 4 for \( m \). Then, the second expression \((n++ == 2)\) is evaluated and found to be TRUE. This yields a TRUE value for the entire logical expression. Since, a TRUE value ORed with anything else is TRUE, further evaluation of the remaining expression is not necessary.

**LOGEXPR3** \((m++ == 5) \lor (n++ == 7) \lor (p++ == 1)\)

then \( m=4 \), \( n=3 \), and \( p=2 \) after this expression has been evaluated.

While evaluating the logical expression from left to right, \( m \) is compared against 3 and found not equal to 3. Since, this is FALSE, the post increment is performed on \( m \) resulting in a value of 4 for \( m \). Then, the second expression \((n++ == 2)\) is evaluated and found to be FALSE. Since, this is FALSE, the post increment is performed on \( n \) resulting in a value of 3 for \( n \). Finally, the last expression, \((p++ == 1)\) is evaluated and found to be TRUE, yielding a TRUE value for the entire logical expression. The post increment is performed on \( p \) resulting in a value of 2 for \( p \).

### 12.7 Character Classification Functions

The following functions are a part of the standard library functions in CTYPE.H. They are used to test for the class of a character. They each take one character as an argument and return a value of true (non-zero int), if the character argument falls into the particular class of characters.

- `islower(c)` returns TRUE if \( c \) is a lowercase character.
- `isupper(c)` returns TRUE if \( c \) is an uppercase character.
- `isalpha(c)` returns TRUE if \( c \) is an alphabetic character, i.e., this is the same as `islower(c) \lor isupper(c)`
- `isdigit(c)` returns TRUE if \( c \) is a decimal digit.
- `isalnum(c)` returns TRUE if \( c \) is an alphabetic character or a digit.
isxdigit(c) returns TRUE if c is a hexadecimal digit.
isspace(c) returns TRUE if c is a whitespace character, i.e., equals space, newline, formfeed, carriage return, tab, or vertical tab.
iscntrl(c) returns TRUE if c is a control character.
In ASCII, these are the characters from 0 to 31 and 255 (DEL).
isgraph(c) returns TRUE if c is a graphic character, i.e., any printing character except the space character.
In ASCII, these are the characters from 33 to 254.
isprint(c) returns TRUE if c is a printing character including the space character.
In ASCII, these are the characters from 32 to 254.
ispunct(c) returns TRUE if c is a graphic character but not a letter or a digit.
i.e., this is the same as isgraph(c) && !isalnum(c).

12.8 Comparing Strings

Strings are compared by comparing successive characters in the strings against each other until an inequality has been established. The following examples illustrate a number of TRUE statements.

"CAT" > "BAT" since, 'C' > 'B'
"cat" > "CAT" since, 'c' > 'C'
"Apple" < "Art" since, 'A' equals 'A' and 'p' < 'r'
"Zen" > "Pandemonium" since, 'Z' > 'p'
"Team" > "Tea"

Strings can be compared using the library functions strcmp and strncmp defined in the <string.h> header file.

int strcmp(s1,s2)
returns an int value < 0 if s1 < s2, 0 if s1 == s2, and > 0 if s1 > s2.

int strncmp(s1,s2,n)
compares at most n characters of string s2 to string s1 and
returns an int value < 0 if s1 < s2, 0 if s1 == s2, and > 0 if s1 > s2.

The program STRCOMP.C shown below illustrates the use of these two functions.

Program 12.1 – STRCOMP.C  Lexicographic comparison of strings

/* Simple string comparisons.                      STRCOMP.C ==...*/
#include <stdio.h>
#include <string.h>

void main()
{
    printf("\n CAT BAT %d",strcmp("CAT","BAT"));
}
# 12.9 Conditional Operator

Sometimes, we need to choose between two expressions based on the truth value of a third logical expression. The conditional operator can be used in such cases.

\[
\text{logexpr} \ ? \ \text{exprT} : \ \text{exprF}
\]

If the logical expression \text{logexpr} evaluates to TRUE, the result is \text{exprT}.

If the logical expression \text{logexpr} evaluates to FALSE, the result is \text{exprF}.

\text{exprT} and \text{exprF} can be any kind of expressions.

\textbf{EXAMPLE 1}

\[
m = (n==4)? 2 : 3;
\]

Here, \(m\) is assigned a value of 2 if \(n\) equals 4. Otherwise, it is assigned a value of 3.

\textbf{EXAMPLE 2}

\[
printf("The integer %d is %s",n,(n\%2==0)?"even":"odd");
\]

If \(n\) is an even integer, then \(n\%2\) equals 0 and the string constant "even" is printed. Otherwise, the string constant "odd" is printed.

\textbf{EXAMPLE 3}

\[
printf("Marks = %5.1f Status: %s",
marks,(marks<35.0)?"FAIL":"PASS");
\]

If the value of \(marks\) is less than 35.0, the string constant "FAIL" is printed. Otherwise, the string constant "PASS" is printed.

\textbf{EXAMPLE 3}

\[
printf("There %s %2d student%s.
(n>1):"are":"is", n,(n>1)?"s","";
\]

can be use to produce grammatically correct output.

There are 36 students.
There is 1 student.
a ? b : c ? d : e
is interpreted as
a ? b : (c ? d : e)

12.10 Points to Remember

** Review Quiz **

** exercises from pp. 101, Kumar and Agrawal

** Programming Exercises **
17

Operator Precedence

13.0 Lesson Goals

13.1 Operator Precedence

When an expression uses a number of operators, there must be a predictable order in which they will perform their operations. For example, let us look at the following expression

\[ x + y / 4 \]

Obviously, the results will depend on whether the addition or the division is performed first. The ANSI standard specifies that the division operator has higher precedence than the addition operator. Therefore, in the above expression, the division is performed first.

It is important to keep in mind the operator precedence of the various operators in the C language while writing expressions that will behave in a predictable fashion. Let us examine another expression

\[ x + y - 4 \]

Which of the two operations will be performed first? Both the operators have an equal precedence in C and, therefore, operator precedence cannot answer this question. The sequence of operations in such cases depends on the direction of associativity. Addition and subtraction associate from left to right. In the above example, this left to right associativity implies that the addition will be performed before the subtraction.

The precedence of the various operators in the C language along with their associativity is given in Table 13.1. We are yet to meet some of these operators but they are given here for the sake of completeness. These operators are explained in detail in later chapters. All operators belonging to a precedence group have the same precedence.

It should be noted here that parentheses ( ) can be used to force a higher precedence of evaluation for any part of an expression. For example, in the expression

\[ (x+y)/4 \]
the addition is performed before the division because of the parentheses. Even when the operator precedence is clearly stated in the ANSI C standard, it is good practice to use parentheses to indicate the order of evaluations in a clear readable fashion. This improves the readability of the programs. For instance,

\[(j++) + (k++)\] is definitely more readable than \[j++ + ++k\]

### Table 13.1 Precedence and Associativity of Operators

<table>
<thead>
<tr>
<th>Operator Precedence Group</th>
<th>Associativity from</th>
</tr>
</thead>
<tbody>
<tr>
<td>( )</td>
<td>left</td>
</tr>
<tr>
<td>[ ]</td>
<td>left</td>
</tr>
<tr>
<td>.</td>
<td>left</td>
</tr>
<tr>
<td>-&gt;</td>
<td>left</td>
</tr>
<tr>
<td>!</td>
<td>right</td>
</tr>
<tr>
<td>~</td>
<td>right</td>
</tr>
<tr>
<td>++</td>
<td>right</td>
</tr>
<tr>
<td>--</td>
<td>right</td>
</tr>
<tr>
<td>+</td>
<td>right</td>
</tr>
<tr>
<td>-</td>
<td>right</td>
</tr>
<tr>
<td>*</td>
<td>right</td>
</tr>
<tr>
<td>(type)</td>
<td>size of</td>
</tr>
<tr>
<td>*</td>
<td>left</td>
</tr>
<tr>
<td>/</td>
<td>left</td>
</tr>
<tr>
<td>%</td>
<td>left</td>
</tr>
<tr>
<td>+</td>
<td>left</td>
</tr>
<tr>
<td>-</td>
<td>left</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>left</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>left</td>
</tr>
<tr>
<td>&lt;</td>
<td>left</td>
</tr>
<tr>
<td>&lt;=</td>
<td>left</td>
</tr>
<tr>
<td>&gt;</td>
<td>left</td>
</tr>
<tr>
<td>&gt;=</td>
<td>left</td>
</tr>
<tr>
<td>==</td>
<td>left</td>
</tr>
<tr>
<td>!=</td>
<td>left</td>
</tr>
<tr>
<td>&amp;</td>
<td>left</td>
</tr>
<tr>
<td>^</td>
<td>left</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>left</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>? :</td>
<td>right</td>
</tr>
<tr>
<td>=</td>
<td>right</td>
</tr>
</tbody>
</table>
### 13.2 The Comma Operator

The comma operator combines two expressions into a single expression. It appears as a pair of expressions separated by a comma.

\[
\text{left\_expr, right\_expr}
\]

When the comma operator is evaluated, the following sequence of steps takes place:

- The comma operator evaluates from left to right, i.e., `left\_expr` is evaluated first.
- All side effects of evaluating `left\_expr` are completed.
- The resultant value of `left\_expr` is discarded.
- `right\_expr` is evaluated.
- The value and type of the final result is the value and type obtained by evaluating `right\_expr`.

Let us look at an example where `k=4` is an `int` and `x` is a `float`.

\[
x = k++, k+3
\]

The evaluation of the left operand assigns a value of 4 to `x`. The side-effect of evaluating the left operand consists of applying the post-increment operator to `k` resulting in a value of 5 for `k`. The result of evaluating the left operand is a `float` value of 4.0. This value is discarded. The resultant of evaluating the right operand is an `int` value of 8. Therefore, the above expression evaluates to an `int` value of 8.

Let us look at a function call involving the comma operator (from Kernighan and Ritchie **).

```
foobar(a, (t=3, t+2), c)
```

This is equivalent to the following code:

```
t=3;
foobar(a, t+2, c);
```

Let us see another example using the comma operator.

```
int k=1;
float x=3.2;
long m;
double d;
d = (k++, x *= 3, m = k+x);
```
The final result of using the comma operator on the three expression will be the `long` value of `m` which equals 11. The comma operator can be used for creating `for` loops with two loop control variables as shown below.

```c
for(j=0, k=n; j<n; j++, k--)
```

### 13.3 Evaluation Sequence of Operands

The order of evaluation of the operands for the operators `&&`, `||`, `?:`, and `,` (comma operator) are clearly defined. For the other operators, the order of evaluation of the operands is implementation dependent. For example, in a statement involving calls to two different functions like

```c
z = foo1(x) + foo2(x);
```

we cannot be sure as to which function call is executed first. If one of the functions changes the value of `x`, this will lead to an implementation dependent value for `z` which is bad programming style. Portability (i.e., implementation independent behavior) is an essential feature of good C programs. In such cases, intermediate temporary variables should be used to force a particular sequence of evaluations. For example, the above statement could be rewritten as

```c
temp = foo2(x);
z = foo1(x) + temp;
```

Another such implementation dependent feature is the order of evaluation of the arguments of a function. For example, in the function call

```c
foobar(++n, pow(2,n));
```

the result is obviously dependent on the sequence of evaluation of the arguments to the functions. A more portable and implementation-independent code is shown below

```c
++n;
foobar(n, pow(2, n));
```

Another example of a statement that can lead to different interpretations on different implementations is shown below (from K & R)

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

As a general rule, one should avoid writing any code that depends on an implementation-dependent order of evaluation.

### 13.4 Points to Remember

**Review Quiz**
Programming Exercises
14.0 Lesson Goals

14.1 Control Flow

In Chapter 12, we learnt how to create logical expressions. The evaluation of a logical expression results in a truth value of either TRUE or FALSE. We can use these results to control the flow of a program in various ways. Using such flow control, we can develop complex programs.

Flow of control can be broadly classified into branching control and looping control. In this chapter, we will look at branching and, in the next chapter, we will learn about looping.

14.2 Statements and Blocks

A valid expression terminated with a semi-colon (;) forms a simple statement. Some simple statements are shown below:

```c
printf("Hello");
k++;
x = y + z;
```

We can combine simple statements into a compound statement or a block by placing them between braces (i.e., '{' and '}') as shown in the example below:

```c
{printf("Hello");
k++;
k *= 5;
}
```

Note that there is no semi-colon at the end of a block. Notice that all the statements of `main()` form a block.

```c
void main()
```
In the following sections, we will study the use of statements in control flow constructs. In all of them, we can use either a simple statement or a block. Later, we will see that blocks can be placed inside blocks creating nested block structures.

### 14.3 If-Else Statement

The if-else provides for conditional execution of a statements or statements in a program based on the truth value of a logical expression. The basic syntax is

```c
if (logexpr)
    statement1;
else
    statement2;
```

Here, `statement` refers to either a simple statement or a block. The if-else statement is shown as a flowchart in Figure 14.1.

The logical expression, `logexpr`, is evaluated first. If it is TRUE, `statement1` is executed. If it is FALSE, `statement2` is executed. For example,

```c
if( x > 0.0)
    rootx = sqrt(x);
else
    { 
        printf("ERROR: Square root of negative number cannot be found");
```

![Figure 14.1 Flowchart for if-else statement](image-url)
exit(EXIT_FAILURE);
}

The else part of the if-else is optional. Therefore, the else part may not be specified in some cases resulting in a plain if statement as shown in the flowchart of Figure 14.2.

```plaintext
if (logexpr) statement;
```

The logical expression, logexpr, is evaluated. If it is TRUE, statement is executed. Otherwise, nothing is done. For example,

```c
if(n < 0) printf("The number is negative");
```

### 14.4 Nested If-Else Statements

The else part of an if-else itself may be an if-else statement. These are known as nested if-else statements. An example is shown in Figure 14.3.
A decision involving choice between multiple alternatives can be expressed using such a nested if-else statement as shown below.

```c
if (logexpr1)
    statement1;
else if (logexpr2)
    statement2;
else if (logexpr3)
    statement3;
else if (logexpr4)
    statement4;
else
    default_statement;
```

Each of the logical expressions is evaluated and if any of these expressions happens to be TRUE, the associated statement is executed. If none of the logical expressions is TRUE, the execution comes to the last else and the default statement is executed. The default statement provides a useful method for catching errors by indicating "impossible" situations when they happen. It is good programming practice to always provide for a default statement.

Very often, the pairing of an else with an if might be ambiguous as shown in the example below (with all whitespace removed).

```c
if (logexpr1) if(logexpr2) statement1; else statement3;
```

Figure 14.3 Flowchart for a nested if-else statements
The question that arises is whether the \texttt{else} is associated with the first \texttt{if} or the second \texttt{if}? The rule here is that an \texttt{else} is associated with the closest previous unmatched \texttt{if}. Therefore, the above code should be interpreted as

```c
if (logexpr1)
    if(logexpr2)
        statement1;
    else
        statement3;
```

In such cases, where there is scope for an ambiguous interpretation, it is best to enclose the inner \texttt{ifs} in braces (creating blocks) as shown in the examples below.

```c
if (logexpr1)
    {
        if(logexpr2)
            statement1;
        else
            statement3;
    }
```
or if it you intened to denote a different relationship

```c
if (logexpr1)
    {
        if(logexpr2)
            statement1;
    }
else
    statement3;
```

### 14.5 Switch Statement

A switch statement is used for choosing one branch out of many possible branches. It is written as

```c
switch (expression)
{
    case const1: statement1;
    case const2: statement2;
    default: default statement;
    case const3: statement3;
}
```

\texttt{expression} is any integer type expression. \texttt{const1, const2, const3}, etc., are integer type constants. If the \texttt{const} value for some \texttt{case} matches the value of \texttt{expression}, execution begins at the \texttt{statement} corresponding to that case. If the value of the expression does not match any of the case values, execution begins at the \texttt{default} case.

The \texttt{default} case if optional. If the value of the expression does not match any of the case values and the \texttt{default} case is not given, then nothing is done by the switch statement. It is good
programming to always include a default case. The default case may appear anywhere in the list of cases but it is usually placed at the end.

Once execution begins at any one of the cases of a switch statement, all the following statements are also executed. This is known as fall through. A break statement placed anywhere stops the fall through execution by forcing termination of the switch at that point. Let us study the flowchart given in Figure 14.4 to understand fall through and the break statement better.

If the value of the expression e equals the constant value c1, then execution begins at statement1 and falls through the other statements of the switch statement until a break statement is encountered. This results in the execution of the following statements in the sequence shown:

```
statement1;
statement2;
```

If the value of the expression e equals the constant value c2, then execution begins at statement2 and falls through the other statements of the switch statement until a break statement is encountered. This results in the execution of the following statements in the sequence shown:

```
statement2;
```
If the value of the expression $e$ equals the constant value $c_3$, then execution begins at `statement3` and falls through the other statements of the `switch` statement. This results in the execution of the following statements in the sequence shown:

```
statement3;
default statement;
statement4;
```

If the value of the expression $e$ equals the constant value $c_4$, then execution begins at `statement4` and falls through the other statements of the `switch` statement. This results in the execution of the following statements in the sequence shown:

```
statement4;
```
If the value of the expression \texttt{e} does not equal any of the given case values (i.e., \texttt{c1}, \texttt{c2}, \texttt{c3} and \texttt{c4}), then execution begins at \texttt{default statement} and falls through the other statements of the \texttt{switch} statement. This results in the execution of the following statements in the sequence shown:

\begin{verbatim}
    default statement;
    statement4;
\end{verbatim}

The program, \texttt{FALLTHRU.C}, demonstrates the fall through feature of the switch statement. It also demonstrates the use of the default case.

Program 14.1 – \texttt{FALLTHRU.C} Program showing fall through in switch statement

\begin{verbatim}
/* ============================= FALLTHRU.C ========================
   Illustrates fall through in switch statement.
  =================================================================* /
#include <stdio.h>
void main()
{
    int j;
    printf("\nEnter an integer value : ");
    scanf("%d",&j);
    switch(j)
    {
    case 0: printf("\ncase 0");
    default: printf("\ndefault case");
    case 1: printf("\ncase 1");
    case 2: printf("\ncase 2");
    }
}
\end{verbatim}

Failure to account for the fall through nature of the \texttt{switch} statement is a common source of errors. You must remember to use the \texttt{break} statement to prevent fall through according to the requirements of your program.

The most common form of a switch statement is shown in Figure 14.5. In this construction, you will notice that each \texttt{case} is terminated with a \texttt{break} statement preventing any form of fall through. You will also notice the placement of the \texttt{default} case as the last case in the switch statement.

The logic of Figure 14.5 can be coded as

\begin{verbatim}
switch(e)
{
    case \texttt{c1} : statement1;\texttt{break};
    case \texttt{c2} : statement2;\texttt{break};
    case \texttt{c3} : statement3;\texttt{break};
    default : \texttt{default statement};
}
\end{verbatim}
14.6 Program Indentation

Proper use of indentation in the text of a program improves the readability of a program and facilitates the finding of bugs in the program. It also clarifies which statements go with which control flow statements. The usual practice is to use 4 spaces for indentation as shown below.

```c
if ( n > 0 )
    printf("\nIt is a positive number");
else
    printf("\n%d is not positive number",n);
```

The plain if statement can be indented in either of the two ways shown below depending on the length of the statement.

```c
if (n > 0 && n < 999) m = p * 4;
if (n > 0 && n < 999)
    printf("\nValid input has been entered.");
```
Nested if-else statements can be indented in two ways depending on the levels of nesting. When few levels of nesting are present, the following style can be used

```c
if (n > 0)
    m = p++;
else
    if(n < 0)
        printf("\nIllegal negative value");
    else
        m = p * p;
```

But when a large number of levels are present, we treat else-if as a single keyword for the purpose of indentation.

```c
if (marks > 80)
    grade = 'A';
else if (marks > 65)
    grade = 'B';
else if (marks > 50)
    grade = 'C';
else if (marks > 35)
    grade = 'D';
else
    grade = 'F';
```

Indentation of blocks is absolutely essential for ease of debugging and improving readability. The indentation style used for blocks varies from one programmer to another. Choose a style that you like from some book or devise your own style. Whichever style you select or devise, the important thing is to use it consistently in all your programs.

*Indentation style in this book (this is known as the Allman style and is popular in Europe)*

```c
if( x < 0.0) {
    printf("\nERROR: Square root of -ve number.");
    exit(1);
}
```

*Indentation style used in K & R*

```c
if( x < 0.0){
    printf("\nERROR: Square root of -ve number.");
    exit(1);
}
```

*Another style*

```c
if( x < 0.0) {
    printf("\nERROR: Square root of -ve number.");
    exit(1);
}
```

### 14.7 goto Statement

The goto statement is used to "jump" to another part of the program marked with a label. It has the syntax

```c
goto label_name;
```
goto label;
    . . .
label: . . .

The label is any valid identifier. The label can be placed anywhere in the module. It may appear at some place after the goto statement or before it. When the goto is encountered, the control is transferred to the statement marked with the label.

Indiscriminate use of the goto creates what is known as "spaghetti code" where it is extremely difficult to track the execution sequence of the statements in the program. In addition, it can be shown that anything written using a goto can be written in some alternative fashion without the goto. In fact, some of the newer programming languages have no equivalent for the goto. Therefore, it is good programming style, from the point of readability as well as maintainability of code, to avoid the goto as much as possible.

But the avoidance of the goto should not become a point of dogmatic belief. In certain cases, the goto may be the most elegant solution to the problem and, in such cases, one should not hesitate in using a goto statement. A good example example is one involving exiting from a deeply nested structure.

14.8 Points to Remember

Review Quiz

Programming Exercises
19

Looping

15.0 Lesson Goals

15.1 while Loop

In this chapter, we will look at various methods for performing repeated tasks, i.e., doing the same thing again and again until some criterion is satisfied. The while statement (whose flowchart is shown in Figure 15.1) provides for one type of looping operation.

while (logexpr)

statement;

The logical expression is evaluated. If it is found to be TRUE, the statement is executed. (As usual, the statement here may be a simple statement or a block.) Following this, the logical expression is evaluated again. If it is again found to be TRUE, the statement is executed once again. This continues until the logical expression yields a FALSE value in some. When that happens, the while loop is terminated.

The statement of the while loop, usually a block of statements, is also referred to as the body of the loop. It is obvious that the body of the loop must contain some statements that effect the truth
value of the logical expression. Otherwise, the logical expression will either be always TRUE (resulting in an infinite loop) or it is always FALSE (resulting in nothing being done by the \texttt{while} statement).

If the logical expression is found to be FALSE the very first time, the \texttt{while} statement does nothing, i.e., the body of the loop is not executed even once.

The following code segment prints out the entire uppercase alphabet in one line.

\begin{verbatim}
int k;
k = 0;
while(k<26) {
    printf("%c",k+'A');
k++;
}
\end{verbatim}

\section*{15.2 \texttt{do-while} Loop}

In the \texttt{while} loop, the logical expression is evaluated before executing the body of the loop. If the body of the loop is executed before the evaluation of the logical statement, we get the \texttt{do-while} statement shown in Fig.15.2.

\begin{verbatim}
do statement while (logexpr);
\end{verbatim}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{do-while.png}
\caption{Flowchart for a do-while statement}
\end{figure}

In the \texttt{do-while} loop, it can be seen that the body of the loop (which can be a simple statement or a block) will be executed at least once. Generally, the \texttt{while} loop is preferred over the \texttt{do-while} loop but the choice depends on the nature of the problem.

The following code segment prints out the entire uppercase alphabet in one line using a \texttt{do-while} loop.
int k;
k = 0;
do
    { printf("%c",k+'A');
k++;
    }
while (k < 26);

Very often, we would like to validate a value input by the user to make sure that it satisfies certain criteria. A do-while loop can be used effectively in such cases. In the following code segment, the user is forced to input an integer value that lies between 1 and 100.

```
int num, isvalid;
/* Accept and validate user input for num */
do
    { printf("\nEnter an integer between 1 and 100 : ");
scanf("%d",&num);
    isvalid = ( num >=1 && num <=100);
    if( !isvalid )
        printf("\nERROR: Invalid input, try again\n");
    }
while(!isvalid);
```

15.3 for Loop

A for loop can be seen as an augmented while loop. Let us reexamine the example code segment of Section 15.1.

```
k = 0; \leftarrow \text{initialization expression}
while(k<26) \leftarrow \text{while loop with logical expression}
    { printf("%c",k+'A');
k++; \leftarrow \text{end-of-loop expression}
    }
```

There are three significant parts in the code segment shown above:

1. an initialization operation
2. a while loop with its logical expression
3. an operation to be performed at the end of each loop.

These three operations occur very frequently in while loops and the for loop provides a concise and elegant notation for combining these three steps into a single statement as follows

```
for (init_expr; logexpr; loop_expr)
    statement;
```

where init_expr is the initialization expression,
logexpr is a logical expression,
loop_expr is an expression to be performed at the end of every loop and statement is the body of the loop (either a simple statement or a block).

A for loop is exactly equivalent to the following code using a while loop as shown in Fig. 15.3:

```c
init_expr;
while (logexpr)
{
    statement;
    loop_expr;
}
```

![Flowchart for a for loop](image)

**Figure 15.3 Flowchart for a for loop**

We can now rewrite the example shown above using a for loop as follows:

```c
for (k = 0; k<26; k++)
    printf("%c",k+'A');
```

The for statement shown above also happens to be the simplest and most common one. It is used to repeat a task n times using a counter variable k.

```c
for (k=0; k < n; k++)
```
Notice that the counting is not done as 1, 2, 3, . . . but as 0, 1, 2, . . . To become a C programmer, you must become familiar with this unusual counting scheme starting from 0 and going up to \((n-1)\). You could write an equivalent for loop as

\[
\text{for } (k=1; k <= n; k++) \\
\hspace{1cm} /* \text{Not recommended} */
\]

but you will never be given admission to the exclusive club of C programmers! This habit of counting from 0 may seem a little strange in the beginning but it leads to very elegant programs when we start using pointers and arrays (Chapters 21 and 22).

**All three expressions in the for statement are optional.** The for statement

\[
\text{for(;logexp;)}
\]

is exactly equivalent to a while statement. When the logical expression is not specified, it is assumed to have a default value of \text{TRUE} and results in an infinite loop as explained in Section 15.6.

### 15.4 break and continue Statements

We had seen the use of the break statement to terminate a switch statement in Chapter 14. The break statement can also be used in a loop to force immediate termination of the loop as shown in the program, \text{PRIME.C}.

**Figure break statement \(\text{*** pp.110, hutchison}\)**

The program \text{PRIME.C} takes an integer, \text{num}, as an input and checks whether or not it is a prime number. For this, we first assume that the number \text{num} is a prime. Then, we check to see if it is divisible by any integer between 2 and \((n-1)\). If it is divisible, then the number is not a prime number and we need not do any more work. Therefore, we make an immediate exit from the loop using a break statement.

**Program 15.\(\ast\) – \text{PRIME.C} Checking whether a number is prime**

```c
/* \text{PRIME.C} \text{Checking for prime numbers.} */
#include <stdio.h>

int main()
{
    int num, k, isprime=1;
    /* Accept and validate user input */
    do
    {
        printf("\nEnter a positive integer (>1) :");
        scanf("\%d",\&num);
        if(num < 2) printf("\nERROR: Invalid input, try again");
    }
    while(num < 2);

    return 0;
}
```

134
/* Check for prime */
for(k=2;k<num;k++)
{
    if(num % k == 0)
    {
        isprime=0;
        break;
    }
}
printf("%d is %s a prime number",num,isprime?""":"NOT");

The _continue_ statement ends the current instance of the loop and begins the next instance.

**Figure continue statement**, pp. 111, hutchison

Examine the program segment given below,

    for (k = 0; k < 5; k++)
        { printf("First");
          if(k%3 == 2) continue;
          printf("Second");
          if(k%3 == 1) continue;
          printf("Third");
        }

The output from the above program segment is shown below (bolotype text). Unlike the break statement, the looping itself is not terminated by the continue statement but only the particular instance of the loop is terminated.

**First** ← k=0, k%3 yields 0  
**Second**  
**Third** ← normal end of loop for k=0  
**First** ← k=1, k%3 yields 1  
**Second** ← end of loop for k=1 forced by _continue_ statement  
**First** ← k=2, k%3 yields 2, end of loop for k=2 forced by _continue_ statement  
**First** ← k=3, k%3 yields 3  
**Second**  
**Third** ← normal end of loop for k=3  
**First** ← k=4, k%3 yields 1  
**Second** ← end of loop for k=4 forced by _continue_ statement  
**First** ← k=5, k%3 yields 2, end of loop for k=5 forced by _continue_ statement

### 15.5 Null Statements in Loops

In some cases, we may have a situation where we have a loop that does not have a body. In such cases, the loop is said to have a _null statement_. A _while_ loop without a body is shown in Fig.15.4.
Figure 15.4 A while statement with null statement

In such cases, the logical expression must have some mechanism for changing itself. The following code segment demonstrates how to read in a string containing whitespace (an alternative to using the `gets` function).

```c
char name[80];
int k=0;
printf("Enter your full name :\n");
while((name[k++]=getchar())!='\n')
    /* null statement */
name[k-1] = '\0';
printf("\n%s",name);
```

Notice the special indentation used for the null statement. The indentation and the accompanying comment make it very clear that the null statement was placed intentionally and is not a bug. A common bug arises when a semi-colon is placed at the end of the `while` statement by mistake as shown below.

```c
while (logexpr);
```

The `statement` is no longer a part of the `while` loop because the semi-colon indicates a null statement for the `while` loop. A similar error might arise with a `for` loop written as follows

```c
for (; ; ;);
```

### 15.6 Infinite Loops

When the logical expression used in a `while`, `do-while` or `for` loop remains TRUE forever, the loop is repeated infinitely (until a hardware interrupt is used or the computer is shut off). These can be serious bugs in programs.
However, on some occasions, we may want to intentionally create an infinite loop where the loop is terminated by means of a `break` statement placed inside the body of the loop or in some other manner (e.g., using the `exit` function). This can be done using either a `while` loop or a `for` loop.

```c
while (1)  // the preferred way to implement an infinite loop
or
for(;1;)
```

Yet another way to implement an infinite loop uses the fact that in the absence of a logical expression in a `for` loop, the logical expression is taken to be TRUE by default. Therefore,

```c
for(;;)
```

is equivalent to `for(;1;)` and is a valid infinite loop but it is not very good from the readability point. The recommended method of creating an infinite loop is to use `while(1)`. Most compilers will generate some kind of warning on this statement but you can safely ignore this warning.

### 15.7 Points to Remember

**Review Quiz**

**Programming Exercises**
20

Programs using Branching and Looping

16.0 Lesson Goals

16.1 Finding Special Integers

A Curious Number: If you split the number 3025 into two parts like this

\[
\begin{array}{c}
30 \\
25
\end{array}
\]
and add the two parts together:

\[
30 + 25 = 55
\]

and square the result: \(55^2\), you get 3025, the original number! There are two other four digit numbers that you can play with in this way. Which are the two numbers?

(Solutions: 2025 & 9801) (from THE HINDU, 29th March, 1997)

We will begin by looking at two different ways to attack this problem. Each represents a different algorithm for the solution of our problem.

Algorithm I - We want to check all 4 digit integers, i.e., integers from 1000 to 9999. For each of these integers, we need to obtain two integers - the first made of the first two digits and the second made of the last two digits. How do we perform this separation? The modulus operator can be used to separate the last two digits.

\[
\begin{array}{c}
n \% 100 \text{ separates last two digits,} \\
e.g., 3025 \% 100 = 25, 2178 \% 100 = 78
\end{array}
\]

The first two digits can be separated by dividing the number by 100 and using the truncation property of division of integers.

\[
\begin{array}{c}
n / 100 \text{ separates the first two digits} \\
e.g., 3025 / 100 = 30, 2178 / 100 = 21
\end{array}
\]
Having obtained the two smaller integers, we can perform a check to see if the given condition is met.

**Algorithm II** - We know that the larger integer can be constructed from the smaller integers. The first two digits make up an integer between 10 and 99. The last two digits can vary between 00 and 99. The larger 4 digit integer can be constructed from these two integers

\[
\text{large} = \text{first} \times 100 + \text{second}
\]

With these three integers in hand, we can now perform a check if the given condition is being met.

*There might be more than one algorithm to solve a problem. You have to choose one of them to implement your solution.*

We will implement the second algorithm here. We will call our program FRACTADD.C.

The major steps in this C program are as follows:

1. Declare three int variables - first, second, num
2. We will need an int variable sum to store the sum of first and second
3. We will need a variable sqrs to store the square of the sum.
4. We will use a for loop to try out all possible values of first from 10 to 99.
5. We will use another for loop nested inside the first loop to try every possible value of second for each value of first.

```c
/* ============================= CURIOUS.C =======================
   Finding numbers such as 3025 which have the following property:
   3025 => 30+25 = 45
   45 * 45 = 3025
   ==============================================================*/

#include <stdio.h>

void main()
{
    int first,second,sum,sqrs,num;
    for(first=10;first<=99;first++)
        for(second=0;second<=99;second++)
            {
                sum = first+second;
                sqrs = sum*sum;
                num = first*100 + second;
                if(sqrs == num) printf("Found %d\n",num);
            }
}
```

16.2 Computing Square Roots

Problem Statement: Write a program to calculate the two roots of a number. The number may be negative, in which case you must print the two imaginary roots using the symbol 'i' to indicate \(\sqrt{-1}\).
Let us look at some numerical values and the expected output from the program.

Given an input of 9.0, we expect the two roots +3.0 and -3.0 to be printed.  
Given an input of -9.0, we expect the two roots +3.0i and -3.0i to be printed.  
Given an input of 0.0, we expect the two roots to be 0.0.

The sqrt function in the standard function library returns the positive square root of a non-negative double argument. Therefore, we must generate the negative root by using the unary negation operator (-) on the positive root returned by the sqrt function.

If the input value x is negative, we will find the roots of -x and attach the symbol "i" in the output printf function.

We will use an if-else-if construct to consider the three different cases in turn, i.e., x is positive, x is zero, and x is negative.

Program 16.2 – SQROOT.C Computing real or imaginary square roots of a number

```c
/* Finding square roots for any real number. */
#include <stdio.h>
#include <math.h>

void main()
{
    double x, root;
    printf("Enter a real number :");
    scanf("%lg", &x);
    if (x > 0.0)
    {
        root = sqrt(x);
        printf("The square roots of %lg are %lg & %lg", x, root, -root);
    }
    else if (x == 0.0)
    {
        printf("The square root of 0.0 is 0.0");
    }
    else
    {
        root = sqrt(-x);
        printf("The square roots of %lg are %lgi & %lgi", x, root, -root);
    }
}
```

Note the clever use of the printf format specification string to attach the symbol "i" to the roots. For the computer, this symbol has no special significance. However, the presence of the symbol in the output has a special significance to the user because of its predefined significance to complex numbers.
### 16.3 Sum of Integers

Problem Statement: Write a program to find the cumulative sum of integers input by the user.

First, we need to enquire from the user how many values need to be summed up. We will store this in an integer `num`.

Next, we need to keep a count of the number of values being input by the user. We will use an integer counter variable, `kount`, for this purpose. The variable `kount` will control the execution of a `for` loop set up to run `num` times.

Finally, we need a variable named `sum` to store the cumulative sum of the values. Note that this variable `sum` must be initialized to a value of 0.0 before we start adding the values. **Initialization of variables plays a very important role in any program.** Remember that an uninitialized variable might contain some unknown garbage value. Once a variable `sum` has been declared, it is allocated some bytes in the memory. The value stored in these bytes may or may not be indicate a value of 0.0 for `sum`. Some computers will initialize all variables to zero but this is not true of all computers. Therefore, it is essential to initialize variables before using them.

#### Program 16.3 – NSUM.C Computing sum of integers

```c
/* Finding the sum of n integers. */
#include <stdio.h>

void main()
{
    int num,kount,value;
    long sum;

    printf("How many numbers to add : ");
    scanf("%d",&num);

    sum=0; /* Initialize sum to zero */
    for(kount=0;kount<num;kount++)
    {
        printf("Enter integer #%3d > ",kount+1);
        scanf("%d",&value);
        sum += value;
    }

    printf("The cumulative sum is %ld",sum);
}
```

### 16.4 Product of Numbers

Problem Statement: Write a program to find the cumulative product of n real numbers input by the user.

This problem is very similar to that of Section 16.3 except that we must initialize the product to 1 instead of 0.
Program 16.4 – NPRODUCT.C  Computing product of numbers

/* ============================== NPRODUCT.C ====================
Finding the product of n real numbers.
================================================================*/
#include <stdio.h>

void main()
{
  int num,kount;
  double value,product;

  printf("\nHow many numbers to multiply : ");
  scanf("%d",&num);
  putchar(’\n’);

  product = 1.0;
  for(kount=0;kount<num;kount++)
  {
    printf("Enter value #%3d > ",kount+1);
    scanf("%lg",&value);
    product *= value;
  }

  printf("\n\nThe cumulative product is %lg",product);
}

16.5 Fibonacci Numbers

Problem Statement: The Fibonacci numbers are generated using a recurrence relation which defines each number in terms of its two predecessors as

\[ F_{k+2} = F_{k+1} + F_k \]

where \( F_0 = 1 \) and \( F_1 = 1 \) by definition. Therefore, \( F_2 \) equals 2, \( F_3 \) equals 3, and so on. Write a program to generate the first \( n \) numbers of the Fibonacci sequence.

Since, \( F_0 \) and \( F_1 \) are given values, we need to print them out without performing any calculations.

Let \texttt{first} and \texttt{second} be two successive numbers of the Fibonacci sequence. Then, the next number, \texttt{new}, is the sum of \texttt{first} and \texttt{second}.

We can now use the two numbers, \texttt{second} and \texttt{new} to generate the next Fibonacci number. For this purpose, we need to replace \texttt{first} and \texttt{second} with \texttt{second} and \texttt{new}. To be more precise, we need to replace \texttt{first} with \texttt{second} and then replace \texttt{second} with \texttt{new}. Now, we are ready to generate the next Fibonacci number.

Program 16.5 – FIBONACC.C  Computing Fibonacci Numbers

/* ============================== FIBONACC.C ====================
Printing the first n Fibonacci numbers.
================================================================*/
Programs using Branching and Looping

143

---

```c
#include <stdio.h>

void main()
{
    int num, kount;
    unsigned long first=1, second=1, new;

    printf("How many Fibonacci numbers to print : ");
    scanf("%d",&num);
    putchar('n');

    printf("Fibonacci number [ 0] is 1
    Fibonacci number [ 1] is 1
    
    for(kount=1;kount<num;kount++)
    {
        new = first + second;
        printf("Fibonacci number [%2d] is %ld",kount+1,new);
        first = second;
        second = new;
    }
}
```

---

** Printing non-Fibonacci numbers

16.6 Factorial Calculation

Problem Statement: Write a program to calculate the factorial, n!, of an integer n. The factorial of n is defined as

\[ n! = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n. \]

In many ways, this problem is similar to the problem in Section 16.4 where we calculated the cumulative product of n numbers. But instead of values being input by the user, we can use the loop counter variable itself as the value to multiply. We will use a variable named `factorial` to accumulate the product.

---

Program 16.6 – FACTORIL1.C Computing the factorial of an integer

```c
#include <stdio.h>

void main()
{
    int num,kount;
    unsigned long factorial;

    printf("Enter a small integer (<10) : ");
    scanf("%d",&num);

    factorial = 1;
    for(kount=1;kount<=num;kount++)
    {
```
factorial *= kount;
printf("\n%d => %ld",kount,factorial);
}
printf("\nThe factorial of %d equals %ld",num,factorial);

16.7 Sides of a Triangle

Problem Statement: Accept three integers representing the sides of a triangle. Write a program to check whether it is possible to construct a triangle with sides having these measures. In a triangle, the sum of any two sides cannot be less than the third side.

We call the program TRIANGLE.C. We use variables side1, side2, and side3 to store the three input integers. Then we check to see that all the three inequalities (each formed by taking two sides at a time) are satisfied. Notice the use of the logical AND operator to ensure that all three inequalities are satisfied. If any of the three inequalities results in FALSE, the triangle is declared to be impossible.

Program 16.7 – TRIANGLE.C Checking for validity of a triangle

/* ============================================================== TRIANGLE.C ==============================================================
   Checking whether the three integers input by user can represent the sides of a triangle.
   ==============================================================*/
#include <stdio.h>
void main()
{
 int side1, side2, side3;
 printf("\nEnter the 3 sides of the triangle :");
 scanf("%d %d %d",&side1, &side2, &side3);
 if( (side1+side2 >= side3) &&
 (side2+side3 >= side1) &&
 (side3+side1 >= side2) )
 printf("\nThese can be the sides of a triangle");
 else
 printf("\nImpossible measures for the sides of a triangle");
}

16.8 Reversing an Integer

Problem Statement: Write a program to accept an integer input by the user and print out the digits in reverse order. For example, if the input is 3025, the output should be 5203.

We will call the program REVINT.C.

How do we obtain the last digit of an integer? We can use the modulus operator for this purpose. If num is the original integer, then (num%10) will be the value of its last digit. For example,
(3025%10) equals 5. Since, we need to write only a single digit to the output, we use a %1d format conversion specification.

Having separated the last digit, how do we obtain the remaining part of the integer? We can divide by 10. For example, (3025/10) equals 302. Where do we store this new value? Since, we no longer need the old value of num, we can assign this new value of num to num itself.

\[
\text{num} = \text{num}/10; \text{ or } \text{num} /= 10;
\]

We see that repeated use of the last two steps will generate the digits of the integer in reverse order as required. But when do we stop? We stop when there are no more digits left in num, i.e., when num becomes equal to zero.

**Program 16.8 – REVINT.C Printing an integer in reverse**

```c
#include <stdio.h>

void main()
{
    unsigned long num;
    short int digit;

    printf("Enter an integer :");
    scanf("%lu",&num);

    printf("The digits in reverse are ");
    while(num != 0)
    {
        digit = num % 10;
        num /= 10;
        printf("%1d",digit);
    }
    putchar(\n');
}
```

**16.9 Generating a Pyramid of Digits**

Problem Statement: Write a program which, given an integer \( n \) as input, generates a pyramid of \( n \) lines using the last digit of the line number (starting from 1). The output for \( n=12 \) is shown below as an example.

```
1
222
3333
444444
55555555
6666666666
7777777777777
888888888888888
```

145
We will call the program PYRAMID.C. What are the major tasks to be completed to generate this pyramid for an input value of \texttt{num}?

1. Print \texttt{num} lines of text.
2. For each line, print a certain number of blank characters.
3. After the blank characters, print the last digit of \texttt{num} a certain number of times.

Let us look at line number \texttt{k}. How many spaces do we need to print at the beginning of line \texttt{k}? We see that for line number \texttt{num}, there are 0 blanks, line (\texttt{num}-1) has 1 blank and line number 1 has (\texttt{num}-1) blanks. We can generalize this to find that line \texttt{k} has (\texttt{num}-\texttt{k}) blank characters.

How many times do we need to print the last digit? Once for line 1, 3 times for line 2, 5 times for line 3, and so on. This can be written as (2\texttt{k}+1) for line \texttt{k}.

Now, we are ready to write three steps shown above in the form of nested loops.

\begin{verbatim}
loop (for all lines from 1 to \texttt{num})
  loop (for all printing all spaces at the beginning of the line)
  loop (for printing the last digit repeatedly)
\end{verbatim}

We use the variables \texttt{line} (same as \texttt{k} in the discussion above), \texttt{kount1}, and \texttt{kount2} as the loop counter variables for these three loops.

\textbf{Program 16.9 – PYRAMID.C} \ \ Generating a special pyramid pattern of characters

```c
/* Printing a pyramid of digits. */
#include <stdio.h>

void main()
{
    unsigned int num, lastdigit, nspaces;
    unsigned int line, ntimes, kount1, kount2;
    printf("\nEnter an integer : ");
    scanf("%lu", &num);
    for(line=1;line<=num;line++)
    {
        /* put spaces front */
        nspaces = num-line;
        for(kount1=0;kount1<nspaces;kount1++) putchar(' ');
        /* put the last digit */
        lastdigit = line%10;
        ntimes=2*line-1;
        for(kount1=0;kount1<lastdigit;kount1++) putchar(' ');
        for(kount1=0;kount1<ntimes;kount1++) putchar('*');
    }
}
```

146
### 16.10 Generating a Pattern of Characters

Problem Statement: Write a program which, given an integer \( n \) as input, generates a square made of characters as shown in the examples below.

**Output for \( n=4 \)**

```
abcd
a*#d
a*#d
abcd
```

**Output for \( n=3 \)**

```
abc
a*c
abc
```

We will call the program CHARSQR.C. We see that if \( n \) is the given integer, we are required to print the first \( n \) characters of the English lowercase alphabet. In other words, for 1 we print 'a', for 2 we print 'b', and so on. How do we obtain the letter given its serial number \( k \) in the alphabet? We simply use ('a'+\( k \)-1). This simple solution is possible because the char variable type in C is considered an integer type and this makes it possible to add an integer to a character in the C language.

Looking into the problem, we observe that we will need a loop to print the \( n \) lines. For each line, we have to perform three operations
1. Print 'a' at the beginning
2. Print the characters in the middle using a loop. For the first and the last line, these are successive letters of the alphabet. For the other lines, we need to print an asterisk ('*').
3. Print the last character based on \( n \).

These steps can be rewritten in the following pseudocode form.

```
loop (for all lines from 1 to \( n \))
    print 'a'
    if (first line or last line)
        loop (print \( n \)-2 characters starting from 'b')
    else
        loop (print the character '*' \( n \)-2 times)
    print last character based on \( n \)
```

---

**Program 16.10 – CHARSQR.C Generating a square pattern of characters**

```c
/* ==============================================================
 CHARSQR.C ==============================================================
 Printing a square of characters.
 ==============================================================*/
```
#include <stdio.h>

void main()
{
    unsigned int num, line, kount1, kount2;

    printf("Enter an integer greater than 1 : ");
    scanf("%lu",&num);

    for(line=1;line<=num && num>1;line++)
    {
        /* put 'a' in front */
        putchar('a');
        /* place the characters in the middle */
        for(kount1=0;kount1<num-2;kount1++)
        {
            if(line==1 || line == num)
                putchar('a'+1+kount1);
            else
                putchar('*');
        }
        /*place the last character*/
        printf("\n","a'+num-1);
    }
}

16.11 Computing Square Root by Iteration

To find the square root of a floating point number, we can use the sqrt function from the math library. But in this section, we will look at an iterative algorithm to obtain the square root of a number. We will then compare the result of our algorithm against the value obtained by the sqrt function. Our goal is to understand the fundamentals of iterative techniques and the use of looping and branching in the writing of iterative programs.

The Newton-Raphson algorithm to find the root of any function f(y) can be written as

$$y_{k+1} = y_k - \frac{f(y_k)}{f'(y_k)}$$

where f(y) is the function whose root we need, f'(y) is the first derivative of f(y) (i.e., df/dy) and y_k is the value of the root after the k-th iteration. Given an initial guess value, y_0, for the root of f(y), we can obtain y_1, y_2, y_3, ..., successively until the values of y stop changing or converge to the solution.

Let us now consider the particular case of

$$f(y) = y^2 - x$$

We see that if f(y) equals 0, then y equals the square root of x. We have transformed the problem of finding a square root of x into a problem of finding a root for f(y).

$$f'(y) = 2y$$
The Newton-Raphson formula for this problem can now be written as

\[ y_{k+1} = \frac{y_k^2 - x}{2y_k} = \frac{y_k - x/y_k}{2} \]

When do we stop these iterations? Do we wait until \( f(y) \) becomes exactly equal to 0.0? Owing to the finite precision used in the representation of real numbers in the computer, it may be impossible to obtain the exact root. Therefore, we should stop when \( f(y) \) becomes less than some small value \( \delta \). We can also stop if the change in value of \( y \) between two iterations becomes smaller than some small value \( \varepsilon \).

So far, we have assumed that the iterations will succeed in obtaining the root, i.e., the iterations converge. What if the iterations do not converge to the solution? We might end iterating for an infinite number of times and not find the root. For almost any iterative algorithm, there are cases where the algorithm will diverge. We must guard our program against such divergence. The simplest method is to set an upper limit on the number of iterations.

Let us review the termination criteria we have developed for the iterations

1. stop if number of iterations exceeds a maximum number \( M \).
2. stop if absolute value of \( f(y) \) becomes smaller than \( \delta \).
3. stop if absolute value of \( (y_{k+1} - y_k) \) becomes smaller than \( \varepsilon \).

Each of these values is defined as a symbolic constant in the program (MAXITER, DELTAMAX, and EPSMAX).

How do we store all the successive values of \( y \) obtained in this iterative procedure? Due to the nature of the algorithm and the termination criteria chosen by us, we really do not need to store all these values. We only need to know the two most recent values obtained during the iterations. We will use two variables, \( y_{old} \) and \( y_{new} \) for this purpose. The \( y_{new} \) value for this iteration becomes the \( y_{old} \) for the next iteration.

Let us take a look at another important issue in writing programs, namely, **input validation and error handling**. In our program, we will ask the user to input a number and a guess value for its square root. What if the user enters a negative value for either of them? Our program must be able to trap such illegal values for input and take some action based on the type of error. A do-while loop is one of the easiest way to repeatedly ask for user input until a valid response is obtained. The pseudocode for this is given below.

```c
    do
        get user input
        check for errors and give feedback to user on errors
    while (input is not valid)
```

The **do-while** loop is exited only when valid input is obtained. Let us look at a simple example of a C program segment which forces the user to input either Y for YES or N for NO.
do {
    printf("Enter Y for yes or N for no >> ");
    ans = getchar();
    if(ans != 'Y' && ans != 'N')
        printf("ERROR: Invalid input");
} while(ans != 'Y' && ans != 'N')

Program 16.11 – SQROOT.C  Computing square root by iteration

/*=============================== SQROOT.C ========================
Program to calculate square root using Newton-Raphson algorithm
=================================================================*/
#include <stdio.h>
#include <math.h>
/* TERMINATION PARAMETERS */
/* Maximum number of iterations */
#define MAXITER 200
/* stop if change between iterations is less than EPSMAX */
#define EPSMAX 1.0e-5
/* stop if fabs(y*y-x) < DELTAMAX */
#define DELTAMAX 1.0E-10
void main()
{
    double x,y,yold,delta,eps;
    int iter;
    /* accept user input */
    do
    {
        printf("Enter a positive number >> ");
        scanf("%lg",&x);
        printf("Enter guess value for the square root >> ");
        scanf("%lg",&yold);
        if(x<=0.0)
            printf("ERROR: number must be greater than 0.0");
        if(yold<=0.0)
            printf("ERROR: guess value must be greater than 0.0");
    } while(x <= 0.0 || yold <=0.0);
    /* begin iterative loop */
    for(iter=0;iter<MAXITER;iter++)
    {
        y = (yold+x/yold)/2.0;
        printf("Iteration #%3d, y = %22.16g",iter,y);
        eps = fabs(y-yold);
        delta = fabs(y*y-x);
        if(eps < EPSMAX) break;
        if(delta < DELTAMAX) break;
        yold = y;
        printf("Square root of %g = %22.16g",x,y);
        printf("Number of iterations = %d, MAXITER = %d",iter,MATIXTER);
        printf("eps = %g, EPSMAX = %g",eps,EPSMAX);
        printf("delta = %g, DELTAMAX = %g",delta,DELTAMAX);
    }
}
16.12 Iterative Solution of Nonlinear Equations

Let us look at another iterative algorithm for finding a root of the equation, $g(x)=0$. We rewrite the equation in a form, $x = f(x)$, by manipulating the terms of the equation. Then, we create the iterative scheme

$$x_{k+1} = f(x_k)$$

For example, given

$$x^5 + 3x^2 - 10 = 0,$$

we can rewrite it as

$$x = (10 - 3x^2)^{1/5}$$

The termination criteria are similar to those used for the Newton-Raphson iteration in Section 16.11. At the end of the program, we check for the convergence of the method and print the appropriate messages.

Program 16.12 – FINDROOT.C Computing a root by simple iteration

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

#define MAXITER 200
#define EPSMAX 1.0e-5
#define DELTAMAX 1.0E-5

int main()
{
    double y,yold,delta,eps,temp;
    int iter,stopflag,converged;

    printf("Enter guess value for the root >> ");
    scanf("%lg",&yold);

    while( !stopflag )
    {
        // accept user input */
        printf("\nEnter guess value for the root >> ");
        scanf("%lg",&yold);

        // begin iterative loop */
        iter=0;
        stopflag=0;
        converged=0;
        while( !stopflag )
        {
            // accept user input */
            yold=0;
            stopflag=0;
            converged=0;
            while( !stopflag )
            {
                // iterative loop */
                yold=xold;
                xold=f(xold);
                if( std::abs(xold-yold)<EPSMAX )
                    stopflag=1;
                if( std::abs(xold-x)<DELTAMAX )
                    converged=1;
            }
        }
    }
    return 0;
}
```
temp = 10.0 - 3.0 * yold*yold;
if(temp < 0.0)
  {
    printf("\nTry another guess value!");
    exit(0);
  }

y= pow(temp,0.2);
printf("\n Iteration #%3d, y = %22.16g",iter,y);
eps = fabs(y-yold);
delta= fabs( (y*y*y+3.0)*y*y - 10.0);
if( eps < EPSMAX || delta< DELTAMAX ) converged=1;
if(converged || iter > MAXITER) stopflag=1;
yold=y;
iter++;
}

/* print final results */
if(converged)
  {
    printf("\n\n Root = %22.16g",y);
    printf("\n\n Number of iterations = %d, MAXITER = %d",
            iter,MAXITER);
    printf("\n eps = %g,   EPSMAX = %g",eps,EPSMAX);
    printf("\n delta = %g, DELTAMAX = %g",delta,DELTAMAX);
  }
else
  {
    printf("\nFailed to converge in %d iterations",MAXITER);
  }

16.13 A Guessing Game

We will develop a program which plays a simple guessing game. The user thinks of a number between 1 and 100. The computer gives a guess value. Then the user has to tell the computer whether the guess is lower than the number in the user's mind or is it greater than the number? This is repeated until the correct value is guessed by the computer.

This works on the binary search algorithm for searching in an ordered list. The first guess value is 50. If the user says that this is lower than the unknown number, then the unknown number must lie between 51 and 100. If the user says that this is higher than the unknown number, then the unknown number must lie between 1 and 49. Let say that the user enters 'h' for high.

We now divide the interval of 1 to 50 into two smaller parts and print a guess value of 25. If the user says that this is lower than the unknown number, then the unknown number must lie between 26 and 50. If the user says that this is higher than the unknown number, then the unknown number must lie between 1 and 24. By repeating this procedure, we make our halve our interval of uncertainty until it becomes small enough to fit only one number, the unknown number.

The binary search algorithm works on any set of data as long as we know that they have been arranged in some order.
Program 16.13 – GUESS.C

**Program to play guessing game using a bisection search algorithm.**
You think of a number between 1 and 100
The computer keeps guessing and each time you respond by stating whether the guess value
is lower than your number
is higher than your number or
is the right number

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

void main(void)
{
    int n, guess=50, low=1, high=101;
    char ans;
    printf("Guess is lower than your value  -> l\n");
    printf("Guess is higher than your value -> h\n");
    printf("Guess is correct           -> r\n");
    printf("Please think of a number between 1 and 100\n");
    for(n=0;n<20;n++)
    {
        printf("Is it %d ? \n",guess);
        printf("Enter (low, high or right) l, h, or r >>",guess);
        ans = getchar(); /* read first character */
        while(getchar() != '
') /* skip the remaining characters in input */ /* validate user input */
            { if(ans=='l' || ans=='h' || ans=='r')
                { printf("\n"| illegal input, input l, h, or r >>");
                    ans = getchar(); /* read first character */
                    while(getchar() != '
') /* skip the remaining characters in input */
                        { if(ans=='r') break;
                            else if(ans=='l') low = guess;
                            else if(ans=='h') high = guess;
                            guess = (low+high)/2;
                        }
            }
            else if(ans=='r') printf("\nI am so smart!!\n");
            else printf("\nYou must be lying !!\n");
    }
}
```

### 16.14 Power Series Computation of sine(x)

**Problem Statement:** A power series expansion for sine(x) can be used to calculate sine(x) for any angle x given in radians.

\[
sine(x) = x - x^3/3! + x^5/5! - x^7/7! + \ldots + (-1)^k x^{2k-1}/(2k-1)! + \ldots
\]

Write a program to calculate sin(x) for any given x using this power series.

We will call this program SINECALC.C. How many terms do we need to achieve an accurate value for the value of sine(x)? We will stop if the value of any term becomes smaller than some small value
ε. Just to be on the safe side, we will also set a maximum number of terms to be used for the calculation.

Let us see how to calculate the k-th term. We see that to find the k-th term, we need to compute
1. the value of \((-1)^k\)
2. the value of \(x^{2k-1}\) and
3. the value of the factorial of \((2k-1)\)

We can use the power function, \texttt{pow}, from the math library for the first two operations and write a loop to calculate the factorial of \((2k-1)\). But that will lead to a very slow and inefficient program. Is there another method to find the k-th term? Let us examine and see whether there is any relation between the term \((k-1)\) and the k-th term. We can write

\[
\begin{align*}
(-1)^k &= (-1)(-1)^{k-1} \\
x^{2k-1} &= x^2 \cdot x^{2k-3} \quad \text{and} \\
(2k-1)! &= (2k-1)(2k-2)(2k-3)!
\end{align*}
\]

We see that we can write a recursive definition of the \(k+1\) term, \(T_{k+1}\), using the k-th term, \(T_k\).

\[
T_k = T_{k-1} \cdot (-1) \cdot \frac{x^2}{[(2k-1)(2k-2)]}
\]

This simplifies the calculation by reducing the number of multiplication operations and factorial calculations.

At the end of the program, we compare the value obtained by our program against the value returned by the standard math library function, \texttt{sin}.

\textbf{Program 16.14 – SINECALC.C  Power series computation of the sine function}

```c
#include <stdio.h>
#include <math.h>
#define EPS  1.0E-2
#define MAXTERMS 40

void main()
{
  int nterm,sign,converged;
  double ang, angsqr, term, sum, xpow, fact;
  printf("Enter an angle in radians : ");
  scanf("%lg",&ang);
  angsqr= ang*ang; /* square of ang */
  sum=ang;
  term=ang; /* first term in sequence */
  nterm=1;
  sign=1;
```
xpow=ang;
fact=1.0;
converged=0;
printf("%2d %16.7lg %16.7lg",1,ang,ang);
while(!converged && nterm < MAXTERMS)
{
    nterm++;
    sign *= -1;
    xpow *= angsqr;
    fact *= (2^nterm-2) * (2^nterm-1);
    term = sign * xpow/fact;
    sum += term;
    printf("%2d %16.7lg %16.7lg",nterm,term,sum);
    if (fabs(term) < EPS) converged=1;
}
printf("\nCalculated value of sin(%lg) = %lg","\nValue from library function = %lg","\n}

16.15 Calculation of Equal Instalment Payments
Let us look at an interesting financial problem. You take a loan of Rs.10,000/- from a
bank at an annual interest rate of 15%. You agree to pay back the amount in 10 equal
instalments at the end of the 1st, 2nd, . . ., and 10th year. What should be the amount of
each instalment? the detailed calculations done using a pocket calculator are shown
in Table 16.1. The figure of Rs.3908.88 in the last cell indicates that our guessed
instalment was too low. We can repeat the calculation for a higher amount. By a trial
and error process, we can arrive at an correct instalment amount which will leave a
value of Rs.0.00 or a small negative value in the last cell.

<table>
<thead>
<tr>
<th>End of Year</th>
<th>Principal</th>
<th>Interest</th>
<th>Total amount due at end of year</th>
<th>Instalment Paid</th>
<th>Balance Due</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10000.00</td>
<td>1500.00</td>
<td>11500.00</td>
<td>1800.00</td>
<td>9700.00</td>
</tr>
<tr>
<td>2</td>
<td>9700.00</td>
<td>1455.00</td>
<td>11155.00</td>
<td>1800.00</td>
<td>9355.00</td>
</tr>
<tr>
<td>3</td>
<td>9355.00</td>
<td>1403.25</td>
<td>10758.25</td>
<td>1800.00</td>
<td>8958.25</td>
</tr>
<tr>
<td>4</td>
<td>8958.25</td>
<td>1343.74</td>
<td>10301.99</td>
<td>1800.00</td>
<td>8501.99</td>
</tr>
<tr>
<td>5</td>
<td>8501.99</td>
<td>1275.30</td>
<td>9777.29</td>
<td>1800.00</td>
<td>7977.29</td>
</tr>
<tr>
<td>6</td>
<td>7977.29</td>
<td>1196.59</td>
<td>9173.88</td>
<td>1800.00</td>
<td>7373.88</td>
</tr>
<tr>
<td>7</td>
<td>7373.88</td>
<td>1106.08</td>
<td>8479.96</td>
<td>1800.00</td>
<td>6679.96</td>
</tr>
<tr>
<td>8</td>
<td>6679.96</td>
<td>1001.99</td>
<td>7681.95</td>
<td>1800.00</td>
<td>5881.95</td>
</tr>
<tr>
<td>9</td>
<td>5881.95</td>
<td>882.29</td>
<td>6764.25</td>
<td>1800.00</td>
<td>4964.25</td>
</tr>
<tr>
<td>10</td>
<td>4964.25</td>
<td>744.64</td>
<td>5708.88</td>
<td>1800.00</td>
<td>3908.88</td>
</tr>
</tbody>
</table>
16.16 Computation of GCD of Two Integers

** LCM calc using GCD

16.17 Calculation of Perfect Numbers

** perfect numbers – sum of factors equals number e.g., 6 = 1+2+3

16.18 Calculation of Pythagorean Triplets

integers a,b,c, a<b<c and a^2+b^2=c^2

16.19 Creating Fractional Approximations for PI

** fractional approximations to pi using loop.

16.20 Points to Remember

Review Quiz

Programming Exercises

**EXERCISE**
1. Rewrite this program using a while loop instead of the for loop.

**EXERCISE**
1. Write a C program to implement the first algorithm outlined above.
2. Modify the program to find 6 digit integers with similar properties.

**modify triangle program to classify the triangle as acute, right or obtuse**

\[ c^2 > a^2 + b^2 \]
if acute, is it equilateral, isosceles or scalene
**EXERCISE**
1. Rewrite this program using a **while** loop instead of the **for** loop.
2. Rewrite this program using a **do-while** loop instead of the **for** loop.

**EXERCISE**
1. Run the program for different guess values and observe the convergence behavior of the program for each value.
2. Study the details of this iterative algorithm from any standard textbook on Numerical Analysis. Under what conditions does it converge?
3. Given the equation \( x^5 + 3x^2 - 10 = 0 \), there is another way to write it in the form \( x = f(x) \).

\[
    x = \sqrt[5]{10 - \frac{x^5}{3}}
\]

Rewrite the program using the above formula and see what happens to the convergence.

**EXERCISE**
1. Rewrite the program using a **do-while** loop instead of a **while** loop. How do the two programs handle an input value of 0? Is there any difference?

**EXERCISE**
1. Study the details of the Newton-Raphson formula from any standard textbook on Numerical Analysis. Under what conditions does it converge?
2. Develop the Newton-Raphson formula to calculate the cube root of a number and modify this program to calculate the cube root of the given number.
3. Modify the program so that the user does not have to input the guess value for the square root, i.e., the program should generate its own guess value. Truncated power series expansions provide a good source for approximations. We can write \( x^{1/2} = (1 + (x-1))^{1/2} = 1 + (x-1)/2 + \ldots \) which when truncated at the second term yields \((1+x)/2\) which can be used as the guess value for the root.

**EXERCISE**
1. Run the program to see what happens when a large value of \( n \) is used as input.

**EXERCISE**
1. Write a program to calculate \( e^x \), \( \log(x) \), and \( \arcsin(x) \) using power series expansions.

**EXERCISE**
1. Write a program which, given an integer \( n \) as input, generates a pyramid of \( n \) lines using the last digit of the line number (starting from 1). The output for \( n=7 \) is shown below as an example.

```
1
22
3333
444444
55555555
6666666666
777777777777
```
2. Write a program which, given an integer \( n \) as input, generates a pyramid of \( n \) lines using the last digit of the line number (starting from 1). The output for \( n=5 \) is shown below as an example.

\[
\begin{align*}
1 \\
12 \\
123 \\
1234 \\
12345 \\
1234 \\
123 \\
12 \\
1
\end{align*}
\]

3. Write a program which, given an integer \( n \) as input, generates a pyramid of \( n \) lines. The output for \( n=7 \) is shown below as an example.

\[
\begin{align*}
1 \\
212 \\
32123 \\
4321234 \\
543212345 \\
65432123456 \\
7654321234567
\end{align*}
\]

4. Write a program which, given an integer \( n \) as input, generates a pyramid of \( n \) lines. The output for \( n=1, 2, 3 \) and \( 5 \) are shown below as examples.

\[
\begin{array}{cccc}
\text{n=1} & \text{n=2} & \text{n=3} & \text{n=5} \\
1 & 2 & 3 & 5 \\
222 & 303 & 505 & \\
3333 & 50005 & \\
& 55555555 & \\
\end{array}
\]

5. Write a program which, given an integer \( n \) as input, generates a pyramid of \( n \) lines using characters. The output for \( n=2 \) and \( 4 \) are shown below as examples.

\[
\begin{array}{cc}
\text{n=2} & \text{n=4} \\
\text{a} & \text{a} \\
\text{bc} & \text{bc} \\
\text{cde} & \text{defg} \\
\end{array}
\]

Find numbers between 1 and 999 for which the sum of the cubes of the digits equals the number, 371, 563 is not.

** For any given value of \( r \), the equation \( x^2 + y^2 = r^2 \) represents a circle. A point \((a,b)\) lies inside the circle if \( a^2 + b^2 < r^2 \).

For any given \( r \), list all the points with integer coordinates which lie inside the circle.
17.0 Lesson Goals

17.1 Files and Streams
The basic entity for storage of data on a computer is the file. A file may contain any kind of data. In this chapter, we will look at the basic operations involved in performing input/output from files.

The files reside on some storage device like a hard disk, floppy drive, compact disk, magnetic tape, etc. The format of the files as well as the use of files is controlled by the operating system of the computer. Our C programs need to have some means of communicating with the data files residing on the storage devices. This communication is performed using a file stream (or data stream). We can visualize the stream as some kind of an information pipeline through which data can flow between the program and the file.

As shown in Figure 17.1, we need one to establish a different stream for every data file that we want to use. In general, we can open as many streams as we need but on many implementations of C, there exists an upper limit on the maximum number of streams that may be open at any one time.

To create a file stream, we use a new kind of variable, the FILE variable. We can declare streams variables of type FILE *.
FILE *stream1, *stream2;

where \texttt{stream1} and \texttt{stream2} are the names of the two streams that we want to define. The stream name must be a valid name according to the rules of naming C variables.

The data type \texttt{FILE} is a \texttt{typedef} for a \texttt{struct} defined in \texttt{stdio.h} and the \* indicates a pointer to a variable of type \texttt{FILE}. Hence, stream variables are often referred to as \textit{file pointers}. Since, we have not encountered \texttt{structs} and \texttt{typedefs}, we will postpone a detailed discussion of the data type \texttt{FILE} to a later chapter. For the time being, we will learn to declare streams as shown above.

\section*{17.2 Opening a File}

Continuing the analogy of the information pipeline, given a file and a stream, the first thing we need to do (before we can exchange any data) is to connect the pipeline to the file. This is accomplished using the \texttt{fopen} function available in the standard I/O library.

\begin{verbatim}
fileptr = fopen(filename, opening mode);
\end{verbatim}

where \texttt{fileptr} is the name of a stream variable of type \texttt{FILE *}, \texttt{filename} if a string containing a valid file name, and \texttt{opening mode} is a string ("w", "r", or "a")

The three opening modes are
\begin{itemize}
\item Read mode ("r") - This opens an existing file for input (reading data).
\item Write mode ("w") - This creates a new file for output (writing data). If an old file of the same name exists, it is erased.
\item Append mode ("a") - This opens an existing file for appending data to the end of the file. The previous contents remain unchanged and the new data is appended at the end of the previous data. If the file does not exist, a new file is created. Therefore, when a file does not exist, the "a" mode is exactly the same as the "w" mode.
\end{itemize}

What happens if the \texttt{fopen} fails to establish the stream between the file and the program due to some reason? The function returns a value of \texttt{NULL} (\texttt{NULL} is a predefined symbolic constant for a pointer given in \texttt{stdio.h}). Therefore, we can check for a \texttt{NULL} value of the stream to see if the \texttt{fopen} operation has failed and take some appropriate action for such an error. In this book, whenever a file opening fails, we will use the \texttt{exit} function from stdlib to exit from the program.

\begin{verbatim}
infile = fopen("SAMPLE.DAT","r");
if(infile == NULL)
{
    printf("\n\aERROR : File SAMPLE.DAT could not be opened");
    exit(1);
}
\end{verbatim}
The first two statements can be combined as shown below

```c
if((infile = fopen("SAMPLE.DAT","r")) == NULL)
{
    printf("\n\aERROR : File SAMPLE.DAT could not be opened");
    exit(EXIT_FAILURE);
}
```

EXIT_FAILURE is a predefined symbolic constant for the `exit` function (usually having a value of 1). It is absolutely essential to perform such error checking for every file that is opened as the possibilities of failure are very high in this operation. Some of the cases where the fopen function might fail are

- invalid file name specified,
- the file being opened in read mode does not exist at the specified location (the default subdirectory or any other location specified in the file name).
- the file is locked by the operating system (e.g., when some other user is using it on a multiuser system)
- the file being opened in write or append mode exists on a write protected media.

### 17.3 Input/Output from Files

Input and output from files is accomplished using a set of functions analogous to the input/output functions which we have seen in Chapters 9 and 10. They all contain an extra argument, fileptr, which must be the name of a stream variable connected to a data file. This implies that all these functions can only be used after the successful completion of a call to `fopen` function, i.e., after a file has been successfully opened.

- **fgetc(fileptr)** returns the next unsigned character input from the stream.

- **fputc (char c, fileptr)** writes the character `c` to the stream.

- **fgets(s,n,fileptr)** reads the next input line of text from the stream into the string `s`. It will read a maximum of `n-1` characters into the character array `s` and terminate it with `\0`. If a newline character, `\n`, is encountered, the input is stopped at the newline character and the newline becomes a part of the string.

- **fputs(s,fileptr)** writes the string `s` to the stream

- **fprintf(fileptr,const char * format, ...)** performs formatted output to the stream
**fscanf(fileptr, const char * format, . . .)**
performs formatted input from the stream.

When using the `fgetc` function, we can detect the end of the file using a special value defined as the symbolic constant `EOF` in `stdio.h`. The following program `TYPEIT.C` demonstrates the use of the EOF value in printing all the characters in a file to the standard output device. The `while` loop terminates when the EOF value is encountered for the value of `c`. Notice that we have used a data type `int` for `c`. This is the correct type of argument for the `getchar` and `fgetc` functions.

**Program 17.1 – TYPEIT.C  Echoing the contents of a file to screen**

```c
/*================================ TYPEIT.C ============================
  Program writes the contents of a file to the standard output.
  =====================================================================*/
#include <stdio.h>
#include <stdlib.h>
void main(void)
{
  FILE * infile;
  int c;
  char filename[50];
  printf("Enter file name to display >> ");
  gets(filename);
  if((infile = fopen(filename,"r"))==NULL)
    {
    printf("\aERROR : could not open file %s",filename);
    exit(EXIT_FAILURE);
    }
  while((c=fgetc(infile)) != EOF)putchar(c);
  fclose(infile);
}
```

### 17.4 Closing a File

Once we have finished all input/output operations related to a file, we should close the connection with the file using the `fclose` function.

```
fclose(fileptr);
```

where `fileptr` is the name of an open file stream. In most implementations, the streams are automatically closed when the program exits but it is good programming practice to use the `fclose` function for every open stream.

### 17.5 Predefined Streams

For every C program, there are three predefined stream names by default - `stdin`, `stdout`, `stderr` - which stand for standard input stream, standard output stream, and the standard error stream. We can use these as the stream names with any of the stream input/output functions given in Section 17.3. For example,
fgetc(stdin) is equivalent to getchar(),
fputs(s,stdout) is equivalent to puts(s), and
fprintf(stdout, . . .) is equivalent to printf( . . .)

17.6 Flushing of Stream Buffers
On most implementations of C, a buffer is used to store outgoing data. Because a write operation to a storage media is a slow operation, the buffer is used to store the output until a sufficient amount of data has accumulated. Once the buffer is full, the contents of the buffer are written to the output file. Due to this, we may not see the output immediately after it is written by the program. Forcing the output to be cleared from the buffer is known as flushing the buffer and is accomplished by the fflush function.

fflush(fileptr);

where fileptr is the name of an open output stream. To flush all output streams, we can use the NULL file pointer

fflush(NULL);

will flush all output buffers. A buffer is also used for input streams but the effect of fflush on an input stream is implementation dependent.

17.7 Error Handling for Files
We have already seen how to detect errors during file opening in Section 17.*. Two other functions are useful in detecting errors during file operations.

int feof(fileptr);

returns a non-zero value (TRUE) if the end of file is encountered for the stream fileptr.

int ferror(fileptr);

returns a non-zero value (TRUE) if an error is encountered for the stream fileptr.

Once we have handled the error in some appropriate manner, we need to clear the errors on the stream using the clearerr function.

clearerr(fileptr);

17.8 Mini Problem - Changing to Upper Case
The following program reads text from one file and puts into another file after converting every character to uppercase. The program uses two streams named infile and outfile for handling the input and output of data.

Program 17.2 – UPCASE.C
Converting a text file to uppercase
```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

void main(void)
{
    FILE *infile, *outfile;
    int c;
    char filename[50];
    printf("Program to convert contents of file to UPPERCASE\n
Enter input file name >> ");
    gets(filename);
    if((infile = fopen(filename,"r")) == NULL) {
        printf("\aERROR : could not open file %s",filename);
        exit(EXIT_FAILURE);
    }
    printf("Enter output file name >> ");
    gets(filename);
    if((outfile = fopen(filename,"w")) == NULL) {
        printf("\aERROR : could not open file %s",filename);
        exit(EXIT_FAILURE);
    }
    while((c = fgetc(infile)) != EOF) fputc(toupper(c), outfile);
    fclose(infile);
    fclose(outfile);
}
```

17.9 Mini Problem - Separating Odd and Even Integers

The program ODDEVEN.C given below demonstrates the use of one input stream and two output streams. The program reads a data file containing integers and separates them into a file containing odd integers and another containing even integers. Notice the use of the `feof` function to detect the end-of-file condition.

Program 17.3 – ODDEVEN.C Separating odd and even integers from a file

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

void main(void)
{
    FILE *infile, *oddfile, *evenfile;
    int n;
    if((infile = fopen("input.dat","r")) == NULL) {
        printf("\n \aERROR : could not open file input.dat\n");
        exit(0);
```
17.10 Mini Problem - Counting Characters, Words & Lines

The following program WRDCOUNT.C counts the number of characters, words, and lines in a given file. For this purpose, a word is a contiguous set of graphic (printable) characters. We detect a word when the present character is a graphic character and the previous character is not a graphic character. We use the isgraph function from ctype library for this purpose. A line is defined as a set of characters ending with a newline character (\n). The results are stored in a file named WRDCOUNT.OUT. Note that this file is opened in append mode ("a") which allows us to accumulate the count statistics of various input files in a single file named WRDCOUNT.OUT.

Program 17.4 – WRDCOUNT.C Counting characters, words, and lines in a text file

```c
/*===================== WRDCOUNT.C ==================
 Program to count characters, words & lines in a file
===================================================*/
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

void main(void)
{
    FILE *infile, *outfile;
    char filenam[50];
    char first,second;
    long int numlines=0,numwords=0,numchars=0;
    printf("Enter name of input file >> ");
    scanf("%s",filenam);
    if((infile = fopen(filenam,"r")) == NULL)
    {
        printf("ERROR: Failed to open file %s",filenam);
        exit(1);
    }
    outfile = fopen("wrdcount.out","a");
    first=fgetc(infile);
    while(!feof(infile))
    {
        fscanf(infile,"%d",&n);
        if(!feof(infile))
        {
            printf("  n = %d
",n);
            if(n%2) fprintf(oddfile, "%d
",n);
            else    fprintf(evenfile,"%d
",n);
        }
    }
```
if(first!=EOF)
{
    numchars=1;
    if(first=='\n')numlines++;
    if(isgraph(first))numwords++;
    while((second=fgetc(infile)) != EOF)
    {
        numchars++;
        if(!isgraph(first) && isgraph(second))numwords++;
        if(second=='\n')numlines++;
        first=second;
    }
    fprintf(outfile,\"\n\n Counting results for %s\",filenam);
    fprintf(outfile,\"\n Number of lines = %ld\",numlines);
    fprintf(outfile,\"\n Number of words = %ld\",numwords);
    fprintf(outfile,\"\n Number of characters = %ld\",numchars);
    printf(\"\nYour output is in wrdcount.out\");
    fclose(infile);
    fclose(outfile);
}

17.11 Mini Problem – Shift Encoding of Strings

In this section, we will look at programs to encode (ENCODE.C) and decode (DECODE.C) a text file using a simple letter substitution code.

17.11 Mini Problem - Encoding and Decoding a Text File

In this section, we will look at programs to encode (ENCODE.C) and decode (DECODE.C) a text file using a simple letter substitution code. In a simple letter substitution code, every letter is replaced by another letter. This makes it impossible to decipher the text in the file unless we have the letter substitution code. In our code, 'a' is replaced by 'i', 'b' is replaced by 'j', and so on. This is specified in the character array code. We also replace all space characters with the ASCII character 1.

We need to define two streams for each of the programs, one for the input text file and the other for the output text file.

For decoding, we need to create a decoding string which should do the reverse mapping of the characters, i.e., 'i' should be replaced by 'a', 'j' should be replaced by 'b', and so on. The following statement in DECODE.C finds the decode character.

\[ decode[code[j]-'a'] = j + 'a'; \]

We see that an integer expression is used as the array index for decode. Let us study this statement for j equals 0. code[0] equals 'i'. 'i'-'a' equals 8 and it gives the position of the letter 'i' in the array decode. Therefore, decode[8] is assigned a value of 'a', i.e., 'i' will be decoded as 'a' which happens to be the correct substitution.
Program 1.2 – H

/*========================= ENCODE.C ===============================
Program to encode a text file using a letter substitution code.
==================================================================*/
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#define SPACECODE 1

void main(void)
{
    char code[]="ijlhkaopnqzbdymwcveuftgsxr";
    char cinp,cout;
    int one;
    char infile[40],outfile[40];
    FILE *inp1,*outp1;
    printf("Enter name of file to encode >> ");
    gets(infile);
    printf("Enter name of coded file     >> ");
    gets(outfile);
    if( (inp1=fopen(infile,"r"))==NULL)
        {printf("ERROR: could not open %s",infile);
        exit(1);}
    outp1=fopen(outfile,"w");
    while( (one=fgetc(inp1)) !=EOF)
        {
            cout = cinp = one;
            if(isalpha(cinp))
                {
                    if(islower(cinp))
                        cout = code[cinp-'a'];
                    else
                        cout = toupper(code[cinp-'A']);
                }
            if(cinp == ' ')
                cout=SPACECODE;
            fputc(cout,outp1);
        }
    fclose(inp1);
    fclose(outp1);
}
decode.c as exercise.

Program 1.2 – H

/*========================= DECODE.C ===============================
Program to decode a text file using a letter substitution code.
==================================================================*/
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#define SPACECODE 1

void main(void)
{
    char code[]="ijklhkaopqbdymwcvuftsxr";
    char cinp,cout,decode[27];
    int one;
    char infile[40],outfile[40];
    FILE *inp1,*outp1;

    printf("Enter name of file to decode >> ");
    gets(infile);
    printf("Enter name of decoded file   >> ");
    gets(outfile);

    /* creating decoding string from code*/
    for(j=0;j<26;j++)
    {
        decode[code[j]-'a'] = j + 'a';
    }

    if( (inp1=fopen(infile,"r"))==NULL)
    {
        printf("ERROR: could not open %s",infile);
        exit(1);
    }
    outp1=fopen(outfile,"w");

    while( (one=fgetc(inp1)) !=EOF)
    {
        cinp = one;
        cout = cinp;
        if(isalpha(cinp))
        {
            if(islower(cinp))
                cout = decode[cinp-'a'];
            else
                cout = toupper(decode[cinp-'A']);
        }
        if(cinp== SPACECODE)cout=' ';
        fputc(cout,outp1);
    }
    fclose(inp1);
    fclose(outp1);
}

17.12 Points to Remember
Programming Exercises

**EXERCISE**
1. Modify the above program to count (a) lowercase characters, (b) uppercase characters, (c) digits, (d) punctuation marks, and (e) whitespace characters in a file.
2. Write a program to take words from a file and put them in another file writing them one word per line as shown below.
   *Input file contents:*
   
   what a brilliant program?
   
   *Output file contents:*
   
   what
   
   a
   
   brilliant
   
   program?

**EXERCISE ODDEVEN**
1. The fscanf function returns an EOF value if an end of file is encountered during read. Modify the above program to use this feature of fscanf function to detect the end of file.
2. Write a program to find the sum of all integers given in a file.

**EXERCISE**
1. These programs encode and decode only the letters of the alphabet and the space character. Modify the above program to encode and decode all the digits (0 through 9), all the punctuation marks (!;{[, etc.) and the newline character (\n) also.
2. Notice that most of the code is common to both the programs. Create a single program which can do both the encoding and the decoding based on some user input.
18.0 Lesson Goals

18.1 What is a Function?

We have briefly looked at functions in Chapters 1 and 10. In this chapter, we will learn more about functions and we will also learn how to develop our own functions.

Functions are an essential ingredient of structured programming. Given any major problem, we can write the solution in several steps. The steps themselves may be very simple. Each of these steps can be implemented as a function.

Whenever we have a simple task that is repeated very often, we develop a function to do this task. This function can then be reused in any program. Such reusability is the hallmark of good software. In other words, functions allow us to avoid reinventing the wheel every time we need a wheel by allowing us to use a wheel created by someone else. For example, the square root function is used very often in mathematical calculations. Therefore, a function named $\text{sqrt}$ has been developed for this task. Further, since the square root function is a universally accepted one, it has been made a part of the standard function library in C. There might be other tasks that you perform repeatedly in your programs for which there is no function in the standard library. In such cases, you can develop your own functions.

Some of the features of well designed functions are listed below:

- A function must do only one small thing but do it very efficiently.
- A function must be small. If the printout (hardcopy) of the function exceeds one or two pages, then it too long.
- Every function must hide some details.

A function must not try to do many things and it must be small. This is in line with our earlier idea that the function is the simplest possible task which cannot be further subdivided. If a function is too long, it is possible that there exists a way to further subdivide the task into smaller tasks.
The function must hide details of the algorithm from the user. For example, when we use the \texttt{sqrt} function from the math library, we need not know the algorithm used to calculate the square root. All we need to know is that given an argument of type \texttt{double}, the square root of this value is obtained.

18.2 Functions in C

Every function receives some data, performs some tasks using the data and sends out some data. The data is in the form of variables which may be inputs to the functions as shown in Figure 18.1. Some variables go in as input and after being modified in some way, come out as output. In addition, in the C language, there can be one value returned by the function.

A function is defined in the following manner in C:

\begin{verbatim}
return-type function-name(list of formal parameters)
{
    function code
    return return-value;
}
\end{verbatim}

Every function must have a name which must a valid identifier according to the rules of the C language.

Every function can return one or zero values as the end result of calling the function.

- When one value is returned, the data type of the returned value must be specified before the name of the function.
- When the function does not return any value, the returned data type is specified as \texttt{void}. 

\begin{figure}
\centering
\includegraphics[width=\textwidth]{function-diagram.png}
\caption{Variables in a C Function}
\end{figure}
• If no return type is specified before a function name, the returned data type is assumed to be int by default.

All the input variables, output variables, and input/output variables are listed in the parameter list. The parameter list has the following form

\[(\text{type1 \ var1, type2 \ var2, \ldots, type-n \ var-n})\]

Each formal parameter is a variable and it is preceded by its variable type. For example,

\[
\text{widget(int n, float x, char p)}
\]

defines a function named \text{widget} with a return value of type int (since no return type is specified) having three formal parameters - the first is an int variable, \text{n}, the second is a float variable, \text{x}, and the last is a char variable, \text{p}. These formal parameters can be used as ordinary variables in the function code as shown below.

\[
\text{widget(int n, float x, char p)}
\{
\text{printf("\n \ n = \%d, x= \%f, p = \%c", n,x,p);}
\text{n++;}
\text{printf("\n New value of n = \%d",n);}
\}
\]

In addition to the formal parameters, we can declare new variables inside the function. These functions are available only inside the function, i.e., they are local to the function. If an int variable \text{x} is declared inside the function \text{foobar} and a float variable \text{x} is declared in the main function, there is no conflict of names because the variables of the function are stored in a separate location in the memory and they are visible only inside the function.

It is possible for a function to have no formal parameters. The list of formal parameters is followed by the actual code of the function.

Using these rules, we see that the following is the minimal function definition in C.

\[
\text{void dummy()}
\{
\}
\]

This is a function with no formal parameters which does nothing and returns nothing. In Chapter 36, we will learn to use such dummy functions in developing stubs for large programs.

The return statement plays an important role in a function. The function ends when a return statement is encountered. If the return statement is followed by an expression, the value of the expression is converted to the return-type and used as the return value. If no return statement is found in a function, the function terminates at the last statement in the function and the return value is undefined.
18.3 Calling a Function

A function can be called by any other program by name. In addition, the calling function must supply some arguments. These arguments must have a one-to-one correspondence with the formal parameters both in type and their place in the list of arguments. If we have defined a function

\[
\text{foobar}\text{(int var1, float var2, char p)}
\]

Then, we can call this function as

\[
\text{foobar}\text{(arg1, arg2, arg3)};
\]

where \text{arg1} must be a constant or a variable of type \text{int}, \text{arg2} must be a constant of variable of type \text{float} and \text{arg3} must be a constant or variable of type \text{char}. The number of arguments supplied in a call to a function must be the same as the number of formal parameters in the function definition and each argument must match its corresponding formal parameter in type.

The value returned by a function can be used in place of any expression of return-type. For example, the return value from \text{foobar} (defined in Section 18.2) can be used wherever an integer expression is required. For example,

\[
\text{if(foobar}\text{(4, 3.14159, 'Y')) printf("Ok");}
\]

Extending this further, we can use a function as the argument for another function as in

\[
\text{sqrt(sin(arctan}(1.0))\text{))}
\]

18.4 Developing Simple Functions

In this section, we will look at some simple function definitions and study how they work.

\[
\text{void sayhello(void)}
\]

\{
   \text{printf("Hello!\n");}
   \text{return;}
\}

function name: \text{sayhello}

return-type: \text{void} (no value is returned)

formal parameters: \text{none} (indicated by \text{void})

This function uses the printf function to print a simple message to standard output.

\[
\text{n_abs(int n)}
\]

\{
   \text{if(n < 0)}
   \text{return (-n);}
   \text{else}
   \text{return n;}
\}
function name: n_abs
return-type: int (default type when no return type is specified)
formal parameters: int n (one parameter)
This function returns the absolute value of the integer n.

float reciproc(float x)
{
    float y;
    y = 1.0/x;
    return y;
}

function name: reciproc
return-type: float
formal parameters: float x (one parameter)
This function returns the reciprocal of x. Notice that we can declare new variables inside a function as we have done for y in this function. We do not have to declare the formal parameters again as they behave like predeclared variables for the function. This variable y will be invisible to all other functions, i.e., it is local to the function reciproc.

char get_yesno(void)
{
    char ans;
    int valid;
    do
    {
        printf("Please enter Y for yes or N for no >> ");
        ans=getchar();
        /* change to upper case*/
        if(ans > 96) ans -= 32;
        /* skip remaining characters in input buffer */
        while(getchar()!='\n');
        valid = ans=='Y' || ans=='N';
        if(!valid)
            printf("ERROR: Illegal input, please try again!");
    }
    while(!valid);
    return ans;
}

function name: get_yesno
return-type: char
formal parameters: none (use of void)
This function returns the letter 'Y' or 'N'. We declare two new variables inside the function, valid and ans.

18.5 Function Prototypes
How does the compiler know that a function is being called with an incorrect number of arguments? For example, if we have defined

```c
foobar(int var1, float var2, char p)
{ some code
}
```

and the main function calls it in the statement

```c
p = foobar(4, 6.78, 8.96);
```

We have an invalid call because the third argument does not of type `char`. But how can the compiler detect this error? To ensure type matching between the arguments in the calling function, we use a **function prototype declaration** (sometimes known simply as the function declaration). It takes the form of the statement

```c
return-type function-name(type1 dummy1, type2 dummy2, ... ,
                          type-n dummy-n);
```

placed in the calling function or at the start of the file containing the calling program. `dummy1`, `dummy2`, etc, are dummy names for the formal parameters which are optional. Even when they are present, they are ignored by the compiler. An alternate form without dummy parameters is

```c
return-type function-name(type1 , type2, ... ,type-n);
```

Notice the semi-colon at the end of the function prototype declaration. A function with no parameters is declared as

```c
return-type function-name(void);
```

and not as

```c
return-type function-name(); /* INCORRECT ! */
```

An empty list `()` signifies an unspecified number of formal parameters in the function. (This is one of the weird legacies of ANSI C version 1.0!).

For the functions developed in the last section, the following are valid prototype declarations.

```c
void sayhello(void);
int n_abs(int n);
```

or

```c
int n_abs(int);
float reciproc(float x);
```

or

```c
float reciproc(float);
char get_yesno(void);
```

Notice that every prototype declaration ends with a semi-colon. The easiest way to put a declaration is to simply copy the header from the function definition, paste it and add a terminating semi-colon.
Prototype declarations are optional in C but they are compulsory in C++. Since, it is a reasonable assumption that every C programmer is a future C++ programmer, get into the habit of declaring prototypes for every function!

18.6 Call by Value

When a function is called with some arguments, a copy of the values of the arguments is made available to the function. For example, if we have defined a function

```c
foobar(int var1, float var2, char var3)
```

and it is being called as

```c
foobar(arg1, arg2, arg3);
```

Then, `var1` is assigned the value of `arg1`, `var2` is assigned the value of `arg2` and `var3` is assigned the value of `arg3`. The function `foobar` can make changes in the values of `var1`, `var2`, and `var3` but it cannot make any changes to the values of `arg1`, `arg2`, and `arg3`. This process wherein a copy of the value is passed but the variable itself remains protected is known as call by value. In call by value, the function cannot make any change in the arguments being used by the calling function.

The following program, CALBYVAL.C, demonstrates this feature of call by value. When this program is executed, the following things happen.

- the function `foobar` is called with `n` as the argument. `n` has a value of 6.
- this value is copied into `x`. Therefore, `x` equals 6 at the beginning of `foobar`.
- the value of `x` is changed to 100
- this has no effect on the value of `n` in main which remains unchanged at 6.

Program 18.1 – CALBYVAL.C Demo of call by value mechanism

```c
/* ============================= CALBYVAL.C ============================
 * This program demonstrates the features of function call-by-value.
 *====================================================================*/
#include <stdio.h>
/* function declaration for foobar */
int foobar(int );

void main(void)
{
    int n=6;
    printf("\n Original value of n = %d",n);
    foobar(n);
    printf("\nValue of n after call to foobar = %d",n);
}
/* function definition for foobar */
int foobar(int x)
{
    printf("\n Input value of x = %d",x);
```
18.7 Mini Problem - Developing Character Type Functions

In this section, we will develop a set of functions for testing character types. We will call these functions \texttt{n\_islower}, \texttt{n\_isupper}, etc., to avoid any possible conflicts with the library function \texttt{islower}, \texttt{isupper}, etc., in \texttt{ctype} library. We check for the appropriate ASCII codes to create the functions, \texttt{n\_islower}, \texttt{n\_isupper}, and \texttt{n\_isdigit}. Then, we write the function, \texttt{n\_isalpha}, using the two functions, \texttt{n\_islower} and \texttt{n\_isupper}. Finally, we write the function, \texttt{n\_isalnum}, using the functions, \texttt{n\_isalpha} and \texttt{n\_isdigit}. Note the prototype declarations for all these functions at the beginning of the program.

Program 18.2 – CHARTYPE.C  User defined character type functions

```c
/* Program containing user-defined character type functions. */
#include <stdio.h>
define TRUE 1
define FALSE 0
int n_islower(char); int n_isupper(char);
int n_isalpha(char);
int n_isdigit(char);
int n_isalnum(char);
void main()
{
char p;
printf("Enter a character >> ");
p = getchar();
while(getchar() != 
if(n_islower(p)!= TRUE); 
if(n_islower(p)!= TRUE); 
if(n_isupper(p)!= TRUE); 
if(n_isdigit(p)!= TRUE); 
if(n_isalnum(p)!= TRUE);
}
int n_islower(char c)
{
if(c >= 97 && c <= 122)
return TRUE;
else
return FALSE;
}
int n_isupper(char c)
{
if(c>=65 && c <= 90)
return TRUE;
else
```
Functions

18.8 Old Style Functions

While referring to some of the older books on C, you might encounter functions defined in the following manner

```
return-type function-name (var1, var2,...,var-n)
    type1 var1;
    type2 var2;
    type-n var-n;
{function code
}
```

For example,

```
/* old style defintion - DO NOT USE THIS STYLE!! */
int foobar (x,n,c)
    float x;
    char n,c;
```

This style of function definition where the type declarations are shown outside the list of formal parameters is known as **old-style or K&R style** function definition. This was allowed in ANSI C 1.0 and continues to be supported by more recent C compilers. This style should not be used in defining your functions and if you encounter any code using old style definitions, convert them to the new style definitions.

18.9 Using Non-ANSI Library Functions

Every commercial C compiler comes with a number of specialized function libraries specially developed for the particular implementation. It is very tempting for a programmer to make heavy use of these functions to simplify her work. But remember, this will make your program difficult to port to another implementation of C. Therefore, it is best to restrict oneself to the standard ANSI C libraries.
However, if one cannot do without some implementation dependent function, it is best to **encapsulate** this function call inside a well-documented user defined function. Shown below is the use of a function `clrscr()` available on Borland's C compiler which we have encapsulated in a function `clear_screen()`.

Program 1.2 – H A

```c
void clear_screen()
{
    /* clrscr clears the screen and
     places the cursor at the top left corner.
     Works in Borland C only.
     #include <conio.h> is required.
    */
    clrscr();
}
```

18.10 Points to Remember

Review Quiz

Programming Exercises

- strcmpi
- double intpow(double, int)
- isprime(n)
- roundoff(value, precisions)
Storage Classes

19.0 Lesson Goals

19.1 Automatic and External Variables

All the variables we have used so far have been declared inside functions. These variables local to the function. For example, let us look at the function foobar defined below

```c
foobar(int n)
{
 float xx;
 . . .
}
```

The variable `xx` comes into existence only when the function foobar is called, i.e., the variable `xx` is allocated some memory location only when the function foobar is called. As soon as we exit from the function foobar, this memory area is lost and the variable `xx` does not exist any longer. Such variables are known as automatic variables (because it automatically comes into existence when a function is invoked) or local variables. By default a variable declared in any function is assumed to be an automatic variable but we can explicitly declare a variable to be of automatic storage class using the keyword `auto`.

```c
auto float xx;
```

This leads to the more complete way of declaring variables as

```c
storage-class type variable-name;
```

The scope or the visibility of an auto variable is limited to the function itself. It is useful to remember that the formal parameters of a function behave like automatic variables by default. It is also possible to declare variables outside of any function as shown below.

```c
long serial_num;
double pi_value;
main(void) {
```
Here the variables `serial_num` and `pi_value` are declared external to all functions. Such variables are known as external or global variables. These are visible to all the functions in the file. In the above example, both `main` and `foobar` can access and modify the values of `serial_num` and `pi_value`. Execute the following program to see this feature of external variables.

**Program 19.1 – EXTVARS.C  Demo of global scope of external variables**

```c
/* This program demonstrates the global scope of external variables. */
#include <stdio.h>
void foobar(void);

int num=1;
float xx=33.33;

void main(void)
{
    printf("\nOriginal values : %d %f",num,xx);
    num += 6;
    xx += 10.0;
    printf("\nModified values : %d %f",num,xx);
    foobar();
    printf("\nAfter call to foobar : %d %f",num,xx);
}

void foobar(void)
{
    num++;
    xx *= 2.0;
    return;
}
```

External variables are allocated memory at the beginning of the execution of the program and they retain their memory space until the end of the program execution.

It is obvious that external variables provide us with an easy and convenient method for information exchange between functions. So far, we had been confined to the use of information exchange using the arguments of the functions but now we can begin to use the global scope of external variables and minimize the number of arguments in functions. These two means of information exchange between functions is shown in Figure 19.1.
The use of external variables (global variables) is considered poor programming practice because it becomes very difficult to find bugs in the programs. Since every function has access to the external variables, it becomes difficult to pinpoint which function is misbehaving when the values of the external variables go haywire. Therefore, the use of external variables should be minimized and, where possible, avoided altogether.

19.2 Static Variables

In this section, we will look at another storage class for variables. Static variables declared inside a function have their scope limited to the function itself but they retain a fixed memory location until the end of the program execution. A static variable is assigned a memory location at the beginning of program execution. If an initial value is given, the initialization also takes place at the beginning of program execution. Unlike the auto variables, static variables are not erased when we exit from a function. Whenever the particular function is called, the function has access to the static variable.

To use the analogy of human memory, a function forgets the values of the auto variables when we exit from a function but the values of static variables are remembered between different calls to the function.

In the following program, we have a static variable first in function foobar initialized to a value of 1. Therefore, when foobar is called for the first time, the logical expression (first) evaluates to TRUE and the if block is executed. In this if block, the value of first is changed to

Figure 19.1 Information exchange through function arguments and external variables
0. In all future calls to `foobar`, the value of `first` remains as 0 and the `if` block is never executed again!

Program 19.2 – ONCEONLY.C  Use of a static variable in a function

```c
#include <stdio.h>

void main()
{
    void foobar(void); /* function prototype declaration */
    int j,n=4;
    for(j=0;j<n;j++) foobar();
}

void foobar(void)
{
    static int first=1;
    if(first)
    {
        first=0;
        printf("This is the first call to the function foobar\n");
    }
    printf("Inside function foobar\n");
}
```

The following program NEXTNUM.C demonstrates the use of a static variable to generate a sequence of integers. The `static` variable `next` is initialized to a value of 0. When the function `nextnum` is called for the first time, the value of `next` is incremented by 1 and a value of 1 is returned. When `nextnum` is called a second time, it will return a value of 2 and so on. We see that even when we exit from `nextnum`, the value of the variable `next` remains in the memory and is available for the next call to `nextnum`.

Program 19.3 – NEXTNUM.C  Using a static variable to generate a number sequence

```c
#include <stdio.h>

void main()
{
    int nextnum(void); /* function prototype declaration */
    int j;
    for(j=0;j<20;j++) printf("%d", nextnum());
}

int nextnum(void)
{
    static int next=0;
    return(++next);
}
19.3 Pseudorandom Numbers

The function \texttt{n_rand} given below generates a pseudorandom sequence of integers by the using a recursive formula

\[ \text{rand} = (\text{rand} \times \text{ia} + \text{ic}) \mod \text{im}; \]

This method is formally known as "linear congruential generation". Using this formula, we can generate integers in the range of 0 to \((\text{im} - 1)\). It works only for special values of \text{ia}, \text{ic}, and \text{im}. These special values ensure that a particular integer is not repeated until all \text{im} possible values have been exhausted in a cycle.

Notice the use of static variables to remember the values of the variables between different calls to the function \texttt{n_rand}. When the function \texttt{n_rand} is called for the first time, the value of \text{jr}and must be set to a seed value. If no seed value is given, we assume a seed value. If a negative value of \text{seed} is passed to the function, it acts as the new seed value and a new sequence of pseudorandom numbers is generated.

** full program for random numbers

Program 19.4 – RAND1.C A pseudorandom number generator

```c
unsigned long n_rand(long int seed)
{ /* returns a pseudorandom integer n_rand 
   where 0 <= n_rand < im 
   a negative value of seed indicates an initialization 
   a positive value of seed is ignored 
 */
    static unsigned long im=714025;
    static unsigned long ia=4096;
    static unsigned long ic=150889;
    static unsigned long jrand, firstime=1;
    /* when function is called for first time */
    if (seed < 0 || firstime)
    {
        if (seed >= 0) seed = -33331;
        firstime = 0;
        jrand = -seed;
        jrand = (jrand \times \text{ia} + \text{ic}) \mod \text{im};
    }
    jrand = (jrand \times \text{ia} + \text{ic}) \mod \text{im};
    return(jrand);
}
```

For more details on random numbers and for a criticism of the \texttt{rand} function of \texttt{stdlib}, refer to Press et al. 1992.

**EXERCISE**

1. Develop a function \texttt{d_rand} using the function \texttt{n_rand}. It should return a pseudorandom double type value \(x\) such that \(0.0 \leq x < 1.0\).
19.4 Points to Remember

The function \texttt{n_rand} given below generates a pseudorandom sequence of integers by the using a recursive formula

\[
\text{rand} = (\text{rand} \times \text{ia} + \text{ic}) \mod \text{im};
\]

This method is formally known as "linear congruential generation". Using this formula, we can generate integers in the range of 0 to \((\text{im}-1)\). It works only for special values of \(\text{ia}, \text{ic},\) and \(\text{im}\). These special values ensure that a particular integer is not repeated until all \(\text{im}\) possible values have been exhausted in a cycle.

Notice the use of static variables to remember the values of the variables between different calls to the function \texttt{n_rand}. When the function \texttt{n_rand} is called for the first time, the value of \texttt{jrand} must be set to a seed value. If no seed value is given, we assume a seed value. If a negative value of \texttt{seed} is passed to the function, it acts as the new seed value and a new sequence of pseudorandom numbers is generated.

\begin{verbatim}
unsigned long n_rand(long int seed)
{
    /* returns a pseudorandom integer n_rand
      where 0 \leq n_rand < im
      a negative value of seed indicates an initialization
      a positive value of seed is ignored */
    static unsigned long im=714025;
    static unsigned long ia=4096;
    static unsigned long ic=150889;
    static unsigned long jrand, firstime=1;
    /* when function if called for first time */
    if(firstime &\& seed \geq 0) seed= -33331;
    if(seed < 0 || firstime)
    {
        firstime = 0;
        jrand = -seed;
        jrand = (jrand*ia+ic)\mod im;
    }
    jrand = (jrand*ia+ic)\mod im;
    return(jrand);
}
\end{verbatim}

For more details on random numbers and for a criticism of the \texttt{rand} function of \texttt{stdlib}, refer to Press et al. 1992. The book also gives a list of alternate sets of values for the constants \(\text{ia}, \text{ic},\) and \(\text{im}\) some of which are reproduced below in Table 19.1.

\begin{table}[h]
\centering
\caption{Alternate values for the constants for pseudorandom numbers}
\end{table}
generating pseudorandom sequences

<table>
<thead>
<tr>
<th>im</th>
<th>ia</th>
<th>ic</th>
</tr>
</thead>
<tbody>
<tr>
<td>6075</td>
<td>106</td>
<td>1283</td>
</tr>
<tr>
<td>7875</td>
<td>211</td>
<td>1663</td>
</tr>
<tr>
<td>53125</td>
<td>171</td>
<td>11213</td>
</tr>
<tr>
<td>139968</td>
<td>205</td>
<td>29573</td>
</tr>
<tr>
<td>233280</td>
<td>9301</td>
<td>49297</td>
</tr>
</tbody>
</table>

Review Quiz

Programming Exercises
24

Recursion

20.0 Lesson Goals

20.1 What is Recursion?

We have seen that functions can call other functions. Can a function call itself? In other words, can a function `foobar` contain a call to `foobar`? In the C language, the answer is "Yes" and this feature is known as recursion. In what kind of problems do we need a recursive solution? When a problem can be subdivided into smaller tasks, one or more of which may be smaller versions of the problem itself, recursion is the natural solution to the problem. The problem is broken down into smaller and smaller problems until we reach a problem that is small enough to be solved.

In the next two sections, we will look at two classic examples of recursion. Later, we will see why these recursive solutions are not the best solutions to the problems.

20.2 Recursive Calculation of Factorials

This is the classic example of recursion cited in many textbooks. The factorial of n can be written in terms of the factorial of (n-1) as

\[ n! = n \times (n-1)! \]

However, this cannot be done for all values of n. For n equals 0, the factorial function is defined to have a value of 1. We can write

\[ n! = 1 \text{ if } n \text{ equals } 0 \]
\[ = n \times (n-1)!, \text{ otherwise.} \]

We see that
- we have defined the factorial function in terms of the factorial function,
- each successive computation of the factorial function has a smaller value of n, and
- the successive values of n lead to a value of 0 and we know the value of 0!. 
The following program FACTRIL2.C contains a function factorial which is called recursively until 0! is obtained.

Program 20.1 – FACTRIL2.C 
Recursive computation of factorial function

```c
/* ============================== FACTRIL2.C ============================
Recursive calculation of factorials.
====================================================================*/
#include <stdio.h>
long factorial(long num);
void main()
{
    long num,factval;
    printf("Enter a +ve integer :");
    scanf("%ld",&num);
    factval = factorial(num);
    printf("Factorial of %ld is %ld",num,factval);
}
long factorial(long num)
{
    if(num == 0)
        return(1);
    else
        return(num*factorial(num-1));
}
```

20.3 Printing an Integer in Reverse

Let us look at the problem of printing the digits of an integer in reverse. We need to print the last digit first. We can break up the problem into the following steps in pseudocode.

- **print_last_digit of n**
  - if the number n is zero, nothing needs to be printed
  - if the number n is not zero,
    - obtain last digit of number using modulus function as n%10
    - print_last_digit of n/10.

We see that
- we have defined the print_last_digit function using the print_last_digit function,
- each successive call to the function print_last_digit gets a smaller value of n, and
- the successive values of n lead to a value of 0 and for n=0, we need not print anything.

This is implemented in the following program, REVINT2.C, which contains a function revdigit which recursively calls itself.

Program 20.2 – REVINT2.C
Printing an integer in reverse using recursion

```c
/* ============================== REVINT2.C ====================
Printing the digits of an integer in reverse order
using a recursive function.
*/
long revdigit(long num);
long main()
{
    long num;
    printf("Enter a +ve integer :");
    scanf("%ld",&num);
    printf("%ld",revdigit(num));
}
long revdigit(long num)
{
    if(num == 0)
        return(0);
    else
        return(revdigit(num/10)+num%10*10);
}
```
```c
#include <stdio.h>
void revdigit(int);
void main()
{
    int num;
    printf("Enter a +ve integer : ");
    scanf("%d",&num);
    printf("The digits in reverse are ");
    revdigit(num);
}
void revdigit(int n)
{
    if(n == 0)
        return;
    else
    {
        printf("%1d",n%10);
        revdigit(n/10);
    }
}
```

20.4 Guidelines for Developing Recursive Functions

Looking at the two case studies reviewed in the last two sections, we can make the following generalizations about the design of a recursive function.

- Each successive call to a recursive function call handles a smaller problem in terms of data or complexity.
- The succession of recursive calls must terminate when some stopping criteria is met. Therefore, every recursive function must contain a branch to some non-recursive statements in order to terminate the recursion. In other words, every recursive function must contain some kind of a branching structure (an if statement or a switch statement) with one branch leading to a recursive call and at least one branch leading to termination.
- The successive calls to the recursive function must lead to the termination branch. Otherwise, we might end up with an infinitely recursive function which never stops executing.

Recursion provides an elegant solution to many problems but the design of recursive functions needs to be done with extreme care.

20.5 Memory Considerations for Recursion

There are various possible ways in which a computer can be made to handle a recursive function. One of the most common methods is to store multiple sets of data for the recursive function, each set corresponding to one call to the function. Let us look at a hypothetical recursive function `recfoo` with the following definition

```c
int recfoo(int num, int k, float val)
{
    int temp;
```
This function has five auto variables, namely, \texttt{num}, \texttt{k}, \texttt{val}, \texttt{temp}, and \texttt{newval}. When \texttt{recfoo} is called, memory is allocated for each of these five variables. When the next recursive call to \texttt{recfoo} is made, memory is allocated to the five variables for this version of \texttt{recfoo}. The code for each new recursive version of the function remains the same but each carries its own set of variables.

Let us see what happens when we calculate the factorial of 5 using our recursive program developed in Section 20.2. The function is called a total of six times with the arguments decreasing from 5 to 0. Each instance of the function has its own copy of the variable \texttt{num}.

\[
\begin{align*}
\text{factorial}(5) & \Rightarrow \text{num}=5 \\
\text{factorial}(4) & \Rightarrow \text{num}=4 \\
\text{factorial}(3) & \Rightarrow \text{num}=3 \\
\text{factorial}(2) & \Rightarrow \text{num}=2 \\
\text{factorial}(1) & \Rightarrow \text{num}=1 \\
\text{factorial}(0) & \Rightarrow \text{num}=0 \\
\end{align*}
\]

This means that we are using up memory for 6 variables of type \texttt{long}! For any \texttt{n}, we will use a minimum of $(n+1)$ long variables to store all the intermediate values of \texttt{n}. As compared to this, our loop version of factorial calculation from Section 16.6 used only two variables, an int variable \texttt{kount} and a long variable \texttt{num}, irrespective of the value of \texttt{num}! We see that the recursive calculation of the factorial is not efficient from the viewpoint of memory usage and such recursive functions should be avoided. A little study will show that the program REVINT2.C of Section 20.3 is also very wasteful with memory as it keeps multiple copies of the variable \texttt{n}.

### 20.6 Printing a String in Reverse

In this section, we look at a program, REVSTR.C, which takes an input string and prints it in reverse using a recursive function, \texttt{revstring}. Each successive call to the function handles input and output of one character. The remainder of the string is left for the next instance of the function to handle until we reach a newline character ‘\n’ signifying the end of the input string. Since, every character is stored only in one instance of the function, we do not waste any memory space in this program.

**Program 20.3 – REVSTR.C**

```c
/* ++++++++++++++++++++++++++++++++ REVSTR.C ++++++++++++++++++++ 
   Printing an input string in reverse using a recursive function. 
   (Based on Kernighan & Ritchie **) 
   +++++++++++++++++++++++++++++++++++++++++++++++++++++*/
#include <stdio.h>
void revstring(void);
void main()
{
```

printf("Enter a string >> ");
revstring();
}
void revstring(void)
{
    char c;
    c=getchar();
    if(c != '\n')
        revstring();
    putchar(c);
    return;
}

20.7 The Towers of Hanoi

The Towers of Hanoi (sometimes known as the Towers of Banaras) is another classic problem for which recursion provides an elegant solution. There are 64 golden disks placed on one of three diamond pins. Each disk is slightly smaller than the disk placed under it. The task is to move all disks from Pin 1 to Pin3 subject to the following rules: (a) Only one disk can be moved from one pin to another pin, (b) a disk must never be placed under another disk of smaller size.

![Figure 20.1 The Towers of Hanoi with 7 disks](image)

We look the problem of moving the bottom disk, disk number 64, from Pin 1 to Pin 3. To accomplish this, we must first move the remaining 63 disks to Pin 2 because the 64th disk is the largest. Next, we need to move the 63 disks from Pin 2 to Pin 3. These steps can be written as

- move 64 disks from Pin 1 to Pin 3 (using Pin 2)
- move 63 disks from Pin 1 to Pin 2 (using Pin 3)
- move disk from Pin 1 to Pin 3
- move 63 disks from Pin 2 to Pin 3 (using Pin 1)

We see that we have broken down the move operation into smaller move operations. This can obviously be implemented as a recursive function. When do we stop the recursion? We stop when the number of disks reduces to zero.

Program 20.4 – HANOI.C Solution to the Towers of Hanoi problem

/*
 * Recursive solution to the Towers of Hanoi problem.
 * (Adapted from Kruse et al.,***)
 */
#include <stdio.h>
#define NUMDISKS 4
void move(int, int, int, int);

void main(void)
{
    move(NUMDISKS, 1,3,2);
}
void move(int ndisks, int a, int b, int c)
{
    if(ndisks>0)
    {
        move(ndisks-1,a,c,b);
        printf("Move a disk from %d to %d",a,b);
        move(ndisks-1,c,b,a);
    }
    return;
}

20.8 Generating Permutations

20.9 Alternatives to Recursion

We have seen that most of the simple problems we have reviewed in this chapter, there are alternate solutions using branching and looping control structures which are more efficient in their usage of memory. It has been proved that recursion can be totally eliminated using other solution algorithms. But, in problems like the Towers of Hanoi, a recursive solution provides the most natural and elegant solution. Therefore, recursion must be used judiciously keeping in mind various factors like simplicity of solution and memory considerations.

20.10 Points to Remember

Review Quiz

Programming Exercises

EXERCISE
1. The Greatest Common Divisor of two positive integers can be calculated by the following recursive formula known as Euclid’s method.

\[
\text{GCD}(m,n) = \begin{cases} 
 \text{GCD}(n,m) & \text{if } n > m \\
 m, & \text{if } n = 0 \\
 \text{GCD}(n, m \mod n) & \text{otherwise}
\end{cases}
\]

Write a recursive function `gcd` with the following prototype to calculate the GCD of two positive integers.

```c
int gcd(int m, int n);
```

** finding n-th partial sum of the continued fraction

\[
\frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \ldots}}}
\]

\[s_1 = 1\]
\[s_2 = 1 + \frac{1}{1}
\]
\[s_3 = 1 + \frac{1}{(1 + \frac{1}{1})}
\]

Find a recursive formula for \(S_n = \text{function}(S_{n-1})\)

Ackermann’s function – pp. 177, kumar and agrawal
21.0 Lesson Goals

21.1 The Need for Arrays

Very often we need to store a lot data of the same type, e.g., marks of students in a class. It would be very cumbersome if we had to define one variable for each student. In addition to the number of variables becoming very large, it becomes difficult to operate on this data. In such cases, arrays can be used to store the entire data set using a single name. The various elements are assigned sequence numbers for purposes of retrieval.

21.2 Defining Arrays

Arrays are defined using a name and a size. The size specifies the number of elements that the array will contain.

   type  name[size];

For example,

   int rank[100];
   char usertext[80];
   float x[12];

The first definition is for an array of 100 integers named rank. The next defines an array of 80 chars named usertext. The last one defines an array of 12 float variables named x. It is good programming practice to use a symbolic constant to define the size of an array because it leads to better maintainability of the program. Let us look at the following array definitions.

   int rollnum[240];
   char grade[240];
   int rank[240];
If we need to change the size of all three arrays from 240 to 300, we need to make changes in three different places. Let us rewrite the same code using a symbolic constant `MAX_STUDENTS`.

```c
#define MAX_STUDENTS 240
int rollnum[MAX_STUDENTS];
char grade[MAX_STUDENTS];
int rank[MAX_STUDENTS];
```

Now, if we have to change the size of the three arrays to 300, we need to make only one change - the value of the symbolic constant is changed to 300.

### 21.3 Accessing an Array Element

Array elements are accessed using the array name and an index number, \( k \), in brackets. Here \( k \) must be a value lying between 0 and \( (\text{size}-1) \) where size is the number of elements used in the array definition. For example,

- `rank[0]` accesses the first element of the array `rank`,

Therefore, we can write statements like

```c
rank[3] = 14;
usertext[0] = 'H';
```

A common error is to use parentheses `()` instead of brackets `[]` in accessing an array element. If we write `rank(3)`, the compiler interprets it as a call to a function named `rank` with an argument of 3.

### 21.4 Initialization of Arrays

Arrays can be initialized at the time of definition by giving the initial values inside a pair of braces separated by commas.

```c
int nums[7] = {2, 4, 6, 8, 10, 12, 14};
char ans[3] = {'Y', 'N', 'U'};
float x[4] = {3.6, 7.6, 0.0, 8.9};
```

Here, we have given a value to each element of the array. When we perform complete initialization in this manner, we can skip the size specification of the array. The compiler will count the number of values given inside the braces and use that as the implied size. For example, we could have written the above definitions as

```c
int nums[] = {2, 4, 6, 8, 10, 12, 14};
char ans[] = {'Y', 'N', 'U'};
float x[] = {3.6, 7.6, 0.0, 8.9};
```

The implied size would be 7 for the array `nums`, 3 for the array `ans`, and 4 for the array `x`. 196
It is possible to initialize only the first few elements of an array. In such cases, all remaining values are automatically initialized to a value of zero. If we write

```c
int nums[7] = {2, 4, 6, 8};
float x[4] = {3.6, 7.6};
```


Note that this syntax is valid only for the initialization of arrays as a part of the definition statement. This cannot be used to assign values to an array in the program. For example,

```c
/* INVALID STATEMENTS */
nums[7] = {2, 4, 6, 8, 10, 12, 14};
nums = {2, 4, 6, 8, 10, 12, 14};
```

are both INVALID assignment statements in a program.

### 21.5 Character Arrays and Strings

Character arrays are defined and initialized in the same manner as arrays of any other data type.

```c
char name[60];
char ans[3] = {'Y', 'N', 'U'};
```

Strings are null terminated character arrays. In other words, to be a valid string, a character array must contain a null character (`\0`) in at least one of its elements. The following character arrays are valid strings.

```c
char game[5] = {'G', 'O', 'L', 'F', '\0'};
```

The following character arrays are not valid strings because they do not contain a null character.

```c
char fruit[5] = {'a', 'p', 'p', 'l', 'e'};
char tool[3] = {'a', 'x', 'e'};
```

Therefore, all strings are character arrays but all character arrays may not be valid strings.

### 21.6 Manipulating Arrays

In using an array, we must remember that

- we CANNOT assign to an array as a whole,
- we can only make assignments to the individual elements of the array, and
- individual elements behave just like ordinary variables of the data type of the array
For example, if we have defined the arrays

```c
#define MAX 20
int nums[MAX], val[MAX];
```

then,

- we cannot make assignments to `nums`,
- we can only make assignments to an element, `nums[k]`, where `0 ≤ k ≤ 19`,
- each element of the array `nums` behaves just like an ordinary `int` variable.

Let us take the simple problem of copying the elements of `nums` to `vals`. We can achieve this using a for loop

```c
for(j=0; j< MAX; j++) vals[j] = nums[j];
```

Notice that we manipulate the individual elements of the array one at a time. Let us next look at the problem of finding the sum of all elements of `nums`.

```c
sum = 0;
for(j=0; j< MAX; j++) sum += nums[j];
```

Once again, we take each individual element of the array and add it to `sum`.

### 21.7 Input/Output of Arrays

Input/output of arrays is also performed one element at a time. The following program `ARRAYIO.C` reads 6 integer value input by the user and prints them into a file named `ARRAYIO.OUT`. Notice the use of `for` loops both for the input and the output.

**Program 21.1 – ARRAYIO.C  Input and output of an array**

```c
/* ==----------------------------------------------------------------------
   | ARRAYIO.C ==----------------------------------------------------------------------
   | Demonstrates input/output of an array.
   */
#include <stdio.h>
#include <stdlib.h>
#define MAX 6

void main()
{
    int j,nums[MAX];
    FILE *outfil;
    for(j=0;j<MAX;j++)
    {
        printf("Enter integer #%2d >> ",j);
        scanf("%d",&nums[j]);
    }
    if((outfil=fopen("ARRAYIO.OUT","w"))==NULL)
    {
        printf("\nERROR: failed to open ARRAYIO.OUT\n");
        exit(1);
    }
```
One Dimensional Arrays 199

```c
for(j=0;j<MAX;j++)
    fprintf(outfil,"Integer #%2d is %d\n",j,nums[j]);
fclose(outfil);
printf("\nValues are stored in the file ARRAYIO.OUT\n");
```

21.8 Passing Arrays to Functions

The following program AVERAGE.C calculates the average of real values input by the user. It uses a function average.

Program 21.2 – AVERAGE.C Computing the average of an array of numbers

```c
/* ============================== AVERAGE.C ====================
Program to demonstrate passing of arrays to a function.
=============================================================*/
#include <stdio.h>
#define MAX 6
/* function prototype declaration */
float average(int nvals, float y[]);
void main()
{
    int j;
    float val[MAX],avgval;
    for(j=0;j<MAX;j++)
    {
        printf("Enter value  #%2d >> ",j);
        scanf("%f",&val[j]);
    }
    /* call to function average */
    avgval = average(MAX,val);
    printf("The average value is %f",avgval);
}
/* function definition for average */
float average(int nvals, float y[])
{
    int j;
    float sum;
    sum=0.0;
    for(j=0;j<nvals;j++) sum += y[j];
    return(sum/nvals);
}
```

Notice the three lines of code given in boldface. In the function call, we only give the name of the array as the argument as in `average(MAX, val)`. In the function definition and the prototype declarations, we use the array name along with an empty pair of brackets “[]” to indicate that y is an array.

We also need an integer argument `nvals` to inform the function how many elements are stored in the array y. In general, a function does not have any information about the size of the array passed as an argument. This information needs to be passed to the function through a separate argument.

21.9 Call by Reference
While studying functions, we learnt that variables are passed to a function only by value, i.e., the function cannot change the value of the argument in the calling function. Arrays are an exception to this rule because they are passed by reference. We will learn about the exact mechanism for this when we learn about pointers in Chapter 22. But for the time being, it is enough to remember that a function can change the values of the elements of any array passed to it. This is illustrated in the following program, CALBYREF.C, where the function foobar changes the values of the elements of the array v of the calling program. When the function foobar changes the value of v[2], the value of v[2] is being changed simultaneously because, internally, the two variables y[2] and v[2] share the same memory. This will become clear when we learn about the connection between arrays and pointers in Chapter 22.

Program 21.3 – CALBYREF.C  
Demo of call by reference for arrays

```c
#include <stdio.h>
#define MAX 6

void foobar(int nvals, int y[]);

void main()
{
    int j, val[MAX]={1,2,3,4};
    printf("Before call to foobar:");
    for(j=0;j<MAX;j++)
        printf("val[%2d] = %d",j,val[j]);
    foobar(MAX,val);
    printf("After call to foobar:");
    for(j=0;j<MAX;j++)
        printf("val[%2d] = %d",j,val[j]);
}

void foobar(int nvals, int y[])
{
    int j;
    for(j=0;j<nvals;j++)
        y[j] += 100;
    return;
}
```

Let us look at another program VECADD.C which uses a function vect_add to add two vectors (add the corresponding elements of two arrays) using the relation

\[ \text{sum}[k] = \text{x}[k] + \text{y}[k] \]

Program 21.4 – VECADD.C  
Adding two vectors

```c
#include <stdio.h>
#define MAX 8

void vect_add(int n, float x[], float y[], float sum[]);

void main(void)
{
    int j;
    ```
float p[MAX], q[MAX], r[MAX];
for(j=0; j<MAX; j++)
{
    printf("Enter p[%2d] and q[%2d] >> ", j, j);
    scanf("%f %f", &p[j], &q[j]);
}

vect_add(MAX, p, q, r);
printf("\n \n SUM VECTOR\t\t\n");
for(j=0; j<MAX; j++)
    printf("%2d  %f\n", j, r[j]);

void vect_add(int n, float x[], float y[], float sum[])
{
    int j;
    for(j=0; j<n; j++)
        sum[j] = x[j] + y[j];
    return;
}

Notice that the function vect_add changes the values stored in the array sum. In this manner, it affects a change in the values of the array r of the calling function. Here, the arrays r and sum share the same memory area for their elements.

21.10 Passing Strings to Functions

We have learnt that whenever we pass an array to a function, we must pass the number of elements as a separate argument. Otherwise, the function does not know how many elements of the array need to be used. The one exception to this rule is while passing strings. When we pass a valid string (i.e., a null terminated array of characters), we do not need to pass the number of characters used because the function can calculate this by looking for the null character that terminates the string.

The following program STLEN.C calculates the string length of some strings using a function nstrlen.

Program 21.5 – STLEN.C Function to compute string length

```c
/* ---------------------------------------------------------- STLEN.C ----------------------------------------------------------
 Function nstrlen calculates string length. 
---------------------------------------------------------------------------------------------------------------------------*/
#include <stdio.h>
int nstrlen(char s[]);

int main()
{
    char str1[]="Hello", str2[]="C Language";
    int n1, n2;
    n1 = nstrlen(str1);
    n2 = nstrlen(str2);
    printf("\n\"%s\" has %d characters.\n", str1, n1);
    printf("\n\"%s\" has %d characters.\n", str2, n2);
}

int nstrlen(char s[])
{
    int j=0;
    while(s[j++]!='\0') /* null statement */
return(j-1);

21.11 Points to Remember

Review Quiz

Programming Exercises
Programs using Files, Arrays and Functions

22.0 Lesson Goals

22.1 Insertion Sorting

Insertion sort is one of the simplest methods of sorting an array. The method used is very similar to the method used by cardplayers to sort their playing cards in order. The following program, PICKSORT.C, demonstrates the use of arrays for sorting the elements of an array using the picksort algorithm.

Notice that the function picksort has two arguments. One of them is the array being sorted, x[], and the second is the number of elements in the array being sorted, nx.

Program 22.1 – PICKSORT.C Insertion sort of an array of numbers

```c
/* ============================= PICKSORT.C ============================
This program demonstrates a straight insertion algorithm for sorting an array of integers. This should be used only for small arrays i.e., for N < 20.
Adapted from: Numerical Recipes in C by Press et al., Cambridge University Press
========================================================================*/
#include <stdio.h>
#include <stdlib.h>
define MAX 20
void picksort(int nx, int x[]);
void main(void)
{
    int j,npts,xval[MAX],xsort[MAX];
    FILE *indat,*outdat;
    char myfile[60];
    printf("Enter name of data file >> ");
    gets(myfile);
    indat = fopen(myfile,"r");
    if(indat == NULL) {
        printf("ERROR: File %s could not be opened");
        exit(1);
    }
    printf("Enter number of data points >> ");
    scanf("%d", &npts);
    for(j = 0; j < npts; j++) {
        scanf("%d", &xval[j]);
    }
    printf("Sorting begins...
");
    for(j = 0; j < npts; j++) {
        picksort(npts, xval);
        xsort[j] = xval[j];
    }
    printf("Sorted array:
");
    for(j = 0; j < npts; j++) {
        printf("%d ", xsort[j]);
    }
    printf("\n\n\nEnd of program.");
}
void picksort(int nx, int x[])
{
    int i, j, value, npts;
    for(i = 1; i < nx; i++) {
        value = x[i];
        j = i - 1;
        while(j >= 0 && x[j] > value) {
            x[j+1] = x[j];
            j = j - 1;
        }
        x[j+1] = value;
    }
}
```

```c
outdat = fopen("sorted.dat","w");
fscanf(indat,"%d",&npts);
for(j=0;j<npts;j++)
{
fscanf(indat,"%d ",&xval[j]);
xsort[j]=xval[j];
}
picksort(npts,xsort);
for(j=0;j<npts;j++)
{
printf("%2d %5d %5d",j,xval[j],xsort[j]);
fprintf(outdat,"%2d %5d %5d",j,xval[j],xsort[j]);
}

void picksort(int n,int x[])
{
    int i,j,temp;
    for(j=1;j<n;j++)
    {
        temp=x[j];
        i=j-1;
        while(i >=0 && x[i] > temp)
        {
            x[i+1]=x[i];
            i--;
        }
        x[i+1]=temp;
    }
}
```

22.2  Selection Sorting

22.3  Bubble Sorting

22.4  Cyclic Permutation of an Array

22.5  Printing Pascal's Triangle

cyclic permutation – left or right of an array or string.

** bubble sort ***

22.3  exchange sort

Printing Pascal's triangle

22.6  Calculating Mean and Standard Deviation
21.* Calculating Standard Deviation – pp.194, hutchison, just – 2 formulas
grading scheme problem – pp.176, kumar and agrawal

22.7 Fitting a Straight Line to Data
21.* Straight Line fit – (x[], y[], &m, &c, &rmsd)

22.8 The Sieve of Eratosthenes
21.* Computing Primes – Sieve of Eratosthenes
array storage

22.9 Computing Prime Numbers

22.10 Computing Factorials of Large Integers
21.* Long factorial calc using strings – program probably exists somewhere

22.11 Numerical Integration using Trapezoidal Rule
21.* Trapezoidal rule – for fixed function.

22.12 Plotting a Line Graph

22.13 Drawing a Histogram
Graph Plotter – one function

histogram - horizontal

22.14 Computing Harmonic Mean
** harmonic mean of an array of integers – must be done in double

22.15 A Hex Dump Program
Hex Dump program

22.* Points to Remember

Review Quiz
Programming Exercises

EXERCISE
1. Use the following set of integers as the input to the function \texttt{picksort} and trace the execution of the sorting algorithm by hand.

\begin{verbatim}
4 10 6 12 8 9 2
\end{verbatim}

EXERCISE
1. Write a program to read an integer array from a file and find its minimum and maximum value using two user defined functions named \texttt{array_min} and \texttt{array_max}.
\begin{verbatim}
int array_min(int n, int a[]);
ing array_max(int n, int a[]);
\end{verbatim}

2. Write another function named \texttt{is_in_array} that returns a value of 1 if a given integer \(b\) is found in \(a\), otherwise it returns 0.
\begin{verbatim}
int is_in_array(int n, int b, int a[]);
\end{verbatim}

3. Write a program to read an input string from keyboard. Write a function named \texttt{cleanup} to remove all non-alphanumeric characters from the string and change all letters to uppercase. Write a function named \texttt{reverse_it} which will return the reverse of the string. Use these functions to determine whether the input string is a palindrome. The following sentences are examples of valid palindromic sentences.

\begin{verbatim}
Madam, I'm Adam!
A man, a plan, a canal - Panama.
Able was I, Ere I saw Elba!
Too hot to hoot.
Rats drown in WordStar.
\end{verbatim}
Doc, note, I dissent! A fast never prevents a fatness; I diet on cod.

The function prototypes are given below.

```
void cleanup(char mystr[]);
Example input: Madam, I'm Adam!
After cleanup: MADAMIMADAM

void reverse_it(char mystr[]);
Example input: MUNDANE
After calling reverse_it: ENADNUM
```

3. Write functions to
   (a) calculate dot product of two vectors
   (b) perform scalar multiplication of a vector
   All vectors are assumed to be arrays of `double` type variables. Write the appropriate `main` function to read trial values for the vectors from a file and check these functions.

4. Write your own version of `strcmp` function named `n_strcmp`. Your function must have same type of arguments as `strcmp`.

5. Write a program that reads a rupee amount and writes it out in words with the word "only" added at the end. The amount can be as large as a few lakhs of rupees.
   Example: 1736728.95
   Rupees seventeen lakhs thirty six thousand seven hundred twenty eight and paise ninety five only
   Hint: Create an array of strings containing the words "one", "two", "eleven", "eighty", "thousand", "lakh", etc.

** print all permutations of a five character string

** polynomial evaluation using Horner's Rule – coeffs in array

Finding root using bisection

207
## 27

**Pointers**

### 23.0 Lesson Goals

### 23.1 The Address of a Variable

Every variable declared in a C program is allocated some bytes in the short term memory (i.e., Random Access Memory or RAM). The number of bytes allocated to a particular variable depends on the type of the variable and the particular implementation of C being used. You can refer back to Chapter 2 and 3 to look at the implementation dependent aspects of memory allocation for the various data types. To obtain the number of bytes used for any data type, we can use the `sizeof` unary operator as demonstrated in the program SIZES.C given in Chapter 8. For example, to find the number of bytes required for an `int` type variable named `xvar`, we can use either `sizeof(int)` or `sizeof(xvar)`.

Every byte of the memory has a sequence number. You can imagine the memory itself as a huge array with the first byte being numbered as 1, the second byte numbered as 2, and so on. Every variable is allocated a set of consecutive bytes in the memory. Let us assume that in a particular implementation of C, an int variable needs two bytes and a float variable needs four bytes. A picture of a section of the memory is shown in Figure 22.1.

![Diagram showing memory allocation for three variables](image)

**Figure 23.1** Diagram showing memory allocation for three variables

We see that
- the `int` variable `num` is allotted the bytes numbered 8234 and 8235 in the memory,
- the `float` variable `xval` is allotted the bytes numbering 8236 through 8239, and
- the `char` variable `usrans` is allotted the byte numbered 8240.
The number of the first byte allotted to a variable is known as “the address of the variable”.

- The address of `num` is 8234,
- the address of `xval` is 8236, and
- the address of `usrans` is 8240.

How do we find the address of a variable? We use a special unary operator called the address operator which is written as the symbol & prefixed to the variable. The following program demonstrates the use of the address operator.

```c
#include <stdio.h>
main()
{
  float xval=8.5;
  char usrans='Q';
  printf("\n\naddress of xval is %p\n",&xval);
  printf("\n value of xval is %f\n",xval);
  printf("\n\naddress of usrans is %p\n",&usrans);
  printf("\n value of usrans is %c\n",usrans);
  return 1;
}
```

Notice the use of the special format specification `%p` to print the value of the address. `%p` is an implementation dependent format to print addresses. Usually the addresses are printed out in hexadecimal form. We do not use any of the integer format specifications even though the address is a kind of integer value. A sample output from this program is shown below.

```
address of xval is FFF2
value of xval is 8.500000
address of usrans is FFF1
value of usrans is Q
```

For the variable `xval`, we note the following points:
- the name of the variable is `xval`,
- the type of variable is `float`,
- the value stored in the variable is `8.5`, and
- the address of the variable is `FFF2`.

Similarly, for the variable `usrans`, we note that
- the name of the variable is `usrans`,
• the type of variable is char,
• the value stored in the variable is 'Q', and
• the address of the variable is FFF1.

Running the program ADDRESS.C on different machines will generate different addresses for the variables num, xval, and usrans. Even on the same machine, these values are not reproducible because for different runs of the programs, the operating system may allocate different memory locations for the variables of our program.

The address of a variable cannot be explicitly changed using an assignment statement. Therefore, any statement of the form

```c
&x = memory address; /* illegal statement */
```

is an illegal statement. The allocation of memory to the variable x and assignment of an address is the task of the operating system of the computer and, therefore, it is illegal to assign an explicit value to an address. Think of the analogy of booking a room in a hotel. You go and request for a single room for Mr.X. The manager looks up his booking chart and gives you room number 210. You cannot demand for a specific room, say room number 105, because you do not know whether it is free or not. In a similar manner, you can ask for "memory room" for an int variable x and the operating system will try to accommodate your variable in some free area of the memory. But you cannot assign an explicit address to x because you do not know if something else is residing in that area of the memory.

Remember the simple rule that the address operator must never appear on the left hand side of an assignment statement.

### 23.2 Pointer Variables

Where do we store an address? None of the data types we have encountered so far can be used for this purpose. Addresses are stored in a special category of variables called pointer variables. Pointer variables are declared in the following manner.

```c
datatype   *ptrvar;
```

where `datatype` is any of the standard data types we have studied, and `ptrvar` is a variable of type "pointer to `datatype". For example,

```c
int   *nptr;
```

declares `nptr` to be a “pointer to an int”, and

```c
char   *cp;
```

declares `cp` to be a “pointer to a char”. It is useful to read a pointer variable declaration in two parts as shown by the underline.


\[
\text{datatype} * \quad \text{ptrvar};
\]

Therefore, \text{ptrvar} is a variable of type \text{(datatype *)}, i.e., a variable of type "pointer to \text{datatype}". For example, \text{nptr} is a variable of type "pointer to an int" variable and \text{cp} is a variable of type "pointer to a char". It is very common to declare pointer variables along with other variables as shown below.

\[
\text{int num, *nptr;}
\]

Here, \text{num} is a variable to type \text{int}. \text{nptr} is a variable of type "pointer to \text{int}", i.e., a variable of type \text{(int *)}.

Just as an \text{int} variable is different from a \text{float} variable, a "pointer to an \text{int}" variable is different from a "pointer to a \text{float}" variable, i.e., \text{(int *)} and \text{(float *)} represent two different types of variables even though they occupy the same number of bytes in memory. A "pointer to \text{datatype}" can only store the address of a variable of type \text{datatype}. Therefore, a pointer to an \text{int} can only store the value of an \text{int} variable and a pointer to a \text{float} can only store the address of a \text{float} variable. Given the declarations

\[
\text{int num, *nptr;}
\text{float xval;}
\text{float *xptr, *yptr;}
\]

the following observations can be made.

\[
\text{nptr} = \&\text{xval};
\text{xptr} = \&\text{xval};
\text{yptr} = \text{xptr};
\]

are legal assignment statements.

\[
\text{nptr} = \&\text{num};
\text{xptr} = \&\text{xval};
\text{xptr} = \text{nptr};
\]

are ILLEGAL assignment statements. In general, if \text{aptr} is a variable of type "pointer to \text{datatype}", then the following are the only legal statements.

\[
\text{aptr} = \&\text{gvar};
\text{aptr} = \text{bptr}; \quad /* \text{pointer variables of same type */}
\]

where \text{gvar} is a variable to type \text{datatype} and \text{bptr} is a variable of type "pointer to \text{datatype}".

If \text{aptr} and \text{bptr} are pointer variables of different types, then a typecast may be used to perform a valid assignment statement. With the declarations

\[
\text{datatype1} * \text{aptr;}
\text{datatype2} * \text{bptr;}
\]
the following two statements are syntactically correct statements.

\[
\begin{align*}
\text{aptr} &= (\text{datatype1} *)\text{bptr}; \\
\text{bptr} &= (\text{datatype2} *)\text{aptr};
\end{align*}
\]

For example, we can write the following

\[
\begin{align*}
\text{int } \text{num}, \ast \text{iptr}; \\
\text{long } \ast \text{jptr}; \\
\text{jptr} &= (\text{long} *) \&\text{num}; /* valid pointer typecast */ \\
\text{iptr} &= (\text{int} *) \text{jptr}; /* valid pointer typecast */
\end{align*}
\]

The program POINTER.C illustrates the use of pointer variables to store the address of a variable.

Program 23.2 – POINTER.C  Declaring and using pointer variables

```c
#include <stdio.h>

main() {
    int q, *qptr;
    q = 25;
    qptr = &q;
    printf("value of q is %d", q);
    printf("address of q is  %p", &q);
    printf("value of qptr is %p", qptr);
    printf("address of qptr is %p", &qptr);
    return 1;
}
```

A sample output from this program is shown below.

value of q is 25
address of q is  FFF2
value of qptr is  FFF2
address of qptr is  FFF0

It is illegal to assign an explicit value to a pointer variable. For example, if \text{xptr} is a pointer variable, then

\[
xptr = 48987; /* INVALID */
\]

is an illegal statement. The one exception to this is the assignment of the value 0 to a pointer variable. The value 0 is a special address and is used to indicate some kind of a special status for a pointer variable. Instead of using 0, it is good programming style to use the predefined symbolic constant NULL. Therefore,

\[
xptr = \text{NULL}; /* VALID */
\]
is a valid statement. We will study the use of NULL valued pointers in the subsequent chapters.

### 23.3 Pointers to void

A special kind of pointer variable can be declared with an unspecified data type. These are known as "pointers to void" and are used as generic pointers in special situations. They are declared as

```c
void  *uptr;
```

### 23.4 Pointers to Pointers

In the program POINTER.C of Section 22.2, we obtained the address of the pointer variable qptr. What kind of a variable could be used to store this address? To store the address of a pointer to an `int`, we will need a variable to type "pointer to a pointer to an int" as illustrated in the code section given below

```c
int *rptr, **sptr;
sptr = &rptr;
```

We could continue this game and declare "a pointer to a pointer to a pointer to an int" to store the address of a "pointer to a pointer to an int". We will postpone the study of such pointer variables to Chapter 23.

### 23.5 Initialization of Pointer Variables

Just as we do for ordinary variables, it is good practice to initialize all pointer variables before they are used in the program. If you do not have any initial value, initialize the pointer variable to a NULL value.

Pointer variables can also be initialized during declaration. A pointer variable can be initialized to the address of any variable that has been previously declared, i.e., declared before the pointer variable. For example, the declaration and initialization in the statements

```c
char q, *rptr;
rptr = &q;
```

can be combined into a single declaration as follows

```c
char q, *rptr = &q;
```

Note carefully that this is NOT the same as

```c
char q, *rptr;
*rptr = &q;  /* Illegal statement*/
```

The declaration

```c
char *rptr = &q, q;  /* Illegal declaration */
```
is illegal because \( q \) is declared after \( rptr \).

### 23.6 The Indirection Operator

The indirection operator denoted by * is a unary operator applied to a pointer variable to find the "value stored at the address in the pointer variable". Therefore, if we write

```c
char qq='Z', *rptr;
rptr = &qq;
```

\( *rptr \) (read "star rptr") will evaluate to 'Z' because the address of qq is stored in rptr and, therefore, the value at the address stored in rptr is the value of qq which happens to be 'Z'. Similarly, if we take

```c
int num=147, *sptr=&num;
```

\( *sptr \) (read "star sptr") will evaluate to 147 because the address of num is stored in sptr and, therefore, the value at the address stored in sptr is the value of num which happens to be 147. The program INDIROP.C illustrates the use of the indirection operator.

**Program 23.3 – INDIROP.C Use of the indirection operator**

```c
/* ============================================================== INDIROP.C ==============================================================
 * Program showing use of * (indirection) operator.                           
 ==============================================================*/
#include <stdio.h>
main()
{
    int p=4, q=6, *qptr;
    qptr = &q;
    printf("\n     q = %d",q);
    printf("\n     *qptr = %d",*qptr);
    printf("\n      *(&q) = %d",*(&q));
    qptr = &p;
    printf("\n     p = %d",p);
    printf("\n     *qptr = %d",*qptr);
    printf("\n return 1;"
}
```

A sample output from this program is shown below.

```
q = 6
*qptr = 6
*(&q) = 6
(&q) = 0FFE
(*(qptr) = 0FFE
```

\( *qptr = 4 \)
From this program we observe that if

\[ qptr = \&qq; \]

then the following expressions are always TRUE.

\[
qq == *(\&qq) \\
\&qq == \&(\*qptr)
\]

In the next few sections, we will look at various important aspects of using pointer variables.

### 23.7 Pointers and Functions

In Chapter 18, we have seen that functions are called by value in the C language. To allow a function to make changes to variables in the calling program, we can pass the address of this variable to the function and then use the indirection operator to change its value. The program MODFUNC.C illustrates this idea.

**Program 23.4 – MODFUNC.C**  
Pasing a pointer to a function

```c
/*  
** Program to demonstrate call by reference.  
** #include <stdio.h>  
void modfunc(int *j, int k);  
main() 
{  
    int x=4, y=5;  
    printf("Values of x & y before calling modfunc: \%d \%d",x,y);  
    modfunc(&x,y);  
    printf("Values of x & y after calling modfunc: \%d \%d",x,y);  
    return 1;  
}  
void modfunc(int *j, int k) 
{  
    *j = 8;  
    k = 9;  
    return;  
}  
*/
```

Running this program, we find that the value of \( x \) is changed to 8 by `modfunc` but the value of \( y \) remains unchanged as it is passed by value. Let us look at the sequence of steps in the execution of this program. Let us assume an address of 8723 for \( x \).

- When `modfunc` is called, the address of \( x \) is assigned to the variable \( j \) and the value of \( y \) is assigned to \( k \). Therefore, \( j \) equals 8723 and \( k \) equals 5 at the beginning of the function `modfunc`.
• The value of \( k \) is changed to 9 but this has no effect on the value of \( y \) which remains unchanged at 5.
• The value at the address in \( j \), i.e., the value stored in \( x \) is changed to 8.

A classic example of call by reference is shown in program SWAPFUNC.C where the values of two variables are swapped by a function \texttt{swapfunc}.

**Program 23.5 – SWAPFUNC.C Function to swap two values**

```c
/* ============================== SWAPFUNC.C ===========================
Program showing swapping two values using call by reference.
====================================================================*/
#include <stdio.h>
void swapfunc(int *x, int *y);
main()
{
    int p=4, q=8;
    printf("\n p = %d, q = %d",p,q);
    swapfunc(&p,&q);
    printf("\n after calling swapfunc\n p = %d, q = %d",p,q);
    return 1;
}
void swapfunc(int *x, int *y)
{
    int temp=*x;
    *x = *y;
    *y = temp;
    return;
}
```

Functions can return pointer variables. For example, the library function \texttt{strchr} has the following prototype

```
char *strchr(char *cs, char c);
```

which indicates that it returns a pointer to a \texttt{char}. The arguments for \texttt{strchr} are a pointer to a \texttt{char} (i.e., name of a string) and a \texttt{char} variable. The function searches the string \texttt{cs} for an occurrence of the character \texttt{c} and returns a pointer to its first occurrence. If the character is not found, the function indicates this failure by returning a \texttt{NULL} valued pointer. The following programs demonstrate the use of the functions \texttt{strchr}, \texttt{strrchr}, \texttt{strstr} and \texttt{strpbrk}.

**Program 23.6 – STRCHR.C Use of strchr library function**

```c
/* =============================== STRCHR.C ==========================
Program demonstrating use of string functions strchr and strrchr.
==================================================================*/
#include <stdio.h>
#include <string.h>
main()
{
    char first[100];
```
char q,*cptr;
printf("\n Input string >> ");
gets(first);
printf("\n Input character to search >> ");
scanf("%c",&q);
/* strchr finds first occurrence of character q */
cptr = strchr(first, q);
/* strrchr finds last occurrence of character q */
/*cptr = strrchr(first,q);*/
if (cptr)
  printf("strchr found character \'%c\' at position %d",q,cptr-first);
else
  printf("strchr did not find character.");
return 0;
}

Program 23.7 – STRSTR.C Use of strstr library function

Program 23.8 – STRPBRK.C Use of strpbrk library function
if (cptr != NULL)
{
    printf("strpbrk found character \"%c\" of second string in",*cptr);
    printf("\n first string at >>%s",cptr);
}
else
{
    printf("strpbrk didn't find any character of second string in");
    printf("\n first string");
}
return 0;
}

The function `strtok` can be used to separate a given string into "tokens". A token is a set of characters separated by some designated separators. The program STRTOK.C demonstrates the use of the `strtok` function.

**Program 23.9 – STRTOK.C Use of strtok library function**

```c
#include <stdio.h>
#include <string.h>
main()
{
    char first[100];
    char *tokens[20];
    char *cptr;
    char *seps = " ,"; /* designated separators */
    int j,num=0;
    printf("\n Input string >> ");
    gets(first);
    cptr = strtok(first,seps);
    while(cptr)
    {
        tokens[num] = cptr;
        cptr = strtok(NULL,seps);
        num++;
    }
    for(j=0;j<num;j++)
    {
        printf("\n   token # %2d is \"%s\",j,tokens[j]);
    }
    return 0;
}
```

A sample output from this program is shown below.

Input string >> C is a powerful language.
token # 0 is "C"
token # 1 is "is"
token # 2 is "a"
token # 3 is "powerful"
token # 4 is "language."
The declaration `char *tokens[20]` declares an array of 20 pointers to `char`. The pointers to the tokens returned by the `strtok` function are stored in this array of pointers. Similarly, we can declare

```c
int *x[100];
```
to be an array of 100 pointers to `int`.

### 23.8 Address Arithmetic

Addition and subtraction of an integer and a pointer variable are valid operations but they do not follow the rules of ordinary arithmetic. If `xptr` is a pointer to datatype and `num` is some integer type variable, then the expression

```c
xptr + num
```
yields an address value equal to address in `(xptr + num * sizeof(datatype))`. For example, if `xptr` is a pointer to an `int` storing an address of 4152 and the size of an `int` on this particular machine is 2 bytes, then `xptr + 6` will yield an address of 4164. This is demonstrated by the output of the program ADDARIT.C given below.

#### Program 23.10 – ADDARIT.C Address arithmetic for pointer variables

```c
/*
 * Addarit.C Address arithmetic for pointer variables
 *================================================================*/
#include <stdio.h>
main()
{
    char c, *cptr=&c;
    int n, *iptr = &n;
    double x, *dptr=&x;
    printf("\n size of char = %d",sizeof(char));
    printf("\n size of int = %d",sizeof(int));
    printf("\n size of double = %d",sizeof(double));
    printf("\n cptr = %p, iptr = %p, dptr = %p",cptr,iptr,dptr);
    cptr++;
    iptr++;
    dptr++;
    printf("\n after incrementing by 1");
    printf("\n cptr = %p, iptr = %p, dptr = %p",cptr,iptr,dptr);
    cptr+=3;
    iptr+=3;
    dptr+=3;
    printf("\n after adding 3");
    printf("\n cptr = %p, iptr = %p, dptr = %p",cptr,iptr,dptr);
    return 1;
}
```

### 23.9 Pointers and Arrays

The name of an array is a pointer variable containing the address of the first element of the array. In other words, if we declare

```c
int *x[100];
```
int xarr[10], *xptr;

xarr is a pointer to an int containing the address of the element xarr[0]. Further, the statements

    xptr = &xarr[0];
    xptr = xarr;
    xptr = &xarr;

are all equivalent. The first two are obvious because the name of the array contains the address of the first element of the array. The last one is an unusual assignment permitted by ANSI C which allows us to obtain the address of the array by using the address operator on the name of the array.

Even though the name of an array is a pointer variable, it is different from ordinary pointer variables with respect to one important operation. The name of an array CANNOT be assigned to by putting it on the LHS of an assignment statement because an array has a fixed location in memory which cannot be changed during the execution of a program. If we declare

    float yarr[10], *fptr;
then
    yarr = fptr; /* Illegal statement */
is an illegal statement. In general, any statement of the type arrayname = expression is illegal.

The program ARRAYPTR.C demonstrates the use of pointer notation to access elements of an array.

Program 23.11 – ARRAYPTR.C Use of array names as pointer variables

/or:==================================================================================
Program using the pointer nature of array names.
==================================================================================*/
#include <stdio.h>
main()
{
    float xarr[5]={2.3,3.1,4.5,1.1,6.8};
    float *aptr, *bptr;
    aptr = xarr;
    bptr = &xarr[0];
    printf("xarr[0]= %f,*aptr =%f, *bptr = %f",xarr[0],*aptr,*bptr);
    aptr++;
    bptr += 3;
    printf("xarr[1] = %f, *aptr = %f",xarr[1],*aptr);
    printf("xarr[3] = %f, *bptr = %f",xarr[3],*bptr);
    printf("*(aptr+2) = %f", *(aptr+2));
    printf("\n bptr-aptr = %d",bptr-aptr);
}

The output from this program is shown below.

xarr[0]= 2.300000,*aptr =2.300000, *bptr = 2.300000
We have seen in Chapter 21 that when an array name is used as an argument for a function, we specify it suffixed with brackets [], e.g., numarr[]. This can be substituted with a pointer type variable as shown in the program PTRARG.C below.

Program 23.12 – PTRARGS.C Passing array pointers to functions

```c
/* ==--------------------------------- PTRARGS.C ==-----------------------------*/
#include <stdio.h>
float minvalue(int n, float *yarr);
main()
{  float minx, xarr[5]={2.3,3.1,4.5,1.1,6.8};
    int n = 5;
    minx = minvalue(n, xarr);
    printf("\n Minimum value is %f",minx);
    return 1;
}
float minvalue(int n, float *yarr)
{  int k;
    float small;
    small = yarr[0];
    for(k=1;k<n;k++)
        if(yarr[k] < small) small = yarr[k];
    return small;
}
```

It is possible to pass only a part of an array to a function by passing the address of the element where we want to start. For example, foobar (&x[3]) and foobar(x+3) are both valid methods of passing the portion of the array x starting at the fourth element of x to a function foobar.

23.10 Arguments for scanf

Recall that in Chapter 18, we had started using the & symbol as a prefix to every argument in a call to the scanf function. It should be obvious by now that the scanf function requires the address of the variable to be input.

We had also seen that for a string variable, we do not need to put the & prefix. This can be explained now. Since every string is an array of characters, the name of the array is a pointer to char and, therefore, it is a valid argument for the scanf function. This is demonstrated in the following lines of a program.
char cc, name[20];
scanf("%c %s", &cc, name);

23.11 Legal Operations on Pointers

The following are legal operations on pointer variables.

- Assignment of a pointer variable to another pointer variable of the same type.
- Assignment of NULL value to a pointer variable (NULL is a predefined symbolic constant with a value of 0).
- Typecast a pointer variable to an integer type (used rarely).
- Addition and subtraction of an integer from a pointer variable.
- Subtracting two pointer variables pointing to two members of the same array.
- Comparing (using relational operators) two pointer variables pointing to two members of the same array.

The following are illegal operations on pointer variables.

- Addition of two pointer variables
- Multiplication and division of pointer variables
- Adding a non-integer value to a pointer variable
- Assignment of a pointer variable of one type to a pointer variable of another type without typecasting.

23.12 Pointers and Strings

Pointers to char used for storing strings can be initialized to a string but a pointer variable of any other type cannot be initialized to an array.

```
char *y = "Hello";
```

is a legal statement equivalent to

```
char y[] = "Hello";
```

But
```
int *p = {1,2,3,4,5,6}; /* illegal */
```

is an illegal statement.
The standard C library has a number of string related functions whose prototypes are given in STRING.H. In this section, we will look at the implementation details of some of these functions. Let us begin by writing our own version of the strlen function.

```c
int strlen(char *s)
{
    int n;
    for(n=0;*s != '\0';s++) n++;
    return(n);
}
```

Trace the function to understand how it works. Let us next look at a version of the strcpy function.

```c
void strcpy (char *s, char *t)
{
    while(*s++ = *t++ != '\0') /* null statement */
}
```

The use of two unary operators on a single operand in *s++ should be studied carefully. What is the correct order in which these two operations will take place? Should this be interpreted as *(p++) or as (*p)++? It is obvious that the two different sequences of operations will yield different results. It is in such instances that we must refer to the operator precedence and associativity tables given in Chapter *. We see that the two operators belong to the same precedence group. The associativity for these operators is right to left. Therefore, the increment operator, ++, will act first and the result will be *(p++) and NOT (*p)++.

** strrev from pp. 247, kumar and agrawal

23.13 Pointers to Streams

We had seen in Chapter 17 a file stream variable is declared as a variable of type FILE *. These are pointer variables of a special type. They point to a collection of data known as structs which are used to store the various pieces of data necessary for using the stream. We will learn more about structures in Chapter 29 but for now, you should remember that a stream variable is a pointer variable.

23.14 Dangling References

Let us look at the definition of a function dangle

```c
int * dangle(int num)
{
    int temp;
    return (&temp);
}
```

and a call to this function from main.
main()
{
    int *iptr, k=56;
    iptr = dangle(k);
}

The function dangle returns the address of the variable temp. The variable temp is an auto variable which ceases to exist once the function dangle is executed. Therefore, we now have the pointer variable iptr storing the address of a non-existent int variable. Any attempt to use the indirection operator on iptr is bound to cause trouble. Such a pointer is said to be a "dangling reference" and must be avoided. The simple rule is that we should never pass the address of a variable outside the scope of its existence.

23.15 Arrays of Pointers

pp.237, kumar and agrawal
add months example also

23.16 Pointers to Functions

23.17 Printing Trigonometric Tables

23.18 Numerical Integration

** check if
    char *x;
    x = "Hello"
is valid

23.* Points to Remember

Review Quiz

("apple" == "apple") TRUE or FALSE

    int x[] = {1,2,3};
    int y[] = {1,2,3};
(x == y) TRUE or FALSE

Programming Exercises

strcmpl
strtolower
strtoupper

integration using Simpson's 1/3 rd rule.
Multidimensional Arrays

24.0 Lesson Goals

24.1 Two Dimensional Arrays

We have seen that a one dimensional array can be used to store a set of similar objects. Once the data is stored in an array, we can use the array index to retrieve a particular value. In some cases, every item of data is related to two indices. For example, we might have a set of students and a set of subjects. Each student secures a certain percentage of marks in every subject. These marks can now be accessed with one index for a student and another index for a subject. Such cases call for storage of data in a two dimensional array. Many mathematical constructs such as matrices and determinants also require a two dimensional array for storage. A two dimensional array is declared as follows:

```
datatype arrname[nrows][ncols];
```

We visualize a two dimensional array as a matrix having rows and columns. For example, a two dimensionall array declared as

```
int q[5][3];
```

is shown graphically in Figure 23.1.
To initialize the two dimensional array with the values shown in Figure 23.1, we can write the declaration as follows.

```c
int q[5][3] = {{88,65,4,12,88},
               {93,3,26,26,72},
               {65,58,26,0,37}};
```

As with one dimensional arrays, any uninitialized values are set equal to 0. The number of rows is optional but the number of columns must be specified. Therefore, the above declaration may be written as follows.

```c
int q[][3] = {{88,65,4,12,88},
              {93,3,26,26,72},
              {65,58,26,0,37}};
```

Now, we can access any of these values by giving the appropriate first and second index. The first index must be lie between 0 and (nrows-1). The second index must lie between 0 and (ncolumns-1). For example,

- `q[0][1]` yields the value 65,
- `q[2][3]` yields the value 0, and
- `q[0][0]` yields the value 88.

### 24.2 Storage of Two Dimensional Arrays

Just like one dimensional arrays, two dimensional arrays are also allocated a single contiguous chunk of memory. However, we have a choice of two schemes for storing the values in a sequential fashion. For the array of Figure 23.1, we could choose to store the values column by column, i.e., `q[0][0]` followed by `q[1][0]`, `q[2][0]`, `q[0][1]`, `q[1][1]`, etc. Such a storage scheme is known as a column major storage. In this scheme, the first index changes fastest as we go through the list of elements. The alternative is to store the values rowwise. For our example array, this would mean storing `q[0][0]` followed by `q[0][1]`, `q[0][2]`, `q[0][3]`, `q[0][4]`, `q[1][0]`, etc. In this
scheme, the last index changes fastest and it is known as row major storage. In the C language, all multidimensional arrays are stored in row major form, i.e., with the last index changing fastest as we go through the sequence of elements.

### 24.3 Pointers and Two Dimensional Arrays

In the metaphor of the C language, a two dimensional array is simply "a one-dimensional array of one-dimensional arrays". Therefore, \( n[5][3] \) can be read as \( (n[5])[3] \) indicating "3 arrays of 5 elements each" or, in other words, "3 rows of 5 elements each". We had seen earlier in Chapter 21 that the name of every array was a pointer variable. Similarly, the name of every two dimensional array is "a pointer to an array of pointers". The program TWODARR.C demonstrates the use of these pointers to access the elements of a two dimensional array.

#### Program 24.1 – TWODARR.C Using pointers to access 2-dimensional array elements

```c
/* ============================================================== TWODARR.C ==============================================================
* Program demonstrates use of pointers to access elements of a 2-dimensional array. *
*===================================================================*/
#include <stdio.h>
main()
{
int q[][5]= {{88,65,4,12,88},
            {93,3,26,26,72},
            {65,58,26,0,37}};
int (*qptr)[5]; /* pointer to an array of 5 ints */
int *qp;
qptr = q;
qp = &q[0][0];
printf("q[0][0] = %d, %d, %d",q[0][0],**qptr,*qp);
printf("q[1][0] = %d, %d, %d",q[1][0],*(qptr+1),*(qp+5));
printf("q[0][1] = %d, %d, %d",q[0][1],*(qptr+1),*(qp+1));
printf("q[2][4] = %d, %d, %d",q[2][4],*(q+2)+4,*(qp+14));
return 0;
}
```

The output from the above program is shown below.

\[
\begin{align*}
q[0][0] &= 88, 88, 88 \\
q[1][0] &= 93, 93, 93 \\
q[0][1] &= 65, 65, 65 \\
q[2][4] &= 37, 37, 37
\end{align*}
\]

In this program, \( qp \) is an ordinary pointer to an \( int \) but notice the special type of pointer variable \( qptr \) declared in this program as

\[
\text{int (*qptr)[5];}
\]

This is different from the following declaration of an array of 5 pointers to \( int \).

\[
\text{int *qptr[5];}
\]
The declaration (*qptr)[5] declares a pointer to "an array of 5 int variables". Therefore, incrementing qptr by 1 increments the address in qptr by the "size of 5 int variables" and it now points to the element q[1][0]. Study the program carefully to understand the use of pointers to access elements of two dimensional arrays.

### 24.4 Multidimensional Arrays

Multidimensional arrays are arrays with multiple indices. The rules for multidimensional arrays can be obtained by generalizing the rules for two dimensional arrays. They are declared as follows.

```c
datatype arrname[n1][n2][n3][n4];
```

The first size, n1, is optional but all other values must be specified. The internal storage is in a row major fashion, i.e., with the last index changing the fastest. We can access an element of this array using

```c
arrname[j1][j2][j3][j4]
```

where

- 0 < j1 < n1,
- 0 < j2 < n2,
- 0 < j3 < n3, and
- 0 < j4 < n4.

### 24.5 Passing Multidimensional Arrays to Functions

When a multidimensional array declared as

```c
datatype arrname[n1][n2][n3][n4];
```

is passed to a function `foobar` as a formal argument, it must contain the exact values of n2, n3, and n4. Mentioning n1 is optional. Therefore, the function prototype can read

```c
foobar( . . . , arrname[n1][n2][n3][n4], . . . );
```

or as

```c
foobar( . . . , arrname[][n2][n3][n4], . . . );
```

### 24.6 Functions for Square Matrices

** isdiagonal for square matrix.
** isupper trianlg
** is lower trianlg
24.7 Solution of Linear System of Equations

Gaussian Elimination
** transpose

24.8 Points to Remember

Review Quiz

Programming Exercises
Dynamic Memory Allocation

25.0 Lesson Goals

25.1 Static Memory Allocation

For all the auto variables in a function, memory is allocated when the function is executing and this memory becomes free when the function exits. The amount of memory allocated to a variable or an array remains fixed from the start of the function to the point of exit from the function. In some cases, this leads to an inefficient use of memory. Arrays are a major culprit in this respect. Let us look at the following program for calculating the average of some values input by the user.

Program 25.1 – AVGVAL1.C Computing average of an array of numbers

```c
/* -=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-= AVGVAL1.C -=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=/
#include <stdio.h>
#define NMAX 100
main()
{
    float xval[NMAX],sum;
    int nval,j;
    printf("Input number of values >> ");
    scanf("%d",&nval);
    if(nval > NMAX)
    {
        printf("\nERROR: Maximum number of values is %d",NMAX);
        exit(0);
    }
    sum = 0.0;
    for(j=0;j<nval;j++)
    {
        printf("\nInput value #%3d >> ",j);
        scanf("%f",xval+j);
        sum += *(xval+j);
    }
    printf("\n\nThe average is %.2f\n",sum/nval);
}```
printf("\n\n Average = %f",sum/nval);
 return 0;
}

A sample output from this program is shown below

Input number of values >> 2
Input value #  0 >> 50.0
Input value #  1 >> 52.4
Average = 51.200000

Another sample output is shown below.

Input number of values >> 200
ERROR: Maximum number of values is 100

In the first case, we have declared an array of 100 values but used only two of these values. The remaining memory space could have been better utilized for storing some other variables. In the second case, we could not accommodate 200 values because the array has a declared size of only 100. So how do we select the best size for an array? We have to study all possible inputs and set the array size to be the maximum of these values. However, in most cases, this will lead to a lot of unutilized memory.

25.2 Dynamic Memory Allocation

The alternative to the static memory allocation is to determine the memory requirements during runtime and allocate the necessary amount of memory. This is known as dynamic memory allocation and there are three functions for achieving this. They are given below.

```c
void *calloc(nobj, objsize);
void *malloc(totsize);
void *realloc(void *p, totsize);
void free(void *p);
```

The `calloc` function allocates memory for `nobj` objects of size `objsize` bytes each. The `malloc` function allocates memory equal to `totsize` bytes. Both functions return a pointer to a `void`. This pointer to a `void` can be recast into a pointer of any other type using typecasting as shown in the following program.

Program 25.2 – AVGVAL2 Dynamic memory allocation for an array

```c
/*
 * Program demonstrates use of dynamic memory allocation
 * for an array.
 */
#include <stdio.h>
main()
{
    float *xval, sum;
    int nval, j;
    printf("\n Input number of values >> ");
    scanf("%d", &nval);
    xval = (float *)calloc(nval, sizeof(float));
    ...
if(xval==NULL)
{
    printf("\n ERROR: Could not allocate memory.");
    exit(0);
}

sum = 0.0;
for(j=0; j<nval; j++)
{
    printf("\nInput value %3d >> ",j);
    scanf("%f",xval+j);
    sum += *(xval+j);
}
printf("\n
 Average = %f",sum/nval);
free((void *)xval);
return 0;
}

In the statement

    xval = (float *)calloc(nval,sizeof(float));

We request for memory to be allocated for nval objects, each occupying the size of a float variable. The calloc function returns a pointer to a void. This pointer is recast into a pointer to a float using the (float *) typecast operator.

The major difference between calloc and malloc is that calloc initializes all the bytes of the allocated memory to 0 values whereas malloc does not perform any initialization. If for some reason, the memory allocation fails, this failure is signalled by returning a NULL pointer value. We use this for error trapping in the above program by checking if (xval==NULL) is true.

The function free deallocates the memory allocated to the pointer.

The realloc function can be used to change the allocation of memory to a pointer variable. This is shown in the function below which concatenates two input strings and stores them in a single string.

Program 25.3 – REALLOC.C Use of realloc library function

/*  ============================== REALLOC.C =========================
Program demonstrates use of realloc function.
====================================================================*/
#include <stdio.h>
main()
{
    char *bigstr, usrstr[80];
    int size;
    /* using malloc to allocate memory */
    printf("\n Input a string >>");
    gets(usrstr);
    size = strlen(usrstr)+1;
    bigstr = (char *) malloc(size);
    strcpy(bigstr,usrstr);
    puts(bigstr);
    /* using realloc to reallocate more memory */
    printf("\n Input second string >>");
```c
gets(usrstr);
size = strlen(usrstr)+1+strlen(bigstr);
bigstr = (char *) realloc(size);
strcat(bigstr,usrstr);
puts(bigstr);
free((void *)bigstr);
return 0;
```

The statement

```
size = strlen(usrstr)+1;
```

calculates the required memory for the string. The extra byte is for the null terminator character at the end of the string. Then, we use the malloc function to dynamically allocate memory. Next, we want to increase the size to accommodate the concatenation of the second string. For this, we use the realloc function. When the realloc function is called, the contents are unchanged up to the minimum of the new and old sizes. Therefore, the old string will remain undisturbed on calling realloc. However, realloc does not guarantee that the new memory block begins at the same address as the old memory block, i.e., the value of bigstr after calling realloc may be different from the value it had after calling malloc.

In the two programs AVGVAL2.C and REALLOC.C shown above, we do not have to call the free function because at the end of the execution of the entire program, all memory is automatically deallocated. Nevertheless, it is good programming practice to deallocate any dynamically allocated memory.

### 25.3 Points to Remember

**Review Quiz**

**Programming Exercises**
26.0 Lesson Goals

26.1 The Preprocessor

Every C program goes through two stages of conversion.

- In the first step, the process of macro substitution is carried out by the preprocessor. The end result is a modified C program. The preprocessor also carries out some other kinds of modifications to the program.
- In the second step, the modified source code is compiled into machine language code.

The preprocessor is a useful feature of the C language and it is important to learn the preprocessor directives. A preprocessor directive in a C program begins with the `#` (hash) character. In Chapter 1, we have already seen a simple preprocessor directive for the definition of symbolic constants as seen in the examples below.

```
#define NMAX       100
#define BIGVALUE   1.e999
```

Macro substitutions by the preprocessor can be broadly classified into three categories.

- Simple macro substitution (substitution of symbolic constants)
- Argumented macro substitution (expansion of macros with arguments)
- Nested macro substitution (macros containing macros)

26.2 Symbolic Constants

Symbolic constants result in simple macro substitution in the program. Wherever the symbolic constant is found, it is replaced by its corresponding value. Some examples are shown below.

```
#define MAXCOUNT 100
#define PI 3.14159
#define CITY "MUMBAI"
```
Let us look at the following code segment containing these symbolic constants.

```c
float areas[MAXCOUNT], radius[MAXCOUNT];
printf("\nCircular areas in %s",CITY);
area[0] = PI * radius[0] * radius[0];
```

After preprocessing, this code will be transformed into the following.

```c
float areas[100], radius[100];
printf("\nCircular areas in %s","MUMBAI");
area[0] = 3.14159 * radius[0] * radius[0];
```

Simple macro substitution will not change anything which is a part of a string. For example, the statement

```c
printf("\n CITY NAME is %s",CITY, CITYMAN);
```

will get transformed to

```c
printf("\n CITY NAME is %s","MUMBAI", CITYMAN);
```

The first occurrence of CITY is inside a string and, therefore, it is immune to macro substitution. The second occurrence of CITY is treated as a symbolic constant. The third occurrence of CITY is a part of a token CITYMAN and is not an independent token by itself. Therefore, it is not substituted.

Care must be taken to prevent unwanted side effects of macro substitution. For example, with a symbolic constant defined as

```c
#define PI 3.14159;
```

the statement

```c
area = PI * rad * rad;
```

gets transformed to

```c
area = 3.14159; *rad *rad;
```

which is a meaningless statement. Similarly, if we define

```c
#define DD 45 - 22
#define EE 11 - 6
```

then, the statement

```c
ratio = DD/EE;
```

gets transformed into
ratio = 45 -22/11 - 6;

which is not the desired result. This could be remedied by writing the original statement as

\[ \text{ratio} = \frac{DD}{EE}; \]

Symbolic constants are used by programmers to improve the readability of their programs. For example, with the definitions

```c
#define EQUALS ==
#define AND &&
#define OR ||
#define TRUE 1
#define FALSE 0
```

we can write

```c
if(x EQUALS y OR (x < 10 AND y > 20));
if(answer EQUALS FALSE);
```

** predefined constants __LINE__, __FILE__, __TIME, date, STDC ??**

26.3 Macros with Arguments

Macros with arguments are expanded by substituting every occurrence of the argument with the value supplied to the macro. Let us look at an example macro definition where we have a macro SQR with one argument.

```c
#define SQR(x)   ((x) * (x))
```

Now, the statements

```c
r = SQR(a);
p = SQR(sin(y));
q = SQR(1/b);
```

will get transformed to

```c
r = ((a) * (a));
p = ((sin(y)) * (sin(y)));
q = ((1/b) * (1/b));
```

Once again, we must be careful to avoid unexpected side effects resulting from incorrect macro definitions. Let us define

```c
#define BADSQR(a)    (a * a)
```

With this macro definition, the expression

```c
BADTSQR(a + 1)
```
gets transformed into

\[(a + 1 \times a + 1)\]

which equals \((2a+1)\) and not \((a+1)^2\)! Some other examples of commonly used macros are shown below.

```c
#define MAX (a,b)   ( ( (a) > (b) ) ? (a) : (b))
#define ABS(x)  (  ((x) > 0) ? (x) : (-(x)) )
#define STREQ(s1,s2)   (strcmp(s1,s2)==0)
```

The parameters of the macro are not replaced within a quoted string. To include a parameter inside a quoted string, we precede it with a \# symbol.

```c
#define myprint(value)   printf(#value " = %d\n",value)
```

With the above definition, `myprint(x)` will get expanded to

```c
printf("x = %d\n",x);
```

The `#value` is replaced with a quoted string "value". Next, the two strings "value" and " = %d\n" are concatenated to form one string.

The `##` operator placed between two arguments leads to a literal concatenation as shown below.

```c
#define join(x,y)    x ## y
```

Then, the expression `join(catch,22)` will lead to the expansion `catch22`.

### 26.4 Macros versus Functions

Let us look at the simple task of finding the square of a number using a function. For each data type, we will need a different function for calculating the square as shown in the prototypes given below.

```c
int sqrint(int n);
float sqrfloat(float x);
long sqrlong(long p);
```

The same task could be done using a macro definition.

```c
#define SQR(a)   ((a) * (a))
```

Now this can be used for any type of variable as shown below.

```c
int i, sqri;
float f, sqrf;
long il, sqrl;
sqri = SQR(i);
sqrf = SQR(f);
```
The C Preprocessor

sqril = SQR(il);

A type of a function argument needs to be specified whereas arguments for macros are typeless because they are used for macro level substitution. This is the main strength of using a macro with arguments instead of a function.

This is also the main drawback of using macros. If a function is passed an argument of the wrong type, it is possible for the compiler to catch this error. For example, the compiler has no way to check that \texttt{SQR(ptr)} is illegal is \texttt{ptr} is declared as a \texttt{char *} variable. Type matching is an important safety mechanism in the development of correct programs. By using macros with arguments, we can bypass this safety mechanism and, by doing so, land up in some trouble.

### 26.5 Nested Macros

Macros can contain other macros and they are all expanded sequentially. For example, let us consider the following macros

```c
#define SQR(a) ((a) * (a))
#define CUBE(x) (SQR(x) * (x))
```

The second macro is equivalent to the following macro definition

```c
#define CUBE(x) (((x) * (x)) * (x))
```

One must be extremely careful to avoid unwanted side effects when using nested macros.

### 26.6 Other Preprocessor Directives

There are a number of preprocessor directives related to conditional compilation of code. The \texttt{if}, \texttt{elif}, \texttt{else}, and \texttt{endif} can be used to select a particular portion of code to be compiled. The following program, \texttt{CONCOMP.C}, demonstrates the use of these directives for conditional compilation.

Program 26.1 – \texttt{CONCOMP.C} Preprocessor directives for conditional compilation

```c
/* ============================== CONCOMP.C ==========================
   Program demonstrates conditional compilation using
   preprocessor directives.
   ==========================================================================
*/
#include <stdio.h>
#define SHORTDEBUG 1
#define FULLDEBUG 1
main()
{
    int q[5]= {88,65,4,12,88};
    int sum=0,j;
    for(j=0;j<5;j++)
    {
        #if SHORTDEBUG
            printf(" Processing element number %d",j);
        #endif
        ...
    }
    #if SHORTDEBUG
        printf("\n Processing element number %d",j);
    #endif
```
#if FULLDEBUG
    printf("\n, value = %d",q[j]);
#endif
sum += q[j];
}  
printf("\n sum = %d",sum);
return 0;
}

The output from this program is shown below.

Processing element number 0, value = 88  
Processing element number 1, value = 65  
Processing element number 2, value = 4  
Processing element number 3, value = 12  
Processing element number 4, value = 88  
Sum = 257

If we change the value of FULLDEBUG to 0, we get the following output.

Processing element number 0  
Processing element number 1  
Processing element number 2  
Processing element number 3  
Processing element number 4  
Sum = 257

If we change both SHORTDEBUG and FULLDEBUG to 0, we get the following output.

sum = 257

Another use of these statements is to include implementation dependent code, for instance, some code containing non-standard C functions (i.e., functions not defined in the ANSI C library). For example,

#ifdef __BORLANDC__
    . . .
    code section 1
    . . .
#elif __MSC__
    . . .
    code section 2
    . . .
#else
    . . .
    code section 3
#endif

The ifdef directive returns TRUE if the symbolic constant is defined. Every compiler has a predefined symbolic constant as it's signature. For example, Borland's compilers use the symbolic constant __BORLANDC__ as their signature. If this symbol if found to be defined, code section 1 is compiled and other two code sections are ignored. If neither __BORLANDC__ or __MSC__ is found to be defined, only code section 3 is compiled and the other two sections are ignored.
26.7 Points to Remember

Review Quiz

Programming Exercises
27.0 Lesson Goals

27.1 Command Line Arguments

All the programs that we have seen so far have no arguments for the function main. It is possible for the operating system to pass arguments to the function main. Such arguments are appended to the command for executing the program and are known as command line arguments. Using command line arguments, we can reduce the number of input prompts in the program.

The function main can be called with two arguments as follows

   main (int argc, char *argv[])

where argc (argument count) is the number of command line arguments present on the command line. argv (argument vector) is a pointer to an array of character strings, i.e., it is a pointer to an array of pointers to strings. Therefore, argv[0] is a pointer to the first command line string and argv[argc-1] is a pointer to the last command line string. The ANSI standard specifies that argv[argc] must be a NULL pointer.

The following program ECHO1.C is from Kernighan and Ritchie. It echoes the command line arguments to the standard output by treating the arguments as an array of strings. A sample command line and the output from the program are shown below.

   c:\echo1 hello 1 2 3
   hello 1 2 3

Program 27.1 – ECHO1.C   Echoing command line arguments

   /*   ======================================================= ECHO1.C =======================================================
      ================================================================================================================*/
Command Line Arguments

#include <stdio.h>

main (int argc, char *argv[])
{
  int j;
  for(j=1;j<argc;j++)
    printf("%s",argv[j],(j<argc-1) ? " " : "");
  printf("\n");
  return 0;
}

The following program ECHO2.C does the same job as ECHO1.C but it uses pointer notation to carry out the task (from Kernighan and Ritchie).

Program 27.2 – ECHO2.C  Echoing command line parameters using pointers

/* =============================== ECHO2.C ========================
   Echo command line arguments using pointer notation.
   ===========================================================================*/
#include <stdio.h>
main (int argc, char *argv[])
{
  int j;
  while(--argc > 0)
    printf("%s",*++argv,(argc > 1) ? " " : "");
  printf("\n");
  return 0;
}

In the following sections, we will develop several small programs using command line arguments for passing input information.

27.2 Sum of Integers

We would like to write a program to add the integers given as command line arguments. The command line will be as follows

   c:\sumint 34  56 77

and the output will be the sum of the integers given as command line arguments. The program needs to convert each string into the equivalent integer value. This is done using the atoi function in the program SUMINT.C.

Program 27.3 – SUMINT.C  Sum of integers given as command line parameters

/*  ===========================================================================
   Sum of integers given as command line arguments.
   ===========================================================================*/
#include <stdio.h>
#include <stdlib.h>
main (int argc, char *argv[])
244

Chapter 31

{
int j, sum=0;
for(j=1;j<argc;j++)
sum += atoi(argv[j]);
printf("
sum = %d",sum);
return 0;
}

27.3

Copying Files

In this section, we will look at a program NCOPY.C which will copy a text file to another using
command line arguments. The program takes the name of the source file and the target file as its
arguments. An additional option can be specified as either '/l' to convert all characters to lowercase
or '/u' to convert all characters to uppercase. The option specifier can appear at place after the
program name. For example, all of the following commands are equivalent.
c:\ncopy
c:\ncopy
c:\ncopy

file1.txt file2.txt /u
/u file1.txt file2.txt
file1.txt /u file2.txt

Program 27.4 – NCOPY.C

Copying a text file to another

/* ============================== NCOPY.C ========================
Copies a text file to another.
===============================================================*/
#include <stdio.h>
#include <stdlib.h>
main (int argc, char *argv[])
{
int sfound,j,cc;
char option=' ';
char file1[40]="",file2[40]="";
FILE *infile, *outfile;
if(argc < 3)
{
puts("\nUsage: ncopy sfile tfile");
puts("
where sfile is source file");
puts("
and tfile is target file");
puts(" Optional flags:");
puts("
/l for converting to lowercase");
puts("
/u for converting to uppercase");
exit(0);
}
if(argc > 4)
{
puts("\nFATAL ERROR : Too many arguments");
exit(0);
}
sfound =0;
/* the option specification can be any of the arguments */
for(j=1;j<argc;j++)
{
if(strchr(argv[j],'/'))
{

244


option = argv[j][1];
if(option != 'u' && option != 'l')
{
    printf("\nFATAL ERROR : Illegal option /%c",option);
    exit(0);
}
else
{
    if(!sfound)
    {
        sfound = 1;
        strcpy(file1,argv[j]);
    }
    else
        strcpy(file2,argv[j]);
}
printf("\nSource File : %s",file1);
printf("\nTarget File : %s",file2);
if(option == 'l')
    puts("\n  Convert to lowercase");
else if(option == 'u')
    puts("\n  Convert to uppercase");
if( (infile = fopen(file1,"r")) == NULL)
    {
        puts("\nFATAL ERROR : Source file could not be opened.");
        exit(0);
    }
if( (outfile = fopen(file2,"w")) == NULL)
    {
        puts("\nFATAL ERROR : Target file could not be opened.");
        exit(0);
    }
while( (cc=fgetc(infile)) != EOF)
{
    if(option=='l') cc = tolower(cc);
    if(option=='u') cc = toupper(cc);
    fputc(cc,outfile);
}
fclose(infile);
fclose(outfile);
return 0;
}

Read the program carefully and note the precautions taken to trap and handle various kinds of errors.

27.4 Points to Remember

Review Quiz
Programming Exercises

calc a # b program with checking.

eample 2, pp.245, kumar and agrawal- base to base conversion.
28.0 Lesson Goals

28.1 Modularity

All the programs we have seen so far use a single file to store the program. But when we are dealing with large projects, it becomes necessary to break up and organize the program into several files. The smaller files are easier to maintain. We had earlier seen how to break up the program into several independent functions for clarity and ease of maintenance. Organizing the program into several files is the next level of modularity useful for large programs. A single file may contain one long function or several small functions related to each other. At other times, we may obtain an object code file from an outside source which we need to link with our programs.

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include "analysis.h"
FILE *infile;
void userinput(void)
{
    . . .
}

int funcalc(char user[])
{
    . . .
}

int nterm = 0;
double delta = 0.1e-6;
void main()
{
    . . .
}

int iterate(double guess)
{
    . . .
}

ANALYSIS.H
#define TRUE 1
#define FALSE 0

int validate(char []);
double findroot(void);
void errortext(int);
int iterate(double);

USERIN.C
#include <stdio.h>
#include "analysis.h"
extern int nterm;
extern double delta;
```
static void cleanup(char user[]) {
    . . .
}

int getinput(char user[]) {
    . . .
}

Figure 27.1 A Multifile Program

A dummy program organized into several files is shown in Figure 27.1. In the subsequent sections, we will learn about the various rules about the scope and linkage of the various program elements shown in this example.

### 28.2 Accessing External Variables & Functions

The scope of an external variable starts at the point of its first declaration to the end of the file. For this reason, in the file MAINPROG.C, the variables $nterm$ and $delta$ are not visible to the functions $userinput$ and $funcalc$. Normally, they are not visible to functions residing in other files. For example, the function $findroot$ in ANALYSIS.C cannot access the variable $nterm$ defined in MAINPROG.C.

If functions in other files need to access these variables, they can be declared to be of external storage class. For example, the extern declarations for $nterm$ and $delta$ used in USERIN.C will allow the functions $cleanup$ and $getinput$ to access the variables $nterm$ and $delta$ defined in MAINPROG.C.

Similarly, the filestream $infile$ is available to the functions in ANALYSIS.C by virtue of the extern declaration of the filestream.

### 28.3 Static Variables and Functions

Variables can be completely hidden from other files if they are declared to be of static storage class. For example, the static variables $iter$ and $oldroot$ in ANALYSIS.C cannot be accessed by any function in any other file.

The static storage class can also be applied to functions. For example, the function $cleanup$ in USERIN.C cannot be accessed by any function which is not in the file USERIN.C.

### 28.4 User Defined Header Files

We have already learnt how to include the header files of the standard library. These files contain the various declarations needed to use the functions correctly. Here we will look at user defined header files.

In the file ANALYSIS.C, we have defined four functions, namely, $validate$, $findroot$, $errortext$, and $iterate$, which we plan to use extensively in our program. Therefore, the
declarations of these functions must be available to the functions in the other files. For example, we need the declaration

```c
extern double findroot(void);
```

in MAINPROG.C. The extern storage is the default storage when no storage class is specified. Therefore, this declaration could as well be written as

```c
double findroot(void);
```

In this form, we could also use this declaration in the file ANALYSIS.C itself. To make life easier in terms of maintaining only a single set of declarations to be used by all functions, we create a header file, ANALYSIS.H, and place all the relevant declarations in it.

It is customary to give the same filename to the header file as the file containing the function definitions, e.g., ANALYSIS.H and ANALYSIS.C.

Notice the use of double quotes in

```c
#include "analysis.h"
```

instead of the usual angle brackets. This indicates that the header file will be found in a different location, i.e., a different search path is implied for this header file which may not be the same as that of the library files (which is indicated by angle brackets). The exact path location is implementation dependent. Usually, the use of double quotes indicates that the header file is located in the same directory as the source file.

### 28.5 Avoiding Repeated Inclusion of Header Files

It is possible for header files to include other header files and, in such cases, it becomes important that we avoid making multiple inclusions of the same file. For example, if we have a file CHECK.H as listed below

```c
#include "symbol.h"
define OK 1
```

and a source file, MAIN.C, with the following lines

```c
#include "check.h"
#include "symbol.h"
```

We have included the file SYMBOL.H twice in this program! The most common technique is to use preprocessor commands to avoid such duplications. We write the include file as follows

```c
#ifndef _SYMBOL_
define _SYMBOL_
/* header file declarations etc. */
#endif
```

```c
```
Now, if the SYMBOL.H file appears a second time in a file, it is not included because the symbolic constant _SYMBOL_ is already defined when the file was included the first time.

28.6 Managing Multifile Programs using Projects

When we have a program containing numerous files, it becomes difficult to keep track of modifications to the various files. If we make a change to one of the source file, say FOOBAR.C, then it is not necessary to compile all the source files again. It is enough to recompile only FOOBAR.C. However, this must be followed by the process of linking all the object files again. Assuming that we have made changes in several files and recompiled them, how do we remember if we have linked the latest versions of all files? These kinds of problems are handled using the concept of a project in most commercial C compilers.

A project manager utility keeps track of the various source files, object files and executable files. It eliminates the time and effort wasted in unnecessary compilation and linking operations. Let us take the example of the programs shown in Figure 27.1. There are three source programs, namely MAINPROG.C, ANALYSIS.C, and USERIN.C. Each of these will be compiled to an object file resulting in three object files, namely MAINPROG.OBJ, ANALYSIS.OBJ, and USERIN.OBJ. These three object files will be linked to create the single executable file, MAINPROG.EXE.

If a change has been made to ANALYSIS.C, the project manager utility must recompile ANALYSIS.C and then link all the three object files. How does it know which files to recompile? This is done using the information about the time of creation of a file. The operating system stores the information of the time and date when the file was created or last modified. By comparing the time stamps of the files, ANALYSIS.C and ANALYSIS.OBJ, it is possible to determine if the source file needs to be recompiled. Similarly, by comparing the time stamps of the object files against the time stamp of the executable file, it is possible to determine of the files need to be linked again.

The project information itself is stored in a separate file with some special extension, e.g., *.PRJ.

28.7 Location of Files

In a multifile program, it is possible for the different files to exist in different subdirectories. The access can be made by the proper specification of the file paths in the programs. For example,

#include "c:\mylib\analysis.h"

But this is a cumbersome process and leads to portability problems. If you ever take this program to another machine, it becomes necessary to create the subdirectory c:\mylib on that machine. It is better to keep the header file in the same directory as the source file and write

#include "analysis.h"

In general, it is good programming practice to keep all files related to one program in a single subdirectory and let the compiler use the default locations for files (which is usually the directory containing the main source file).
28.8 Points to Remember

**Review Quiz**

**Programming Exercises**
User Defined Data Types

29.0 Lesson Goals

29.1 Typedef

We have seen that the C language has numerous in-built data types like int, char, double, etc. The typedef keyword can be used to create new data type names. For example, we can declare

```c
typedef unsigned long LARGENUMBER;
```

and use LARGENUMBER as a valid data type in our program as shown in the declarations below

```c
LARGENUMBER julianday, numdays;
```

We will use the convention of naming typedefs in UPPERCASE throughout this book. The new data type name can be used wherever a data type is required, e.g., typecasts, function declarations, etc.

```c
/*function prototype declaration*/
unsigned short day_of_week(LARGENUMBER numdays);
/* typecast */
julianday = (LARGENUMBER) ndays;
```

Note that the typedef does not create a new data type. It merely provides for an alias or synonym for an existing data type. Use of typedefs improves the readability of the program.

The typedef also comes in useful for implementation dependent data types. For example, the data type (i.e., type of integer) returned by the sizeof operator is implementation dependent. This is hidden from the user by specifying the return type to be size_t where size_t is itself a typedef like the one shown below.

```c
typedef unsigned size_t;
```
In this way, the portability of a program is improved because only the `typedef` needs to be changed when the program is moved to a new platform.

Some of the older compilers do not implement the wide character data type, `w_char`, specified in ANSI C for accommodating non-English languages. They simply declare it to be an alias of `char` itself.

```
typedef char wchar_t;
```

### 29.2 Union

A union is a variable that holds different kinds of data in a single storage area. At any one time, it can hold any one kind of data. For example, we can declare

```c
union u_foo
{
    float xval;
    long num;
    char name[20];
} foobar;
```

`u_foo` is called a tag and defines a particular type of union. `foobar` is a union of type `u_foo`.

Now `foobar` can hold a float value or a long value or 20 characters. The memory allocation for `foobar` will depend on the largest of these three data types.

The syntax for accessing a member of the union is to write `union.member`. For example,

```
foobar.xval = 34.56;
```

It is the user's responsibility to make sure that the value retrieved is the same as the value that has been stored. Otherwise, we will retrieve some junk values. Let us examine the following code segment.

```
float yval;
foobar.num = 44444;
yval = foobar.xval;
```

The resulting value of `yval` will be meaningless. The last value stored in `foobar` was a `long` value (using `foobar.num`) and, therefore, `foobar` now contains a valid `long` value. Trying to retrieve a `float` value will obviously give meaningless results.

### 29.3 Enumerated Constants

When we want to define a set of related symbolic constants, it is possible to use the enumerated datatype. Let us take the example of defining a symbolic constant for each weekday.

```
#define SUN 0
#define MON 1
#define TUE 2
```
This can be replaced by an enumerated data type defined as follows.

```c
enum WEEKDAYS {SUN, MON, TUE, WED, THU, FRI, SAT};
```

This definition assigns each of the constants an integer value starting from 0. With this definition, every occurrence of WED will be replaced by its equivalent integer value of 3. The use of this `enum` type is shown in the program WEEKDAY.C given below.

```
Program 29.1 – WEEKDAY.C Computing the weekday for any date

/ * ========================================================================== WEEKDAY.C ==========================================================================
Finding the weekday for any given date.
============================================================================*/
#include <stdio.h>

void main()
{
    enum WEEKDAY {SUN, MON, TUE, WED, THU, FRI, SAT};
    enum WEEKDAY wkday;
    int rem[] = {0,3,3,6,1,4,6,2,5,0,3,5,1};
    int year, month, date, temp;
    printf("\nInput year  : ");
    scanf("%d",&year);
    printf("\nInput month : ");
    scanf("%d",&month);
    printf("\nInput date  : ");
    scanf("%d",&date);
    temp = year+date+5;
    if(month > 1) temp += rem[month-1];
    temp += (year-1)/4;
    if(year%4 == 0 && month>2) temp ++;
    wkday = temp %7;
    printf("\n The day of the week is ");
    switch(wkday)
    {
    case SUN: printf("Sunday");break;
    case MON: printf("Monday");break;
    case TUE: printf("Tuesday");break;
    case WED: printf("Wednesday");break;
    case THU: printf("Thursday");break;
    case FRI: printf("Friday");break;
    case SAT: printf("Saturday");break;
    }
}
```

To start the enumeration from some other integer, we can initialize one of the constants to an integer value. The subsequent constants get assigned sequential values. For example, the definition

```
enum WEEKDAYS {SUN, MON, TUE=6, WED, THU, FRI, SAT};
```

assigns values of 0 to SUN, 1 to MON, 6 to TUE, 7 to WED, 8 to THU, 9 to FRI and 10 to SAT.
29.4 Points to Remember

Review Quiz

Programming Exercises
30.1 Structures

Very often, we encounter a set of closely related data items. For instance, every student in a class has a gender (male or female), a roll number and date of birth. It is possible to create three arrays to store this information, one for storing the gender data, one for the roll number and one for the date of birth. But a more natural grouping is one in which all data connected with a particular student is kept together in some fashion. In the C language, this grouping of closely related data is done using structures.

30.2 Declaring Structures

A structure is a derived data type which can be declared in many ways. In general, the syntax of a structure declaration is as follows

```c
struct struct-tag
{
    member1;
    member2;
}
struct1, struct2;
```

The `struct-tag` is an optional name for a structure. Each member of a structure must be declared by type and name just like any ordinary variable. Here `struct1` and `struct2` are structures declared. The structure declaration must end with a semi-colon.

Let us look at an example.

```c
struct date
{
    unsigned short day;
    unsigned short month;
    unsigned int year;
} today, yesterday;
```
A new type of structure identified by the tag *date* has been declared. This will have three data members, namely, *day*, *month*, and *year*. Two struct variables, *today* and *yesterday*, have been declared to be structures of type *date*. Memory is allocated for the storage of the members of *today* and *yesterday*.

An alternative method is to declare the template (skeleton) for the structure without declaring any struct variables. Later this template can be used whenever new struct variables are required.

```c
/* recommended style for declaring structures */
struct date
{
    unsigned short day;
    unsigned short month;
    unsigned int year;
};
struct date today, yesterday;
```

Since, the tag is optional, it is possible to declare untagged structures as follows

```c
/* do not use this style of declaration */
struct
{
    unsigned short day;
    unsigned short month;
    unsigned int year;
} today, yesterday;
```

But this declaration cannot be reused for any another structures of the same type. Therefore, this is not a good style of programming. Yet another popular method of declaring structures uses a *typedef* to create a synonym for a struct.

```c
/* using typedef for declaring structures */
/* Highly recommended style */
typedef struct
{
    unsigned short day;
    unsigned short month;
    unsigned int year;
} DATETYPE;
DATETYPE today, yesterday;
DATETYPE tomorrow;
```

This creates highly readable code because we do not have to repeat the keyword *struct* everytime we declare or use a *struct*. Note that we continue with our convention of using uppercase names for typedefs.

At this point, we are ready to understand the name "FILE" that we had used to define filestreams. *FILE* happens to a typedef declared in *stdio.h*. The following is the complete declaration of *FILE* from the Borland C++ Compiler v.3.1.
/* Definition of the control structure for streams */
typedef struct {
    int             level;          /* fill/empty level of buffer */
    unsigned        flags;          /* File status flags */
    char            fd;             /* File descriptor */
    unsigned char   hold;           /* Ungetc char if no buffer */
    int             bsize;          /* Buffer size */
    unsigned char   _FAR *buffer;   /* Data transfer buffer */
    unsigned char   _FAR *curp;     /* Current active pointer */
    short           istemp;         /* Temporary file indicator */
    token;          /* Used for validity checking */
}       FILE;                           /* This is the FILE object */

(Copyright (c) by Borland International)

Other compilers might have a different set of members.

Structures can be declared to have arrays and structures as members. For example, let us create a structure to store data on a student. The structure must contain a character array to store the name and a struct to store the date of birth of the student.

    struct stud_rec
    {
        char name[40];
        unsigned long roll_num;
        struct date dob; /* date of birth */
    };
    struct stud_rec John;

or

    typedef struct
    {
        char name[40];
        unsigned long roll_num;
        DATETYPE dob; /* date of birth */
    } STUDENT;
    STUDENT John;

30.3 Initializing Structures
A structure can be initialized at the time of declaration by giving a set of constant values to each of its members. For example,

```c
struct date today = {15, 9, 1997};
DATETYPE yesterday = {14, 9, 1997};
```

Remember that a similar expression cannot be used for assigning values to a struct. For example,

```c
DATETYPE yesterday;
yesterday = {14, 9, 1997}; /* INVALID */
```

is an illegal statement.

### 30.4 Memory Allocation for Structures

Separate memory is allocated to each member of struct. Therefore, in most implementations, the number of bytes for a struct will be the sum of bytes required for each individual member of the struct. We can use the `sizeof` operator with a struct to find out the number of bytes required.

```
sizeof (struct date)
or
sizeof today
```

will return the number of bytes needed for the struct `today`. In general, we should not be surprised if the number of bytes required for a struct is more than the sum of the bytes required for its individual members. On some computers, additional bytes might be used up to satisfy memory alignment requirements.

### 30.5 Accessing Data in Structures

A member of the struct is referred to by using the dot operator (structure member operator) and writing `structname.member`. For example, `today.year` refers to the `year` member of the struct `today`. Note that the dot operator falls into the highest precedence category in the operator precedence table given in Chapter 13.

Similarly, `John.name[3]` refers to the 4th character in the name member of the struct `John`.

When structures contain other structures as members, the dot operator can be used repeatedly to access members of members as shown below.

```
John.dob.year
```

which refers to the member `year` of the struct `dob` which itself is a member of the struct `John`.

Structures can be assigned the values of another structure of the same type. If two structs have been declared as follows

```
struct date today, birthday={1,1,1960};
```
Then, the following assignment is valid

```c
today = birthday;
```

### 30.6 Arrays of Structures

We can declare arrays of structs just like we declare arrays of other data types.

```c
struct stud_rec newstudent[200];
STUDENT newstudent[200];
```

When referring to the members of a particular struct, we have to specify the index of the particular struct. For example, `newstudent[33].roll_num` refers to the roll number of the student with index number 33.

### 30.7 Structures and Pointers

We can declare a pointer to a structure just as we would do for an ordinary data type.

```c
struct date today, *dateptr=&today;
printf("The current year is \%d",(*dateptr).year);
```

Note the use of parentheses around `dateptr`. Without the parentheses, `*dateptr.year` would be interpreted as `*(dateptr.year)` because the dot operator has a higher precedence than the `*` indirection operator. Since, pointers to structures are used very frequently, an alternate operator is used to indicate a member of the struct referred to by the pointer.

```c
dateptr->year
```

is the exact equivalent of

```c
(*dateptr).year
```

The structure pointer operator is written by placing a "minus" symbol followed by the "greater than" symbol, i.e., `->`.

### 30.8 Passing Structures to Functions

Structures can be used as the arguments of a function and/or as the return data type of a function just like any other ordinary data type. They can be passed either by value or by reference just like any other data type and the same rules hold for structures. The following example shows a function prototype where the function is passed the value of `today`

```c
struct date next_date(struct date today);
```

The following function prototype uses pointers to structs
unsigned long diff_days(DATETYPE *today, DATETYPE *dob);

When large structures have to be passed to a function, it is faster to pass a reference because considerable time may be taken to copy the entire structure. However, it should be kept in mind that the data in any structure passed by reference can be changed by the function.

eample: cartesian to polar transformation function, using two structs.

30.9 Self Referential Structures

It is illegal for a structure to contain an instance of itself.

/* illegal structure definition */
struct illegal
{
  struct illegal foo;
  
}

But a structure can contain a pointer to an instance of itself. Such a structure is known as a self-referential structure.

/* self-referential structure definition */
struct selfref
{
  struct selfref *fooptr;
  
}

These self-referential structures play a very important role in the practical implementation of many useful data structures like linked lists, trees, etc.

30.10 Structures and Unions

Structures are used to store homogeneous data. What do we do if we need to store non-homogeneous data in structures? This can be accomplished by using unions as members of structures. The following example, adapted from Tanenbaum et al. is an excellent instance of the use of unions for storing non-homogeneous data.

We consider the data related to the insurance policies issued by an insurance company. Depending on whether the policy is for life insurance, auto insurance, or for home insurance, certain data is different from one policy to another. We can store this by creating different structures for each kind of policy and then creating a union of the three types of structures. An integer variable kind is used to store what kind of insurance policy is stored in a particular structure.

#define LIFE 1
#define AUTO 2
#define HOME 3

struct policy
{
int policy_num;
char name[40];
int amount;
float premium;
int kind; /* LIFE, AUTO or HOME */
union
{
  struct
  {
    char beneficiary[40];
    struct date birthday;
  } life;
  struct
  {
    int deduction;
    char license_num[20];
    int year;
  } auto;
  struct
  {
    int deduction;
    int yearbuilt;
  } home;
};

30.11 Points to Remember

Review Quiz

Programming Exercises
Date and Time Functions

31.0 Lesson Goals

31.1 Using Date and Time Functions

The header file TIME.H is a part of the standard C library and contains the prototypes and definitions for the various functions related to time and date. `time_t` is a predefined `typedef`, usually of some integral type, for storing time values. A predefined struct definition, `struct tm`, is available for storing detailed time and date information. The members of this `struct` are shown below.

```c
struct tm {
    int    tm_sec;        // seconds (value between 0 and 59)
    int    tm_min;        // minutes (value between 0 and 59)
    int    tm_hour;       // hours (value between 0 and 23)
    int    tm_mday;       // day of the month (value between 1 and 31)
    int    tm_mon;        // month (value between 0 and 11)
    int    tm_year;       // year (between 0 and 99)
    int    tm_wday;       // day of the week (value between 0 and 6)
    int    tm_yday;       // day of the year (value between 0 and 365)
    int    tm_isdst;      // flag for Daylight Saving Time
};
```

The year member gives the number of years since 1900. The `tm_wday` member represents Sunday as 0, Monday as 1, and so on. Daylight Saving Time is a feature used in the USA where the clock timings are changed twice every year by exactly one hour.

The basic function to obtain the current date and time is the function `time` whose prototype is given below.

```c
time_t time(time_t *tptr);
```
It sets the value of *tptr to the value of current date and time. It returns the current date and time. If we do not want to use an argument, *tptr can be replaced with a NULL value.

There are a number of other functions to manipulate the value obtained by the time function. They are the difftime, mktime, asctime, ctime, gmtime, localtime, and strftime functions having the following prototypes.

double difftime(time_t t1, time_t t2);
time_t mktime(struct tm *tptr);
char *asctime(const struct tm *tptr);
cchar *ctime(const time_t *tptr);
struct tm *gmtime(const time_t *tptr);
struct tm *localtime(const time_t *tptr);
size_t strftime(char *s, size_t max, const char *format,
               const struct tm *tptr);

The following program TIMEFUNC.C demonstrates the use of these functions.

Program 31.1 – TIMEFUNC.C Using the time and date library functions

/* ============================= TIMEFUNC.C =========================
 Program showing use of time and date functions.
=================================================================* /
#include <stdio.h>
#include <time.h>
#include <math.h>
int main(void)
{
    time_t t, tgm;
    struct tm *localt_ptr, *gmt_ptr;
    double timediff;
    int j,k;
    char datestr[80];
    /* use of date and time functions */
    time(&t);
    printf("Current date & time           : %s", ctime(&t));
    localt_ptr = localtime(&t);
    printf("Local time is                 : %s", asctime(localt_ptr));
    gmt_ptr = gmtime(&t);
    printf("Coordinated Universal time is : %s", asctime(gmt_ptr));
    tgm = mktime(gmt_ptr);
    timediff = difftime(tgm,t);
    printf("Difference = %lf seconds",timediff);
    localt_ptr = localtime(&t);
    /* demonstration of the strftime function */
    strftime(datestr,80,"\nDATE: %d-%m-%Y",localt_ptr);
    puts(datestr);
    strftime(datestr,80,"\nA, Day #j of %b",localt_ptr);
    puts(datestr);
    strftime(datestr,80,"\nX %p %Z",localt_ptr);
    puts(datestr);
    strftime(datestr,80,"\nX %d-%m-%Y",localt_ptr);
    puts(datestr);
    /* members of the struct */
    printf("\nseconds: %d",localt_ptr->tm_sec);
Date and Time Functions

printf("\n minutes : %d",localt_ptr->tm_min);
printf("\n hours   : %d",localt_ptr->tm_hour);
printf("\n day     : %d",localt_ptr->tm_mday);
printf("\n month   : %d",localt_ptr->tm_mon);
printf("\n year    : %d",localt_ptr->tm_year);
printf("\n wday    : %d",localt_ptr->tm_wday);
printf("\n yday    : %d",localt_ptr->tm_yday);
printf("\n DST flag: %d",localt_ptr->tm_isdst);
return 0;
}

A sample output from this program is shown below.

Current date & time           : Tue Apr 28 14:47:40 1998
Local time is                 : Tue Apr 28 14:47:40 1998
Coordinated Universal time is : Tue Apr 28 18:47:40 1998
Difference = 14400.000000 seconds
Current date and time is:
DATE: 28-04-1998
Tuesday, Day #28 of April
Day #118 and week #17 of the year
14:47:40 PM EDT
seconds : 40
minutes : 47
hours  : 14
day   : 28
month : 3
year  : 98
wday  : 2
yday  : 117
DST flag: 1

31.2 Measuring Elapsed Time

We often need to measure the time taken to execute a program or a portion of a program. This time is measured in ticks. A predefined symbolic constant, CLK_TCK, defined in TIME.H specifies the number of ticks per second. To convert the time measured in ticks to seconds, we divide the number of ticks by CLK_TCK. The following program shows how to measure the elapsed time for a program using the clock function. The clock function returns a value of type clock_t which is a predefined typedef.

Program 31.2 – CLKTIME.C  
Measuring elapsed time using the clock function

/* Program showing use of clock function to calculate elapsed time.

#include <stdio.h>
#include <time.h>
*/

#include <math.h>
#define LARGENUM 200000

int main(void)
{
    clock_t etime;
    unsigned long j;
    printf("Ticks per second = %6.2f",CLK_TCK);
    /* finding processor time elapsed using clock function */
    /* do a time consuming calculation */
    for(j=0;j<LARGENUM;j++)
        log(pow(4444.55L,20.5L));
    etime = clock();
    printf("Elapsed time = ");
    printf("%lu ticks = %8.2f secs.",etime,etime/CLK_TCK);
    /* repeat the calculation */
    for(j=0;j<LARGENUM;j++)
        log(pow(4444.55L,20.5L));
    etime = clock();
    printf("Elapsed time = ");
    printf("%lu ticks = %8.2f secs.",etime,etime/CLK_TCK);
    return 0;
}

A sample output from this program is shown below.

Ticks per second =  18.20
Elapsed time = 63 ticks =     3.46 secs.
Elapsed time = 126 ticks =     6.92 secs.

31.3 Points to Remember

Review Quiz

Programming Exercises
32.1 Sorting using the qsort Function

Sorting is an operation that needs to be carried out very often in programs. The C function library contains a function, qsort, for the sorting of an array. This is an implementation of the quicksort sorting algorithm. In this book, we will learn to use the qsort function without going into the algorithmic details of quicksort. For more details, you can refer to any standard textbook on Data Structures or Numerical Analysis. The qsort function is a part of stdlib and has the following prototype.

```c
void qsort(void *base, size_t nelem, size_t width, int (*fcmp)(const void *, const void *));
```

where
- `base` is the address of the first element (element number 0) of the array,
- `nelem` is the number of elements in the array,
- `width` is the size of each element of the array, and
- `fcmp` is a pointer to a function which takes two constant pointers to `void` (`const void *`) and returns an `int` value. The name of a function acts as a pointer to the function.

The function `fcmp` is a user defined function which specifies the relative position of the two elements passed to it in the following fashion. The function `fcmp`
- returns a negative integer if *a precedes *b in the sorted array,
- returns 0 if they *a and *b are equivalent for the purpose of sorting, and
- returns a positive integer if *a follows *b in the sorted order.

The following program, SORTFLOT.C, demonstrates how to sort an array of `float` values using the `qsort` function.
Program 32.1 – SORTFLOT.C    Sorting an array of float values

/* ==============================================================
 * SORTFLOT.C ==============================================================
 * Sorting an array of float values using qsort.
 * ==============================================================*/
#include <stdio.h>
#include <stdlib.h>
int flotcmp(const void *a, const void *b);

main()
{
    float xval[8]={3.3,4.4,1.1,0.3,0.7,5.6,2.2,1.1};
    size_t nelem, esize;
    int j;
    nelem = 8;
    esize = sizeof(float);
    qsort((void *)xval,nelem, esize, flotcmp);
    printf("\n SORTED VALUES\n");
    for(j=0;j<8;j++)
        printf("xval[%d] = %f",j,xval[j]);
    return 0;
}

int flotcmp(const void *a,const void *b)
{
    float diff;
    float *p, *q;
    p = (float *)a;
    q = (float *)b;
    diff = (*p - *q);
    if(diff<0.0)
        return(-1);
    else if (diff == 0.0)
        return(0);
    else
        return(1);
}

Notice the important pointer conversions used in the program. The address of the array xval is
typecast into a pointer to a void by writing (void *)xval. In the function flotcmp, the
pointers are typecast into pointers to float because we are dealing with an array of float values.
Instead of writing

    float *p, *q;
    p = (float *)a;
    q = (float *)b;
    diff = (*p - *q);

more experienced programmers may write

    diff = (*(float *)a - *(float *)b);

The const specifier makes sure that the values of the arguments, i.e., a and b, cannot be changed in
the body of the function. The const specifier makes it illegal have any statement that can change the
value of either a or b in the function.
By exchanging the values 1 and -1 returned by `flotcmp`, we can obtain a decreasing order sort. The output from this program is shown below.

```
SORTED VALUES
xval[0] = 0.300000
xval[1] = 0.700000
xval[2] = 1.100000
xval[3] = 1.100000
xval[4] = 2.200000
xval[5] = 3.300000
xval[6] = 4.400000
xval[7] = 5.600000
```

The following program sorts the characters of a string. Notice that specifying the number of elements using `strlen` function keeps the null terminator character unaffected during the sorting.

```
Program 32.2 – SORTCHR.C  Sorting the characters in a string
/*  ===========================================================================
    SORTCHR.C  ===========================================================================
    Sorting an array of int values using qsort.
    ===========================================================================*/
#include <stdio.h>
#include <stdlib.h>
int charcmp(const void *a, const void *b);
main()
{
    char name[]="BrobdingnagianProgram";
    printf("BEFORE SORTING : %s",name);
    qsort((void *)name,strlen(name),1,charcmp);
    printf("AFTER SORTING : %s",name);
    return 0;
}
int charcmp(const void *a, const void *b)
{
    return(*(char *)a - *(char *)b);
}
BEFORE SORTING : BrobdingnagianProgram
AFTER SORTING : Bpaaabdgggiimnnnoorrr
```

The following program, SORTSTR.C, demonstrates the sorting of an array of strings using the `qsort` function. We use the `strcmp` function to compare the strings.

```
Program 32.3 – SORTSTR.C  Sorting an array of strings
/*  ===========================================================================
    SORTSTR.C  ===========================================================================
    Sorting an array of strings using qsort.
    ===========================================================================*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int funcmp(const void *a, const void *b);
```
main()
{
    char words[6][8] = {"cat", "CAT", "cattle", "apple", "1234", "zoo"};
    int j;
    qsort((void *)words, 6, sizeof(words[0]), funcmp);
    printf("AFTER SORTING IN REVERSE LEXICOGRAPHIC ORDER\n");
    for (j = 0; j < 6; j++)
        puts(words[j]);
    return 0;
}

int funcmp(const void *a, const void *b)
{
    /* for sorting in lexicographic order */
    /* return (strcmp((char *)a, (char *)b)); */
    /* for sorting in reverse lexicographic order */
    return (-strcmp((char *)a, (char *)b));
}

The output from this program is shown below.

AFTER SORTING IN REVERSE LEXICOGRAPHIC ORDER
zoo
cattle
cat
apple
CAT
1234

32.2 Searching using bsearch

The binary search is an efficient algorithm for searching in a sorted array of values. The function, bsearch, from the stdlib library performs a binary search. However, we must remember to sort the array before using bsearch. In the program, SERCHINT.C, given below, we first sort an array of int values using the qsort function. Then, we search for the occurrence of particular key values in the array using the bsearch function.

Program 32.4 – SERCHINT.C Searching in an array of integers

/*
 * =============== SERCHINT.C ===============
 * Sorting an array of int values using qsort and
 * searching for a key in the array using bsearch.
 * ===============
 */
#include <stdio.h>
#include <stdlib.h>
int intcmp(const void *a, const void *b);
main()
{
    int key, *iptr;
    size_t nelem, esize;
    int j;
    nelem = 9;
    esize = sizeof(int);
    qsort((void *)xval, nelem, esize, intcmp);
printf("\n SORTED ARRAY\n");
for(j=0;j<8;j++)
   printf("\nxval[%d] = %d",j,xval[j]);
key = 11;
 iptr = (int *)bsearch((void *)&key,(void *)xval,nelem,esize,intcmp);
if(iptr!=NULL)
   printf("\nkey = %d found at index %d",key,iptr - xval);
else
   printf("\nkey = %d not found",key);
key = 34;
 iptr = (int *)bsearch((void *)&key,(void *)xval,nelem,esize,intcmp);
if(iptr!=NULL)
   printf("\nkey = %d found at index %d",key,iptr - xval);
else
   printf("\nkey = %d not found",key);
return 0;
}
int intcmp(const void *a,const void *b)
{
   int *p, *q;
   p = (int *) a;
   q = (int *) b;
   return(*p - *q);
}

SORTED ARRAY
xval[0] = 3
xval[1] = 11
xval[2] = 11
xval[3] = 17
xval[4] = 22
xval[5] = 33
xval[6] = 44
xval[7] = 56
key = 11 found at index 2
key = 34 not found

Owing to the nature of the binary search algorithm, the bsearch function may not return the first occurrence of the key as seen from the output. The key value of 11 occurs at index 1 and 2 but bsearch found the second occurrence of the key value.

32.3 Sorting and Searching in Structs

Sorting an array of struct variables is an operation that is required very often for database related applications. When sorting an array of struct variables, we can choose any of the members of the struct to be the sorting key. Different comparison functions are required for sorting with the various possible keys. The program STUDRECS.C reads values into an array of structs from a text file named STUDRECS.DAT. Subsequently, we use qsort to sort the data by different keys.

The function bsearch can be used to search an array of structs for the occurrence of a particular key value. In the program STUDRECS.C, we search for the occurrence of a particular value of the rollnum member of the struct.

Program 32.5 – STUDRECS.C Sorting and searching student records
/* ============================= STUDRECS.C ====================
   Sorting an array of structs using qsort and searching for a key value using bsearch. 
   ===============================================================================*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define NMAX 20

typedef struct
   {
   char lastname[12];
   char firstname[12];
   char sex;
   int rollnum;
   float gpa;
}STUDRECORD;

void read_record(STUDRECORD *recptr, FILE *fptr);
void display_record(int recnum, STUDRECORD *recptr);
int lastname_cmp(const void *a, const void *b);
int rollnum_cmp(const void *a, const void *b);
int gpa_cmp(const void *a, const void *b);

main()
{
   STUDRECORD srecs[NMAX],key,*kptr;
   int j,nstudents = 6,nrec;
   FILE *infile;
   if( (infile=fopen("STUDRECS.DAT","r")) ==NULL) 
   { 
      printf("\nFATAL ERROR : Unable to open SORTRECS.DAT\n");
      exit(0); 
   }
   for(j=0;j<nstudents;j++) 
      read_record(srecs+j,infile);
   fclose(infile);
   /* initialize search key */
   strcpy(key.lastname,"");
   strcpy(key.firstname,""),
   key.sex='F';
   key.rollnum=0;
   key.gpa=0.0;
   /* display unsorted records */
   printf("\nUNSORTED RECORDS\n");
   for(j=0;j<nstudents;j++)
     display_record(j,srecs+j);
   /* display records sorted by lastname*/
   qsort((void *)srecs,6,sizeof(STUDRECORD),lastname_cmp);
   printf("\n\nSORTED BY LASTNAME\n");
   for(j=0;j<nstudents;j++)
     display_record(j,srecs+j);
   /* display records sorted by decreasing gpa*/
   qsort((void *)srecs,6,sizeof(STUDRECORD),gpa_cmp);
   printf("\n\nSORTED BY GPA\n");
   for(j=0;j<nstudents;j++)
     display_record(j,srecs+j);
   /* display records sorted by rollnum*/
   qsort((void *)srecs,6,sizeof(STUDRECORD),rollnum_cmp);
   printf("\n\nSORTED BY ROLL NUMBER\n");
   for(j=0;j<nstudents;j++)

/* search for a record with given roll number */
key.rollnum = 237;
kptr = (STUDRECORD *)bsearch((void *)&key,
(void *)srecs,6,sizeof(STUDRECORD),rollnum_cmp);
if(kptr != NULL)
{
    nrec = kptr - srecs;
    printf("\nKey: rollnum = 237 found at record # %d",nrec);
    display_record(nrec,srecs+nrec);
}
else
    printf("\nKey not found");
return 0;

int lastname_cmp(const void *a, const void *b)
{
    STUDRECORD *p, *q;
p = (STUDRECORD *)a;
q = (STUDRECORD *)b;
return strcmp(p->lastname,q->lastname);
}

int rollnum_cmp(const void *a, const void *b)
{
    STUDRECORD *p, *q;
p = (STUDRECORD *)a;
q = (STUDRECORD *)b;
return (p->rollnum - q->rollnum);
}

int gpa_cmp(const void *a, const void *b)
{
    STUDRECORD *p, *q;
float diff;
p = (STUDRECORD *)a;
q = (STUDRECORD *)b;
diff = p->gpa - q->gpa;
if(diff < 0.0)
    return(1);
else if(diff > 0.0)
    return(-1);
else
    return(0);
}

void read_record(STUDRECORD *recptr, FILE *fptr)
{
    char temp[41]="\n"
    char *cc;
    int newline='\n';
fgets(temp,40,fptr);
    if((cc=strchr(temp,newline)) != NULL) *cc='\0';
    strcpy(recptr->lastname,temp);
fgets(temp,40,fptr);
    if((cc=strchr(temp,newline)) != NULL) *cc='\0';
    strcpy(recptr->firstname,temp);
fgets(temp,40,fptr);
    if((cc=strchr(temp,newline)) != NULL) *cc='\0';
    strcpy(recptr->lastname,temp);
fgets(temp,40,fptr);
    recptr->sex = temp[0];
```c
fgets(temp,40,fptr);
recptr->rollnum = atoi(temp);
fgets(temp,40,fptr);
recptr->gpa=atof(temp);
fgets(temp,40,fptr);
return;
}

void display_record(int recnum, STUDRECORD *recptr)
{
    printf("\n %3d. ",recnum);
    printf("%3s ",(recptr->sex=='M'||recptr->sex=='m')?"Mr.":"Ms.");
    printf("%-12s",recptr->lastname);
    printf("%-12s",recptr->firstname);
    printf("  %5d",recptr->rollnum);
    printf("  %f",recptr->gpa);
    return;
}
```

A sample output from this program is shown below.

**UNSORTED RECORDS**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Rollnum</th>
<th>GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Mr. Rao</td>
<td>134</td>
<td>7.80000</td>
</tr>
<tr>
<td>1</td>
<td>Ms. Gupta</td>
<td>301</td>
<td>9.50000</td>
</tr>
<tr>
<td>2</td>
<td>Mr. Mahapatra</td>
<td>237</td>
<td>6.70000</td>
</tr>
<tr>
<td>3</td>
<td>Ms. Trivedi</td>
<td>201</td>
<td>8.90000</td>
</tr>
<tr>
<td>4</td>
<td>Mr. Pathak</td>
<td>177</td>
<td>8.30000</td>
</tr>
<tr>
<td>5</td>
<td>Ms. Nair</td>
<td>279</td>
<td>8.90000</td>
</tr>
</tbody>
</table>

**SORTED BY LASTNAME**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Rollnum</th>
<th>GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Ms. Gupta</td>
<td>301</td>
<td>9.50000</td>
</tr>
<tr>
<td>1</td>
<td>Mr. Mahapatra</td>
<td>237</td>
<td>6.70000</td>
</tr>
<tr>
<td>2</td>
<td>Ms. Nair</td>
<td>279</td>
<td>8.90000</td>
</tr>
<tr>
<td>3</td>
<td>Mr. Pathak</td>
<td>177</td>
<td>8.30000</td>
</tr>
<tr>
<td>4</td>
<td>Mr. Rao</td>
<td>134</td>
<td>7.80000</td>
</tr>
<tr>
<td>5</td>
<td>Ms. Trivedi</td>
<td>201</td>
<td>8.90000</td>
</tr>
</tbody>
</table>

**SORTED BY GPA**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Rollnum</th>
<th>GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Ms. Gupta</td>
<td>301</td>
<td>9.50000</td>
</tr>
<tr>
<td>1</td>
<td>Ms. Nair</td>
<td>279</td>
<td>8.90000</td>
</tr>
<tr>
<td>2</td>
<td>Ms. Trivedi</td>
<td>201</td>
<td>8.90000</td>
</tr>
<tr>
<td>3</td>
<td>Mr. Pathak</td>
<td>177</td>
<td>8.30000</td>
</tr>
<tr>
<td>4</td>
<td>Mr. Rao</td>
<td>134</td>
<td>7.80000</td>
</tr>
<tr>
<td>5</td>
<td>Mr. Mahapatra</td>
<td>237</td>
<td>6.70000</td>
</tr>
</tbody>
</table>

**SORTED BY ROLL NUMBER**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Rollnum</th>
<th>GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Mr. Rao</td>
<td>134</td>
<td>7.80000</td>
</tr>
<tr>
<td>1</td>
<td>Mr. Pathak</td>
<td>177</td>
<td>8.30000</td>
</tr>
<tr>
<td>2</td>
<td>Ms. Trivedi</td>
<td>201</td>
<td>8.90000</td>
</tr>
<tr>
<td>3</td>
<td>Mr. Mahapatra</td>
<td>237</td>
<td>6.70000</td>
</tr>
<tr>
<td>4</td>
<td>Ms. Nair</td>
<td>279</td>
<td>8.90000</td>
</tr>
<tr>
<td>5</td>
<td>Ms. Gupta</td>
<td>301</td>
<td>9.50000</td>
</tr>
</tbody>
</table>

**Key:** rollnum = 237 found at record # 3

3. Mr. Mahapatra , Vinay  237 6.700000
32.4 Points to Remember

Review Quiz

Programming Exercises
33.0 Lesson Goals

33.1 Direct Read/Write

In Chapter 11, we have seen how to perform formatted input/output operations on a file. These operations were carried out in a sequential manner in the file and each I/O operation was guided by a format specification.

Sometimes, we wish to directly access a block of data for reading or writing without bothering about the exact details of the data. For example, we might want to write the contents of an entire structure to a file without accessing the members. Two functions are available in the standard I/O library (STDIO.H) for unformatted input/output - fread and fwrite. They are meant for use only on binary files and should not be used for text files (ASCII files). fread has the following prototype.

```c
size_t fread(void *ptr, size_t size, size_t nobj, FILE *stream)
```

The first parameter is the address of the memory area where the input will be stored. The second parameter is the size of the object being read. The third parameter specifies the number of objects being read. The fourth and the last parameter is the name of the input stream which must be initialized in binary mode, i.e., using rb flag for access. For example,

```c
int myint;
fwrite(&myint, sizeof(int), 1, fileptr);
```

will read 1 integer of size given by sizeof(int) from the filestream fileptr into the variable myint. Similar operations can be carried out on structures.

```c
struct date
{
    unsigned short day;
```
unsigned short month;
unsigned int year;
} today, birthdays[MAX];
/* direct read for 1 structure */
fread(&today, sizeof(struct date), 1, fileptr);
/* direct read for an array of structures */
fread(birthdays, sizeof(struct date), MAX, fileptr);

The first call to fread reads the data for the struct today. The second call reads a block of data to fill the entire array of structures birthdays.

The fwrite function which has the following prototype can be used to directly write binary information to a filestream.

size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *stream)

How do we trace any errors that might occur during direct read/write operations? The function feof returns a TRUE value if an end-of-file is detected for the filestream. The function ferror returns a TRUE value if an error is detected for the filestream. They have the prototypes

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

Another function perror(userstr) prints the user specified string userstr along with an implementation dependent error message. This can provide an explanation of the error that has occurred.

### 33.2 Random Access Files

Even with the use of direct (unformatted) read/write operations, we are constrained to move sequentially along a file. When we need to move to any specified location in a file, we need to use the file positioning functions in the standard I/O library. Such access is also known as random access (a misnomer like RAM because the access is not to a random location in the file but to a specified location). Let us take a brief look at the various functions available to us.

    int fseek(FILE *fp, long offset, int origin)

This function locates us in a filestream based on the parameters offset and origin. The parameter can have a value of 0, 1, or 2 corresponding to beginning of file, current position, and end of file. Three predefined symbolic constants -- SEEK_SET, SEEK_CUR, and SEEK_END -- corresponding to 0, 1, and 2 are available for improving readability.

    long ftell(FILE *fp)

This function returns the current position in the file.

    int fgetpos(FILE *fp, fpos_t *ptr)
    int fsetpos(FILE *fp, fpos_t *ptr)
The type `fpos_t` is a predefined `typedef` suitable for storing file positions, particularly for text files. `fgetpos` stores the current position in `*ptr` for later use by the `fsetpos` function.

```c
void rewind(FILE *fp)
```

When we are not sure about our position after executing a random access operation, using the `rewind` function is a good idea. It positions the file to the beginning of the file and also clears any errors that may have occurred during file operations.

### 33.3 File Opening Modes

- `r` (open an existing file for reading)
- `w` (open a new file for writing)
- `a` (open an existing file for append)
- `r+` (open an existing file for update (reading + writing))
- `w+` (open a new file for update)
- `a+` (open an existing file for reading and appending)

The letter `b` placed after the first letter indicates the same mode for a binary file. For example, specifying `rb` opens an existing file for reading in binary mode.

### 33.4 Points to Remember

**Review Quiz**

**Programming Exercises**
34.1 Input Validation

Accepting incorrect input values is a major source of errors in any program. The input could take the form of interactive input keyed in by the user or values read from data files. Therefore, user input validation forms a major part of every robust program. In this section, we will look at some examples of input validation.

We first look at the problem of accepting an int value. We need to guard against the entry of any non-integral value. We can do this by reading the input into a string and checking for the existence of any illegal characters in this string. In many programs, we also need the input value to be in a certain range specified by a minimum value and a maximum value. The function get_int given below performs these tasks.

**** make full program

Program 34.1 – H A

```c
int get_int(int min, int max, char *prompt)
{
    #define FALSE 0
    #define TRUE 1
    char valid_symbs[]=" 0123456789-";
    char input[30];
    int input_ok,num;
    do 
    {
        input_ok = TRUE;
        printf("\n%s : ",prompt);
        gets(input);
        if(strspn(input,valid_symbs) != strlen(input))
        {
            input_ok=FALSE;
            ```
printf("\nIllegal character in input, try again");
}
sscanf(input,"%d",&num);
if(num < min || num > max)
{
    input_ok=FALSE;
    printf("\nValue out of range(%d-%d), try again",min,max);
}
while(input_ok == FALSE);
return(num);
}

In a similar fashion, the function get_float shown below performs input validation for a float value.

Program 34.2 – H

float get_float(char *prompt)
{
    #define FALSE 0
    #define TRUE 1
    char valid_symbs[]=" 0123456789-.eE");
    char input[30];
    int input_ok;
    float value;
    do
    {
        input_ok = TRUE;
        printf("\n%s : ",prompt);
        gets(input);
        if(strspn(input,valid_symbs) != strlen(input))
        {
            input_ok=FALSE;
            printf("\nIllegal character in input, try again");
        }
    }
    while(input_ok == FALSE);
    sscanf(input,"%f",&value);
    return(value);
}

The function get_a_char given below obtains a single character input from the user. You might feel tempted to simply use a call to getchar to obtain a single character but this creates a number of problems due to the buffered nature of input. For example, let us look at the following program segment for reading two characters into two variables cc1 and cc2.

cc1 = getchar();
cc2 = getchar();

But they will not work! Because the getchar function takes its input from the input buffer. When you enter the first character, say 'P', and press the ENTER key, cc1 is assigned a value of 'P' and cc2 is assigned a value of '
'. Where did the newline character come from? When you pressed the ENTER key, you entered a newline character into the input buffer as the next character after 'P' and this is what is read by the second call to getchar. Therefore, never try to read a single character
using `getchar`. Always read a string and extract the single character from the string as shown below in the function `get_a_char`. The function also performs the smart operation of ignoring any leading blank spaces in the input string. The function accepts a second argument which is a pointer to a string containing the valid characters acceptable for this input operation.

**Program 34.3 – H A**

```c
char get_a_char(char *prompt, const char *valid)
{
#define FALSE 0
#define TRUE 1
    char input[30];
    int input_ok, cc;
    do
    {
        input_ok = TRUE;
        printf("\n%s : ", prompt);
        gets(input);
        /* skip leading spaces */
        cc = input[strspn(input, " ")];
        printf("\ncc = %c", cc);
        if (strchr(valid, cc) == NULL)
        {
            input_ok = FALSE;
            printf("\nIllegal character, valid characters are \"%s\", valid);
        }
    } while (input_ok == FALSE);
    return (cc);
}
```

The following program uses the three functions given above to obtain validated input for an `int`, a `float` and a `char` variable.

**Program 34.4 – VALIDINP.C Validation of user input**

```c
/*  ============================= VALIDINP.C ============================
Performing validation of user input.
====================================================================*/
#include <stdio.h>
#include <string.h>
int get_int(int min, int max, char *prompt); 
float get_float(char *prompt);
char get_a_char(char *prompt, const char *valid);

int main(void)
{
    char userprompt[] = "Input roll number";
    char vprompt[] = "Input price of widget";
    char ans, valid_ans[] = "yYnN";
    char aprompt[] = "Enter Y for yes or N for no";
    int roll_num;
    float price;
    /* getting an integer value between 100 and 200 from user */
```
34.2 Using Non-Local Jumps

When you are deep in the nested structure of function calls, it becomes difficult to retrace your entire path back through the sequence of function calls when an error occurs. In such cases, we can perform a non-local jump using the `setjmp` and `longjmp` functions whose definitions are in the include file `SETJMP.H`. Their prototypes are given below.

```c
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

Here `jmp_buf` is a predefined data type for storing the current state of the program in a buffer. When the function `setjmp` is called, the various data regarding the state of the program are stored in the buffer `env`. Under normal execution `setjmp` returns a value of 0. At some nested call to a function, when the `longjmp` function is called, the program control jumps back to this `setjmp` call and the program state as recorded in `env` is restored. You can think of this as a `goto` executed from one function to a line of another program.

Program 34.5 – LONGJMP.C Using setjmp and longjmp library functions for error recovery

```c
/* ============================== LONGJMP.C ===========================
   Program showing use of setjmp and longjmp for error handling.
===================================================================*/
#include <stdio.h>
#include <setjmp.h>
define NMAX 3
float foobar(int p, int q);
float barfoo(int p, int q);
jmp_buf env; /* jump buffer to store state information */
main(void)
{
    int p,q,j;
    float quot;
    for(j=0;j<NMAX;j++)
    {
        printf("Enter integers to divide : ");
        scanf("%d %d",&p,&q);
        if(setjmp(env) == 0)
            printf("Regular execution");
            quot = foobar(p,q);
            printf("Quotient = %f",quot);
        else
            printf("Exception handled successfully!");
```
Trapping and Handling Errors

float foobar(int m, int n)
{
    return(barfoo(m,n));
}
float barfoo(int p, int q)
{
    if(q)
        return((float)p/q);
    else
    {
        printf("\nERROR: Denominator is 0");
        longjmp(env,2);
        return 0.0;
    }
}

A sample output from this program is shown below.

Enter integers to divide : 3 4
Regular execution
Quotient = 0.750000
Enter integers to divide : 3 0
Regular execution
ERROR: Denominator is 0
Exception handled successfully!
Enter integers to divide : 3 5
Regular execution
Quotient = 0.600000

34.3 Using the assert Macro

Using the assert macro defined in <assert.h> is another good method to trap errors. The macro takes a logical expression and checks whether it is TRUE. If TRUE execution proceeds as usual. If the expression evaluates to FALSE (i.e., the assertion fails), the program produces an error message and terminates. The program ASSERT.C demonstrates the use of this macro for error trapping.

Program 34.6 – ASSERT.C Use for assert macro to test assertions

```c
/* PROGRAM SHOWING USE OF ASSERT MACRO FOR ERROR TRAPPING. */
#include <stdio.h>
#include <assert.h>
main(void)
{
    int num, den;
    printf("\nEnter two integers to divide : ");
    scanf("%d %d", &num, &den);
    assert(den != 0);
    printf("\nEnter two integers to divide : ");
    scanf("%d %d", &num, &den);
    assert(den != 0);
    printf("\nEnter two integers to divide : ");
    scanf("%d %d", &num, &den);
    assert(den != 0);
    printf("\nEnter two integers to divide : ");
    scanf("%d %d", &num, &den);
    assert(den != 0);
    printf("\nEnter two integers to divide : ");
    scanf("%d %d", &num, &den);
    assert(den != 0);
```
printf("\nQuotient is %f",(float)num/den);
    return 1;
}

A sample output from this program is shown below.

Enter two integers to divide : 5 0
Assertion failed: den !=0, file ASSERT.C, line 11
Abnormal program termination

34.4 Points to Remember

Review Quiz

Programming Exercises
39

Bit Level Programming

35.0 Lesson Goals

35.1 The Bitwise Operators

All the operations that we have performed so far have been on variables stored in multiple number of bytes. The number of bytes for a variable depends on its data type and the implementation dependent sizes. But, in some problems, we need to access the data at the level of individual bits. One such area is access of operating system parameters which are usually coded into bit patterns. The second use of bitwise operations is for achieving data compression, i.e., storage of more data in fewer bytes. The availability of bit level operators is one of the strongpoints of the C language which gives it the flexibility required for system software development.

The bitwise operators are listed below.

\[
\begin{align*}
\& & \text{bitwise AND} \\
\mid & \text{bitwise OR} \\
\wedge & \text{bitwise EX-OR (exclusive OR)} \\
\sim & \text{bitwise NEGATION (complement)}
\end{align*}
\]

The first three are binary operators while the last one is a unary operator. The result of applying these operators to bits are shown in Table 34.1.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a &amp; b</th>
<th>a</th>
<th>b</th>
<th>a ^ b</th>
<th>~a</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The bitwise operators are applicable only to the signed and unsigned integral data types (i.e., char, short, int, and long). For better portability in using char variables, use either signed
char or unsigned char because char is implementation dependent. The program BITOPER.C demonstrates the use of these bitwise operators.

Program 35.1 – BITOPER.C Using the bitwise operators

/*  ============================ BITOPER.C ===========================
   Storage optimization using bit storage.
   ===================================================================*/
#include "bitfuncs.h"
main(void)
{
    unsigned char aa, bb, cc;
    unsigned int ax, bx;
    printf("\n Input aa and bb (between 0 and 255) >> ");
    scanf("%d %d",&ax,&bx);
    aa = ax;
    bb = bx;
    printf("\n      aa = %3d  =  %s",aa,CharToBitPat(aa));
    printf("\n      bb = %3d  =  %s",bb,CharToBitPat(bb));
    cc = aa & bb;
    printf("\n aa & bb = %3d  =  %s",cc,CharToBitPat(cc));
    cc = aa | bb;
    printf("\n aa | bb = %3d  =  %s",cc,CharToBitPat(cc));
    cc = aa ^ bb;
    printf("\n aa ^ bb = %3d  =  %s",cc,CharToBitPat(cc));
    cc = ~aa;
    printf("\n     ~aa = %3d  =  %s",cc,CharToBitPat(cc));
    cc = ~bb;
    printf("\n     ~bb = %3d  =  %s",cc,CharToBitPat(cc));
    return 1;
}

A sample output from this program is shown below. Perform the calculations of the bitwise operations by hand and convince yourself that the output is indeed what it should be. We have used a function CharToBitPat which is defined in Section 34.2.

Input aa and bb (between 0 and 255) >> 134 87
aa = 134  =  10000110
bb =  87  =  01010111
aa & bb =   6  =  00000110
aa | bb = 215  =  11010111
aa ^ bb = 209  =  11010001
~aa = 121  =  01111001
~bb = 168  =  10101000

35.2 Bitwise Shifting

There are two operators for performing bitwise left shift and right shift by a specified number of bits. The left shift operator is << and the right shift operator is >>. The syntax for the usage of these operators is to write

```
var >> nbit
```
where nbit is the number of bits by which we want to shift the bit pattern. For example, the expression (xx >> 3) will perform a 3 bit right shift on the bit pattern of xx. The empty bits formed by shifting are filled with 0 values. The program BITSHIFT.C given below demonstrates the results of using the shift operators.

**Program 35.2 – BITSHIFT.C Using the bitwise shift operators**

```c
/*  ==============================================================
 *  BITSHIFT.C  ====================================================
 *  Trying out bitwise shift operations.                          
 *  ==============================================================*/
#include "bitfuncs.h"
main(void)
{
    unsigned char aa, bb;
    unsigned int ax, j;
    printf("\nInput aa (between 0 and 255) >> ");
    scanf("%d",&ax);
    aa = ax;
    printf("\n      aa  =  %s",CharToBitPat(aa));
    printf("\nBitwise Right Shift\n");
    for(j=1;j<=8;j++)
    {
        bb = aa >> j;
        printf("\n      aa >> %1d  =  %s",j,CharToBitPat(bb));
    }
    printf("\n      aa  =  %s",CharToBitPat(aa));
    printf("\nBitwise Left Shift\n");
    for(j=1;j<=8;j++)
    {
        bb = aa << j;
        printf("\n      aa << %1d  =  %s",j,CharToBitPat(bb));
    }
    return 1;
}
```

A sample output from this program is shown below. Notice that a left shift or a right shift of more than 7 bits results in a uniform result of 0 for an unsigned char variable.

**Input aa (between 0 and 255) >> 1**

aa = 10111011

**Bitwise Right Shift**

- aa >> 1 = 01011101
- aa >> 2 = 00101110
- aa >> 3 = 00010111
- aa >> 4 = 00001011
- aa >> 5 = 00000101
- aa >> 6 = 00000010
- aa >> 7 = 00000001
- aa >> 8 = 00000000

**Bitwise Left Shift**

- aa << 1 = 01101101
- aa << 2 = 11101100
- aa << 3 = 11110000
- aa << 4 = 10110000
We will now look at two functions - the first converts an unsigned char to its equivalent binary representation stored in a string. The second does the reverse, i.e., it takes a string representation of an 8 bit binary value and converts it to the equivalent unsigned char value.

*** SHIFT TO DIVIDE BY 2, 4, 8

** make this a full program

Program 35.3 – BITFUNCS.C  
User defined functions for bit patterns

/ *  ===============================================================  BITFUNCS.C  ================================================================  
Functions for converting to and from bit patterns.  
===================================================================*/
#include "bitfuncs.h"

char * CharToBitPat(unsigned char c)
{
    char *s,j;
    unsigned char mask=128;
    s = (char *)malloc(9);
    for(j=0;j<8;j++)
    {
        s[j] = (c & mask) ? '1' : '0';
        mask >>= 1; /*right shift of 1 bit in mask*/
    }
    s[8]='\0';
    return(s);
}

unsigned char BitPatToChar(char *s)
{
    unsigned char j,temp=0;
    if(strlen(s) > 8)
    {  
        printf("\nFATAL ERROR : Bit pattern exceeds 8 bits\n");
        exit(0);
    }
    for(j=0;j<8;j++)
    {
        if(s[j] > '1' || s[j] < '0')
            printf("\nFATAL ERROR : Illegal bit value %c",s[j]);
        exit(0);
    }
    temp *=2;
    temp += s[j]-'0';
    return(temp);
}
35.3 Storage Optimization

We can use bitwise operators to squeeze our memory requirements for a program. Let us take an example where we want to store eight boolean values. Ordinarily, we would use 8 char variables to store these values. But fundamentally speaking, a boolean value requires only 1 single bit to store either TRUE (1) or FALSE (0). Using bit operations, it is possible to store all 8 boolean values in a single 8 bit variable. In the following program, we use the 8 bits of the variable flags to store eight different boolean values and we see how we can manipulate these values using the bitwise operators.

Program 35.4 – BITSTOR1.C Storage optimization using bits for storage

```c
/*  ============================ BITSTOR1.C ===========================
Storage optimization using bit storage.
===================================================================*/
#include "bitfuncs.h"
#define BIT0 1
#define BIT1 2
#define BIT2 4
#define BIT3 8
#define BIT4 16
#define BIT5 32
#define BIT6 64
#define BIT7 128
#define SOMEVALUE 139

main(void)
{
    unsigned char flags=SOMEVALUE;
    printf("\nExtracting bit values\n\nBit 0  = %1d", (flags & BIT0)? 1 : 0);
    printf("Bit 1  = %1d", (flags & BIT1)? 1 : 0);
    printf("Bit 2  = %1d", (flags & BIT2)? 1 : 0);
    printf("Bit 3  = %1d", (flags & BIT3)? 1 : 0);
    printf("Setting bit 2 to 1 (bit ON)\n");
    flags |= BIT2;
    printf("New bit pattern is %s",CharToBitPat(flags));
    printf("Setting bit 0 to 0 (bit OFF)\n");
    flags &= ~BIT0;
    printf("New bit pattern is %s",CharToBitPat(flags));
    printf("Toggling bit 5 and bit 2\n");
    flags ^= BIT5;
    flags ^= BIT2;
    printf("New bit pattern is %s",CharToBitPat(flags));
    return 1;
}
```
There are several interesting things to note. Firstly, notice the selection of the symbolic constants `BIT0` through `BIT7` and the values assigned to them. These values ensure that only one bit has a value of 1 and the rest are all 0s. For example, in `BIT3`, only bit number 3 is 1 and all other bits are 0.

Next, look at the operations required to manipulate the bits as repeated below.

```c
flags |= BIT6;
flags &= ~BIT6;
```

The first operation changes bit 6 to 1. The second changes it to 0. This can be stated in the following identities.

\[
\begin{align*}
A \lor 1 &= 1 & \text{setting a bit} \\
A \lor 0 &= A & \text{no change to bit} \\
A \land 0 &= 0 & \text{resetting a bit} \\
A \land 1 &= A & \text{no change to bit}
\end{align*}
\]

The following statement toggles the value of bit 6, i.e., if the old value was 0, it is changed to 1 and if the old value was 1, it is changed to 0.

```c
flags ^= BIT6;
```

This toggling behavior of the exclusive OR operator can be seen in the following identity.

\[
\begin{align*}
A \oplus 1 &= \neg A & \text{toggling a bit} \\
A \oplus 0 &= A & \text{no change to bit}
\end{align*}
\]

A sample output from the program `BITSTOR1.C` is shown below.

```
Bit pattern for 139 is 10001011
Extracting bit values
Bit 0 = 1
Bit 1 = 1
Bit 2 = 0
Bit 3 = 1
Setting bit 2 to 1 (bit ON)
New bit pattern is 10001111
Setting bit 0 to 0 (bit OFF)
New bit pattern is 10001110
Toggling bit 5 and bit 2
New bit pattern is 10101010
```

We have seen how to store eight boolean values in a single byte. But at what price do we achieve this saving in memory requirement? The price is paid in terms of the additional time required to perform the bitwise operations for retrieval and storage of data into these bits.

You might have noticed that the use of bitwise operators in rather clumsy looking code. In the next section, we look at a more elegant implementation of the same idea using bit fields.

### 35.4 Bit Fields
There is a special class of structs which can hold variables in fractional bytes. The members of these structs are known as bit fields and each has a specified width in bits. Every member must be either an unsigned int or a signed int.

```c
struct sdate
{
    unsigned int num1 : 4;
    unsigned int num2 : 3;
};
```

In the example shown above, we have declared a bit field of 4 bits named `num1` and a bit field of 3 bits named `num2`. It should be obvious that `num1` can store values lying between 0 and 15 and `num2` can store values between 0 and 7.

The implementation of fields is highly implementation dependent. Unnamed fields with no name but specified width are allowed. A field width of 0 forces alignment to the next word in memory. Fields are not arrays. The address operator (&) cannot be applied to fields. Access to the bit fields uses the usual struct member or dot (.) operator. The following program BITSTOR2.C shows how we can store eight boolean values in a single struct named `flags`.

**Program 35.5 – BITSTOR2.C Storage using bit fields**

```c
/*  ============================ BITSTOR2.C ===========================
Storage optimization using bit fields.
===================================================================*/
#include "bitfuncs.h"

struct bitstore
{
    unsigned int bit0 : 1;
    unsigned int bit1 : 1;
    unsigned int bit2 : 1;
    unsigned int bit3 : 1;
    unsigned int bit4 : 1;
    unsigned int bit5 : 1;
    unsigned int bit6 : 1;
    unsigned int bit7 : 1;
};

void display(struct bitstore);

main(void)
{
    struct bitstore flags={1,1,0,0,0,0,0,1};
    printf("\nExtracting bit values\n");
    printf("\nBit 0  = %1d",flags.bit0);
    printf("\nBit 1  = %1d",flags.bit1);
    printf("\nBit 2  = %1d",flags.bit2);
    printf("\nBit 3  = %1d",flags.bit3);
    display(flags);
    printf("\n Setting bit 2 to 1 (bit ON)\n");
    flags.bit2 = 1;
    display(flags);
    printf("\n Setting bit 0 to 0 (bit OFF)\n");
    flags.bit0 = 0;
    display(flags);
    printf("\n Toggling bit 5 and bit 2\n");
    display(flags);
}
```
if(flags.bit5)
    flags.bit5 = 0;
else
    flags.bit5 = 1;
if(flags.bit2)
    flags.bit2 = 0;
else
    flags.bit2 = 1;
display(flags);
return 1;
}

void display(struct bitstore b)
{
    printf("\n Bit Pattern is ");
    printf("%1d%1d%1d%1d",b.bit7,b.bit6,b.bit5,b.bit4);
    printf("%1d%1d%1d%1d",b.bit3,b.bit2,b.bit1,b.bit0);
    return;
}

The output from the above program is shown below.

Extracting bit values
Bit 0  = 1
Bit 1  = 1
Bit 2  = 0
Bit 3  = 1
Bit Pattern is 10001011
Setting bit 2 to 1 (bit ON)
Bit Pattern is 10001111
Setting bit 0 to 0 (bit OFF)
Bit Pattern is 10001110
Toggling bit 5 and bit 2
Bit Pattern is 10101010

35.*  Points to Remember

Review Quiz

Programming Exercises
36.0 Lesson Goals

36.1 Good Programming Style

The C language is a very flexible language and offers many different ways to code the same thing. Each of these codes might achieve the same result but it is possible to differentiate between them on the aesthetic grounds of "programming style". Developing a neat and consistent programming style is very essential for a good programmer. In this chapter, we will look at some aspects of programming style. The more serious reader is directed to the excellent book by Steve Oualline - "C Elements of Style".

A good programming style aims at improving the following aspects of a program:

- **Readability** - Your programs will be read by other programmers. For this, it is essential to make your programs more readable.
- **Maintainability** - Your programs will need maintenance in the future. This might take the form of bug fixes or it might be in the form of extensions to its functionality. Your program should be designed to make maintenance easy.
- **Portability** - Your program may have to be transported to a different environment (i.e., a different compiler on a different operating system). Your program should have built in features to make such porting easy.
- **Usability** - The end user needs a "user-friendly" program to work with.
- **Efficiency** - You want to make your program efficient in terms of memory usage and increased speed of execution.

36.2 Readability and Maintainability

Always remember that your C programs must communicate their logic and meaning to another programmer. If a program is not very readable, you yourself may fail to understand it after a few
years. Always think: "Can I understand this program if someone else had written it and given it to me?" The following are some points to observe in making programs more readable and maintainable.

**Whitespace**
- Use blank lines to break up your programs into "paragraphs".
- Use whitespace to improve the appearance of the program.
- Use spaces before and after every operator to improve readability.

**Comments**
- Provide sufficient comments in your programs.
- Write the comments while writing the code. Do not leave it for a later time.
- Comments do not add to the executable program size because they are ignored by the compiler.
- Begin each program file with brief prefatory comments.

**Variable and Function Names**
- Never use the names of C library functions as variable or function names.
- Never use `argc` or `argv` as variable names.
- Never use the characters 'l' (lowercase L) as it can easily be confused for the digit '1' (one).
- Never use the characters 'O' (uppercase O) as it can easily be confused for the digit '0' (zero).
- Never reuse the name of an external variable for an internal variable even though it is allowed in C.
- Choose good descriptive names for variables.
- When combining many words to form a name, you can either separate them with an underscore (e.g., `get_one_char`, `print_matrix`) or you can write the first character of each word in uppercase (e.g., `GetOneChar`, `PrintMatrix`).
- Use some convention to mark pointer variables, e.g., you could use a suffix `_ptr` or `Ptr` for all pointer variables.

**Constants**
- Write floating point constants with at least one digit on either side of the decimal point, e.g., 6.0, 0.7, 340.0, etc. Do not write .7, 6., etc., because the decimal point becomes inconspicuous and might be missed out by the reader.
- Use lowercase 'e' for the exponent as in 45.6e77.
- Use uppercase A through F for hexadecimal constants because lowercase constants may be confused for identifier names.
- To indicate long constants use uppercase L and never the lowercase l (it could be confused for the digit 1).
Statements

- Prefer short statements to long statements.
- Put one statement on one line as far as possible.
- If putting two or more short statements together improves readability, put them on a single line.
- Use a consistent indentation style for your code. You can choose between either the Allman style used in this book or the K&R style used in Kernighan and Ritchie.
- Usual indentation is four spaces for each level.
- Avoid using `++` and `--` operators inside another statement. Put them as separate statements. This makes it easier to interpret the code.
- Avoid using assignment statements inside another expression.
- Mark any null statement with a short descriptive comment.
- Use one `printf` statement for each line of output.
- Every `switch` statement must have a `default` case.
- Avoid using `goto` as far as possible.
- When writing complex expressions, make the precedence explicit by using parentheses. Do not force the reader to look up operator precedence tables.
- Avoid heavily nested control structures or expressions.

External Variables

- Minimize the use of external variables.
- Declare all external variables in a header file.

Functions

- Avoid "old style" ("K & R style") function declarations. Use the ANSI style declarations.
- Give an explicit return type for every function.
- If a function has no parameters, the parameter list must be given as `(void)`. Remember that `()` indicates an unspecified parameter list and not a null parameter list.
- Put a `return` statement in every function.
- Keep functions simple. A function should perform only one task.
- Keep functions short. The listing of a function should be around one or two pages long. Break up longer functions into a set of smaller functions.

Preprocessor Directives

- Use uppercase names for symbolic constants.
- Never redefine any of the C keywords or names of standard library functions.
- Check for unwanted side effects of argumented macros.
• Enclose arguments of argumented macros in parentheses.
• Place all \#include statements at the beginning of the file after the prefatory comments.
• Use conditional compilation to exclude code from execution. The most common use of this for the debugging code.

36.3 Improving Portability

An important consideration is to allow for a future port of your program to another C compiler working on another operating system. The following points must be kept in mind for this.

Characters
• Use only the standard set of ASCII characters from 33 to 127 in your programs.
• Use of the extended IBM character set is not portable.
• Do not assume any character equivalent for the '\n' escape sequence. On some machines, it is changed to ASCII character 10 (linefeed), on some machines it is converted to ASCII code 13 (carriage return), and on some others it is replaced by a combination of both the linefeed and the carriage return characters.

Data Types
• If the choice of a data type might have to change from one machine to another, use a typedef to defined an alias for this data type.
• Never assume a size for any data type (except for char which uses one byte). The sizes are implementation dependent and must be obtained using the sizeof operator or from the symbolic constants defined in the two header files, LIMITS.H and FLOAT.H.
• A char may be signed or unsigned depending on the implementation. Specify signed char or unsigned char for better portability.

Line Length
• Use a maximum of 72 characters per line as some monitors support only this length.

Files
• Include files and data files must reside in the same subdirectory as the program files.
• Avoid giving absolute path specifications for any file as these may not be portable.

Functions
• If you use any non-ANSI function from any compiler's non-standard library, encapsulate it in a user defined function and document its use using clear comments.
• Enclose any non-standard function call in a conditional compilation clause to prevent erroneous compilation on other compilers.
36.4 Usability

Creating a user friendly program is one of the obvious goals of any good program. The following points must be kept in mind for improving usability.

**Predictable Behavior**
- The program should not behave in an unusual or unexpected manner.

**User Input**
- User input (interactive input from keyboard or data input from a file) must be validated before being accepted. The program should be able to reject garbage input values. A smart program should also have other validation methods like range checking built into it.
- Avoid asking the user for unnecessary "mickey mouse" input. The user should never be asked to input anything which can be computed from the available data.
- The user should not be asked to memorize anything during the execution of a program. The user prompts should be designed with this rule in mind.
- Whenever possible, use end-of-data indicators (i.e., "sentinel values"), instead of asking the user to input "number of data items".

**Errors and Warnings**
- Error messages must be marked as errors.
- Warnings must be marked as warnings.
- Error and warning messages must be simple and easy to understand.

36.5 Efficiency Considerations

Once our program begins to function properly, i.e., starts performing according to our requirements, we can begin looking into efficiency considerations. We can try to improve speed of execution and efficiency of memory usage by observing the following points.

- Integer arithmetic is much faster than floating point arithmetic.
- Check to see if a `char` variable will be enough to store an integer value. Small integer values require less memory if stored in `char` variables.
- Using pointers to sequentially access array elements is faster than using indexed access.
- A compound assignment operator improves execution speed. Therefore, `x += y` will execute faster than `x = x + y`.
- The increment and decrement operators are faster than the equivalent assignment expressions. Therefore, `j++` will execute faster than `j = j + 1`.
- Use of multidimensional arrays makes the program slow.
• Input/output from storage media (e.g., hard disk, floppy diskette, magnetic tape, etc.) makes the program slow and, therefore, such operations should be minimized.

• If a computation has to be repeated more than once, store the result of the first computation in a new variable and eliminate the subsequent computations.

• Use dynamic memory allocation instead of static memory allocation for large arrays and structures to save on memory.

### 36.6 Points to Remember

#### Review Quiz

#### Programming Exercises
References


