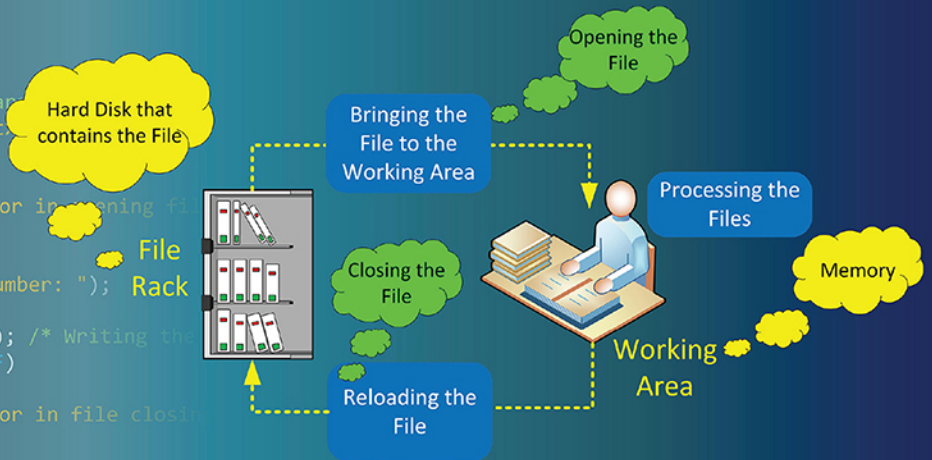


C PROGRAMMING

Learn to Code

Sisir Kumar Jena

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
    int x;
    FILE *fp; /* Declaration of file pointer */
    fp=fopen("Sample.txt","r");
    if(fp==NULL)
    {
        printf("Error in opening file\n");
        exit(0);
    }
    printf("Enter a number: ");
    scanf("%d",&x);
    fprintf(fp,"%d",x); /* Writing the data to file */
    if(fclose(fp)==EOF)
    {
        printf("Error in file closing\n");
        exit(1);
    }
}
```



CRC Press
Taylor & Francis Group

A CHAPMAN & HALL BOOK

C Programming



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

C Programming

Learn to Code

Sisir Kumar Jena



CRC Press

Taylor & Francis Group

Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

First edition published 2022

by CRC Press

6000 Broken Sound Parkway NW, Suite 300, Boca Raton, FL 33487-2742

and by CRC Press

2 Park Square, Milton Park, Abingdon, Oxon, OX14 4RN

CRC Press is an imprint of Taylor & Francis Group, LLC

© 2022 Sisir Kumar Jena

Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, access www.copyright.com or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. For works that are not available on CCC please contact mpkbookspermissions@tandf.co.uk

Trademark notice: Product or corporate names may be trademarks or registered trademarks and are used only for identification and explanation without intent to infringe.

Library of Congress Cataloging-in-Publication Data

Names: Jena, Sisir Kumar, author.

Title: C programming : learn to code / Sisir Kumar Jena.

Description: First edition. | Boca Raton, FL : Chapman & Hall/CRC Press, [2022] | Includes bibliographical references and index. | Summary: "The C programming language is a popular language in industries as well as academics. Since its invention and standardized as ANSI C, several other standards known as C99, C11, and C17 were published with new features in subsequent years. This book covers all the traits of ANSI C and includes new features present in other standards. The content of this book helps a beginner to learn the fundamental concept of the C language. The book contains a step-by-step explanation of every program that allows a learner to understand the syntax and builds a foundation to write similar programs. Besides, exercises and illustrations present in this book make it a complete textbook in all aspects"-- Provided by publisher.

Identifiers: LCCN 2021027981 (print) | LCCN 2021027982 (ebook) | ISBN 9781032036250 (hbk)

| ISBN 9781032036274 (pbk) | ISBN 9781003188254 (ebk)

Subjects: LCSH: C (Computer program language) | Computer programming.

Classification: LCC QA76.73.C15 J46 2022 (print) | LCC QA76.73.C15

(ebook) | DDC 005.13/3--dc23

LC record available at <https://lcn.loc.gov/2021027981>

LC ebook record available at <https://lcn.loc.gov/2021027982>

ISBN: 978-1-032-03625-0 (hbk)

ISBN: 978-1-032-03627-4 (pbk)

ISBN: 978-1-003-18825-4 (ebk)

DOI: 10.1201/9781003188254

Typeset in Palatino

by SPi Technologies India Pvt Ltd (Straive)

In memory of my father, my guide, Baikuntha Nath Jena



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Contents

Preface.....	xvii
Acknowledgments.....	xix
Organization of this Book.....	xxi
Author Biography.....	xxiii
1. Introduction to the Computer	1
1.1 Introduction	1
1.2 Definition and Characteristics of a Computer System	1
1.3 History of the Computer.....	2
1.4 Basic Computer Organization.....	4
1.4.1 Input Devices.....	5
1.4.2 Memory	6
1.4.3 Central Processing Unit	6
1.4.4 Output Devices.....	7
1.5 Computer Memory	7
1.5.1 Registers	8
1.5.2 Cache Memory	8
1.5.3 Primary Memory.....	8
1.5.4 Random Access Memory	10
1.5.5 Read Only Memory	10
1.5.6 Secondary Memory.....	10
1.5.7 Hard Disk Drive.....	10
1.5.8 Solid State Drive.....	11
1.6 Introduction to the Operating System	12
1.6.1 Hardware and Software.....	12
1.6.2 Operating System.....	12
1.6.3 Functions of an Operating System	13
1.7 Review Questions	14
1.7.1 Long Answers.....	14
1.7.2 Short Answers	14
1.7.3 Practical Exercises.....	15
References	16
2. Number Systems.....	17
2.1 Introduction	17
2.1.1 Non-positional Number System.....	17
2.1.2 Positional Number System	17
2.2 Positional Number Systems	18
2.2.1 Decimal Number System.....	18
2.2.2 Binary Number System.....	20
2.2.3 Hexadecimal Number System	21
2.2.4 Octal Number System	22

2.3	Number Conversion	23
2.3.1	Binary to Decimal	23
2.3.1.1	Approach 1	24
2.3.1.2	Approach 2	24
2.3.2	Binary Fraction to Decimal Conversion	25
2.3.3	Binary to Decimal Conversion	26
2.3.4	Decimal Fraction to Binary Fraction	27
2.3.5	Decimal to Octal Conversion	28
2.3.6	Octal to Decimal Conversion	29
2.3.7	Octal to Binary Conversion	29
2.3.7.1	Procedure 1	29
2.3.7.2	Procedure 2	30
2.3.8	Binary to Octal Conversion	30
2.3.8.1	Procedure 1	30
2.3.8.2	Procedure 2	31
2.3.9	Decimal to Hexadecimal Conversion	32
2.3.10	Hexadecimal to Decimal Conversion	33
2.3.11	Hexadecimal to Binary Conversion	33
2.3.11.1	Procedure 1	33
2.3.11.2	Procedure 2	34
2.3.12	Binary to Hexadecimal Conversion	34
2.3.12.1	Procedure 1	34
2.3.12.2	Procedure 2	35
2.4	Review Questions	36
2.4.1	Conversion Questions	36
3.	Problem Solving through Flowcharts and Algorithms	39
3.1	Introduction	39
3.2	Problem-solving Approach	40
3.3	Algorithm Design	41
3.3.1	Characteristics of an Algorithm	42
3.4	Basics of an Algorithm	43
3.4.1	Advantages of Using an Algorithm	44
3.4.2	Example: Write an Algorithm to Add Two Numbers and Produce the Sum	44
3.4.3	Algorithm 3.1	45
3.5	Flowcharts	45
3.5.1	Advantages of Using a Flowchart	45
3.5.2	Flowchart Symbols	46
3.5.3	Flowchart Drawing Guidelines	46
3.6	Example Problems	48
3.7	Basics of a Programming Language	53
3.7.1	Low-level Languages	53
3.7.1.1	Machine-level Languages	54
3.7.1.2	Assembly-level Languages	54
3.7.2	High-level Languages	55
3.7.2.1	Compiler vs. Interpreter	57
3.7.2.2	Advantages	57

3.8	Review Questions	57
3.8.1	Objective Type Questions	57
3.8.2	Practice Problems.....	57
3.8.3	Subjective Questions.....	58
	Reference	59
4.	Introduction to C Programming.....	61
4.1	Introduction	61
4.2	History of C.....	62
4.3	Executing a C Program.....	64
4.3.1	Editing	64
4.3.2	Compiling	64
4.3.3	Linking	65
4.3.4	Executing.....	65
4.4	Structure of a C Program	65
4.4.1	Documentation.....	65
4.4.2	Header Files	65
4.4.3	Global Variables	66
4.4.4	main() Function	66
4.4.5	Subprograms	67
4.4.6	Your First C Program.....	67
4.5	Compilers and Editors for Executing C Programs.....	69
4.5.1	Editors.....	69
4.5.2	Compilers.....	69
4.5.3	Executing Your First C Program	71
4.5.3.1	Mac	71
4.5.3.2	Windows.....	72
4.5.3.3	Linux	72
4.6	Review Questions	73
4.6.1	Objective Questions.....	73
4.6.2	Short Answer Questions	73
4.6.3	Programming Questions.....	73
4.6.4	Long Questions	75
	References	75
5.	Constants, Variables, and Data Types.....	77
5.1	Introduction	77
5.2	C Character Sets	77
5.3	Keywords	78
5.4	Variables and Identifiers	79
5.5	Data Types.....	80
5.5.1	Primary Data Types	81
5.5.2	Integer Data Types	81
5.5.3	Floating Point Types	82
5.5.4	Character Data Types	83
5.5.5	Void Types.....	83
5.6	Declaration of Variables	84
5.7	Constants.....	86
5.7.1	Integer Constants.....	86

5.7.2	Real Constants.....	87
5.7.3	Fractional Form.....	87
5.7.4	Exponential Form.....	87
5.7.5	Character Constants.....	87
5.7.6	String Constants.....	87
5.8	Learn to Code Examples.....	88
5.9	Escape Sequences.....	91
5.10	Review Questions.....	92
5.10.1	Objective Questions.....	92
5.10.2	Programming Questions.....	92
5.10.3	Subjective Questions.....	94
6.	Operators and Expressions.....	95
6.1	Introduction.....	95
6.2	Arithmetic Operators.....	96
6.3	Relational Operators.....	97
6.4	Assignment Operators.....	98
6.5	Logical Operators.....	99
6.6	Increment and Decrement Operators.....	100
6.7	Conditional Operators.....	103
6.7.1	Nested Conditional Operators.....	105
6.8	Bitwise Operators.....	105
6.8.1	Bitwise AND, OR, XOR.....	106
6.8.2	One's Complement (~) Operator.....	107
6.8.3	Two's Complement Representation.....	107
6.8.4	Left Shift Operator (<<) and Right Shift Operator (>>).....	109
6.9	Special Operators.....	112
6.9.1	The Comma Operator.....	112
6.9.2	The sizeof Operator.....	113
6.10	Expressions.....	113
6.10.1	Evaluation of Expressions.....	114
6.10.2	Rules for Evaluation of Expressions.....	114
6.11	Type Conversion.....	115
6.11.1	Implicit Type Casting.....	115
6.11.2	Explicit Type Conversion.....	116
6.12	Operator Precedence and Associativity.....	116
6.13	Review Questions.....	118
6.13.1	Objective Type Questions.....	118
6.13.2	Programming Questions.....	119
6.13.3	Subjective Type Questions.....	120
7.	Basic Input/Output.....	123
7.1	Introduction.....	123
7.2	Unformatted Functions.....	124
7.2.1	getchar() and putchar().....	124
7.2.2	gets() and puts().....	125
7.2.3	getch() and getche().....	126
7.2.4	putch().....	127

7.3	Formatted Functions.....	128
7.3.1	printf() Function.....	128
7.3.2	Formatting with printf()	129
7.3.3	scanf() Function	134
7.3.4	Formatting with scanf	134
7.4	Review Questions	137
7.4.1	Short Answer Questions	137
7.4.2	Programming Questions.....	138
7.4.3	Subjective Questions.....	139
8.	Control Structures.....	141
8.1	Introduction	141
8.2	Selection with if Statements.....	143
8.2.1	Some Points to Remember.....	145
8.3	if-else Statement	146
8.3.1	Write a Program to Check Whether a Number Entered by the User is Zero or Nonzero	148
8.3.2	Write a Program to Calculate the Travel Fare of a Person	149
8.4	Nested if-else Statements	150
8.4.1	Write a Program to Find the Biggest Among Three Numbers	151
8.5	if-else-if Ladders.....	151
8.5.1	Write a Program to Perform as a Four-Function Calculator.....	152
8.6	Compound Statements.....	154
8.7	Multiway Selection with Switch Statements.....	155
8.7.1	Some Points to Remember.....	157
8.8	goto Statement.....	159
8.8.1	Notes on goto.....	159
8.9	Introduction to Loops.....	160
8.10	while Loops.....	161
8.11	do-while Loops.....	164
8.11.1	Difference Between while and do-while Loops.....	166
8.12	for Loops.....	167
8.12.1	Some Solved Problems (Printing Patterns)	171
8.13	Unconditional Branching: Break and Continue.....	173
8.13.1	break Statements	173
8.13.2	continue Statements.....	174
8.14	Review Questions	176
8.14.1	Short Questions	176
8.14.2	Long Questions	176
9.	Functions	179
9.1	Introduction	179
9.2	The Need for Functions.....	181
9.3	Types of Function.....	182
9.4	User-defined Functions	182
9.5	Components and Working of a Function	186
9.5.1	Calling Function.....	186
9.5.2	Called Function.....	186
9.5.3	Function Prototype	187

9.5.4	Function Definition.....	187
9.5.5	Function Call	188
9.5.6	Actual Arguments.....	188
9.5.7	Formal Arguments.....	188
9.5.8	Return Type	188
9.6	Categories of a Function	191
9.6.1	A Function Without Arguments and Without Return Types.....	191
9.6.2	A Function Without Arguments and With Return Types	191
9.6.3	A Function With Arguments and Without Return Types	193
9.6.4	A Function With Arguments and With Return Types	193
9.7	Recursion.....	195
9.7.1	Example: Find the Value of x^y	198
9.7.2	Programming Examples	201
9.8	Storage Classes	204
9.8.1	Automatic Storage Class.....	205
9.8.2	Register Storage Class	206
9.8.3	Static Storage Class.....	206
9.8.4	External Storage Class.....	207
9.9	Review Question	209
9.9.1	Objective Questions.....	209
9.9.2	Subjective Questions.....	210
9.9.3	Programming Questions.....	210
10.	Arrays and Strings	213
10.1	Introduction	213
10.2	Need for Arrays.....	214
10.3	Types of Arrays.....	214
10.4	1D Arrays	215
10.4.1	Declaration of 1D Arrays	215
10.4.2	Initialization of Arrays	216
10.4.3	Accessing Array Elements.....	217
10.4.4	Characteristics of an Array	218
10.4.5	Entering Data in an Array	219
10.4.6	Displaying the Content of an Array.....	220
10.4.7	Programming Examples	221
10.4.7.1	Write a Program to Create an Array of N Elements and Write the Code to Find the Biggest Number and the Smallest Number Present in the Array.....	221
10.4.7.2	Write a Program to Search for an Element Present in the Array, the Number of Times the Element is Present, and Print the Element's Positions	222
10.4.7.3	Write a Program to Print the Binary Equivalent of a Decimal Number Using an Array.....	223
10.4.8	Points to Note.....	224
10.5	2D Arrays	225
10.5.1	Introducing Matrices	225
10.5.2	Declaration of a 2D Array.....	226
10.5.3	Representation of a 2D Array in Memory	226
10.5.3.1	Row Major Order	227

10.5.3.2	Column Major Order	227
10.5.4	Initialization of 2D Array.....	228
10.5.5	Accessing the Elements of a 2D Array.....	229
10.5.6	Entering Data in a 2D Array.....	231
10.5.7	Exploration of a 2D Matrix.....	234
10.5.8	Programming Examples	235
10.5.8.1	Write a Program to Add All the Elements Present in the Main Diagonal of a 2D Matrix.....	235
10.5.8.2	Write a Program to Add the Elements of Each Column and Print it in the Following Format.....	236
10.5.8.3	Write a Program to Add Two Matrices	238
10.5.8.4	Write a Program to Multiply Two Matrices.....	240
10.6	Multidimensional Arrays.....	242
10.6.1	Declaration and Representation of 3D Arrays.....	242
10.6.1.1	Write a Program to Declare a 3D Array, Input Some Numbers, and Display the 3D Array	244
10.7	Character Arrays: Strings.....	245
10.7.1	Declaration of a String.....	245
10.7.2	Initialization of a String.....	245
10.7.3	Reading a String.....	246
10.7.3.1	Disadvantages of the <code>scanf()</code> Function.....	246
10.7.3.2	Reading Strings with the <code>gets()</code> Function	247
10.7.4	Displaying the String.....	247
10.7.5	Programming Examples	249
10.7.5.1	Find the Length of a String	249
10.7.5.2	Count the Number of Words Present in a String.....	249
10.7.5.3	Reverse the String.....	250
10.7.5.4	Check Whether the String is a Palindrome or Not.....	251
10.8	String Functions	252
10.8.1	<code>strcpy</code> (Destination, Source)	253
10.8.2	<code>strcat</code> (Destination, Source).....	253
10.8.3	<code>strcmp</code> (First, Second).....	253
10.8.4	Programming Examples Using String Functions.....	254
10.9	Review Questions	255
10.9.1	Objective Questions.....	255
10.9.2	Subjective Questions.....	255
10.9.3	Programming Exercises	256
11.	Pointers	259
11.1	Introduction	259
11.2	Basic Knowledge	260
11.3	Pointer Variables	261
11.3.1	Declaration of Pointer Variables	261
11.3.2	Working with Pointers	261
11.3.3	Workout.....	263
11.4	Pointer to Pointer (Double Pointer).....	265
11.5	Void Pointers.....	266
11.6	Null Pointers.....	268
11.6.1	What is the Meaning of NULL?.....	268

11.7	Constant Pointers	268
11.7.1	Pointers to Constants.....	272
11.8	Pointer Arithmetic.....	272
11.9	Pointers and Functions.....	276
11.9.1	Pass by Value	276
11.9.2	Pass by Reference or Address	277
11.9.2.1	Problem: Write a Program to Swap Two Numbers Using Functions.....	280
11.10	Pointers and Arrays	282
11.11	Passing Arrays to Functions	285
11.11.1	Write a Program to Pass an Array to a Function and Find the Largest and Smallest Numbers Present in that Array	290
11.12	Pointers and 2D Arrays	291
11.13	Pointers and Strings.....	293
11.13.1	Passing a String to a Function.....	294
11.13.2	Write a Program to Reverse a String Using a Function.....	294
11.14	An Array of Pointers.....	295
11.15	Pointers to Functions	297
11.16	Review Questions	300
11.16.1	Objective Questions.....	300
11.16.2	Subjective Questions.....	302
11.16.3	Programming Exercises	302
12.	Structures and Unions	305
12.1	Introduction	305
12.2	Declaring a Structure	307
12.2.1	Tagged Structure Declaration.....	307
12.2.2	Structure Declaration Using typedef	308
12.2.3	Declaring Structure Variables.....	308
12.2.3.1	Declaring Structure Variables Using the Structure Name.....	308
12.2.3.2	Declaring Structure Variables after the Closing Braces	309
12.3	Initializing a Structure.....	311
12.4	Accessing Structure Members	312
12.4.1	Accessing Members Using the dot (.) Operator	313
12.5	Learn to Code Examples	315
12.6	Arrays of Structures.....	316
12.7	Structures within Structures (Nested Structures).....	318
12.7.1	Declaration of Nested Structures.....	318
12.7.1.1	Declare the Structure with One Declaration.....	319
12.7.1.2	Declare the Structure Separately	319
12.7.2	Accessing the Members of a Nested Structure	319
12.7.3	Nested Structure Initialization.....	320
12.8	User-defined Data Type: <i>typedef</i>	321
12.8.1	Uses of typedef.....	322
12.9	Pointers and Structures	323
12.9.1	Accessing Structure Members Using a Pointer	323
12.9.2	A Pointer as a Member of a Structure	324
12.9.3	Self-referential Structures	324
12.10	Structures and Functions	328

12.10.1	Passing Individual Members of a Structure.....	328
12.10.2	Passing the Whole Structure Using the Pass by Value Concept	330
12.10.3	Passing the Whole Structure Using the Pass by Address Concept.....	332
12.11	Unions.....	333
12.11.1	Declaration of a Union	333
12.11.2	Member Accessing.....	336
12.12	Structures vs. Unions.....	336
12.12.1	Size of Unions and Structures	337
12.12.2	Sharing Memory and Member Accessing	337
12.13	Bitfields.....	338
12.13.1	Declaration of a Bitfield	340
12.13.2	Uses of Bitfields.....	342
12.14	Enumeration	343
12.15	Review Questions	346
12.15.1	Objective Questions.....	346
12.15.2	Subjective Questions.....	346
12.15.3	Programming Exercises	346
13.	Dynamic Memory Allocation	349
13.1	Introduction	349
13.1.1	Process of Memory Allocation	350
13.1.1.1	Text Segments	351
13.1.1.2	Data Segments	351
13.1.1.3	Stack Segments	351
13.1.1.4	Heap Segments.....	351
13.2	Types of Memory Allocation	351
13.2.1	Static Memory Allocation.....	351
13.2.2	Dynamic Memory Allocation.....	351
13.3	Dynamic Memory Allocation Process.....	352
13.3.1	The malloc() Function.....	353
13.3.2	The calloc() Function.....	356
13.3.3	The realloc() Function	356
13.3.4	The free() Function.....	360
13.4	Review Questions	361
14.	File Handling.....	365
14.1	Introduction	365
14.1.1	Difference between Console I/O and File I/O.....	366
14.2	Basics of File I/O.....	367
14.2.1	What is a File?.....	367
14.2.2	File Handling Process Flow.....	367
14.3	Opening a File.....	368
14.4	Closing a File	370
14.5	File Functions with Examples	371
14.5.1	The fprintf() and fscanf() Functions.....	371
14.5.1.1	Writing and Reading an Integer Using fprintf() and fscanf()	372
14.5.2	The putw() and getw() Functions.....	374

14.5.2.1	Writing and Reading More than One Integer Using the <code>putw()</code> and <code>getw()</code> Functions.....	374
14.5.2.2	Reading Numbers from a File and Checking Them for Even or Odd.....	375
14.5.3	The <code>fputc()</code> and <code>fgetc()</code> Functions.....	377
14.5.3.1	Writing and Reading a Character Using <code>fputc()</code> and <code>fgetc()</code>	378
14.5.3.2	Writing and Reading Multiple Characters Using <code>fputc()</code> and <code>fgetc()</code>	379
14.5.3.3	Count Number of Characters, Lines, Tabs, and Blank Spaces Present in a File.....	380
14.5.4	The <code>fputs()</code> and <code>fgets()</code> Functions.....	381
14.5.4.1	Writing and Reading a String Using <code>fputs()</code> and <code>fgets()</code>	382
14.6	Other Programming Examples	383
14.7	Review Questions	386
15.	The Preprocessor.....	389
15.1	Introduction	389
15.2	Preprocessor Directives.....	389
15.3	Macro-substitutions	390
15.3.1	Writing Macros with Arguments.....	392
15.3.2	Removing a Macro.....	392
15.4	The <code>#include</code> Preprocessor.....	392
15.5	Conditional Preprocessors.....	395
15.5.1	The <code>#ifdef</code> and <code>#endif</code> Preprocessor Directives.....	395
15.5.2	The <code>#ifndef</code> and <code>#endif</code> Directives	396
15.5.3	The <code>#if</code> and <code>#endif</code> Directives	396
15.6	Other Preprocessor Directives.....	396
15.6.1	<code>#line</code> Directives	397
15.6.2	<code>#error</code> Directives.....	398
15.6.3	<code>#pragma</code> Directives.....	400
15.7	Review Questions	400
16.	Command Line Arguments.....	403
16.1	Introduction	403
16.1.1	The Code::Block IDE.....	404
16.2	Executing a Program Using a Command Prompt.....	405
16.2.1	Installing the minGW Compiler	405
16.2.2	Compiling and Executing a Program	407
16.3	Fundamentals of the Command Line Argument	410
16.4	Using Command Line Arguments	411
16.5	Review Questions	413
	Appendix A: ASCII Character Table	415
	Appendix B: Integer Representation.....	417
	Index	423

Preface

The C programming language is a general-purpose language of great importance to students, researchers, and software professionals. The TIOBE programming community index is an indicator of the popularity of programming languages. C was at the top of the list in March 2021. Experts believe that the C language serves as a preparatory step for individuals who aspires to learn other high-level languages like Java and Python. Since its invention and standardization in 1989, it is most popular among embedded systems and operating system developers. Several new standards of this language are currently in use by developers and are known as C99 and C11. This book is written for those who have no or only some basic prior knowledge about programming languages.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Acknowledgments

The idea of writing a book on C programming is quite an old thought of mine. I have been writing the draft while teaching this subject. Having made some handouts for my students, which they appreciated, I was motivated to write a book. I want to thank my students for continuously suggesting that I write it. Some other people also motivated me and supported me in completing it.

A heartfelt thanks to my wife, Priya Arundhati, and my son, Kritansh, for their encouragement, care, and love. Without them, my life would not be joyful. My parents and my family members are my strength and excellent motivators for me, always.

A sincere thanks to Aastha Sharma, Senior Editor, CRC Press – Taylor & Francis Group, for understanding my thoughts when writing this book and suggesting to me the necessary additions for the final proposal. I am also grateful to Shikha Garg, Senior Editorial Assistant, CRC Press – Taylor & Francis Group, who helped me produce the final manuscript. I am also thankful to all editorial members who assisted me during the production process of this book.

Finally, I would sincerely like to thank my friends and colleagues who directly or indirectly support me in doing this work.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Organization of this Book

To learn any programming language, we require a basic knowledge of computer fundamentals and number systems – covered in Chapters 1 and 2. Overall, the book contains 16 chapters organized so that an individual (beginner, intermediate programmer, or expert) will gain maximum benefit if they follow it sequentially.

As mentioned, Chapter 1 provides a brief overview of a computer system's fundamental components: input, output, memory, and processing devices. It also introduces the overall functions of an operating system that a programmer should know. Chapter 2 describes number systems, their types, and their conversion. Anyone knowing computer fundamentals and number systems can skip these first two chapters. A programmer should know how to find a solution to a given problem. Two primary components involved in solving a problem are algorithms and flowcharts. A detailed description of these is provided in Chapter 3. Chapter 4 introduces the C language, starting with its history and standards (e.g., C89, C99, C11). A detailed description of a C program structure and how to execute it in different environments is explained in this chapter.

Chapter 5 introduces language tokens such as constants, variables, and data types. The C language has a rich set of operators which are described in Chapter 6. The chapter also explains expressions and how to execute them according to the operator's precedence and associativity. Chapter 7 introduces input/output functions that a programmer uses to read inputs from the user and produces output on the screen. Chapter 8 describes control statements that include all decision-making and looping constructs. The concept of modular programming through functions is introduced in Chapter 9. The power of a function allows a programmer to divide a more significant problem into smaller subtasks and execute them by calling them when necessary. In Chapter 10, arrays and strings are explored to provide insight into storing and sequentially retrieving information. Chapter 11 explores pointers. The pointer concept is a vital part of the language and provides a mechanism to access memory content dynamically. It also enhances the language's features to support data structure. Chapter 12 discusses several concepts such as structure, union, bit fields, and enumerations. All these concepts are unique to the C language and enhance its characteristics. Chapter 13 addresses the dynamic memory allocation concept used to allocate memory dynamically and optimize memory allocation. Permanently storing information on a file and manipulating its content is a required feature for any programming language. Hence, in Chapter 14, we explore the concept of file handling in C. Other miscellaneous concepts such as the preprocessor and command-line arguments are described in Chapters 15 and 16.

This book includes more than 270 illustrations to explain the features of the C language. Every chapter begins with a discussion of a real-life scenario to explain the importance of that chapter, before describing the mechanisms supported by C to tackle that issue. All the fundamental concepts of C are covered with pleasing and feature-rich examples.

Suggestions to improve the content are always welcome. We request all our readers to send their findings, errors, comments, views, and feedback to make it a better book in this field. Please send your suggestions to: sisiriitg@gmail.com.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Author Biography

Sisir Kumar Jena is presently working as an assistant professor in the Department of CSE, DIT University, Dehradun. He was the HOD and an assistant professor in the Department of CSE at Nalanda Institute of Technology, Bhubaneswar, India, during 2007–15. He has more than ten years of teaching experience and five years of experience as a research scholar at IIT Guwahati. He has been pursuing his Ph.D. in Computer Science and Engineering at IIT, Guwahati, while writing this book. He has presented and published many research papers and book chapters at refereed international conferences and in journals. His interest area includes digital VLSI design and testing, approximate computing, IoT, and security in hardware.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

1

Introduction to the Computer

1.1 Introduction

Perhaps the most powerful and resourceful tool ever created by humanity is the computer. The term “computer” could mean a device that calculates. But nowadays a computer can do a variety of jobs. If you take any field, be it engineering, healthcare, automotive devices, gaming, or entertainment, everywhere a computer is used to do the task. This chapter introduces how computers came into existence and describes the different components associated with a computing device. After completion of this chapter, the reader will have learnt the following:

1. What a computer is, and how may we define it.
2. A brief history of the computer and how it came into existence.
3. What the different components of a computer system are, and how they are organized.
4. Be able to define a memory subsystem, its categories, and its organization.
5. Understand the importance of an Operating System (OS) and its functionality.

This chapter’s content is purely elementary and not meant for those who know the basics of computer systems. Those with a moderate knowledge of any computer language can skip this chapter. We will start the chapter by introducing the definition of a computer system and its characteristics.

1.2 Definition and Characteristics of a Computer System

The development of a computer system has a long history, which includes the work done by several great minds. In this modern era, we can consider a computer as a system that processes data and produces useful information. Though there is no formal definition that defines a computer, we will try to propose a definition:

A computer is an electronic device that receives input through an input device, stores it in the storage device (memory), and manipulates or processes the data to produce information (output) through an output device.

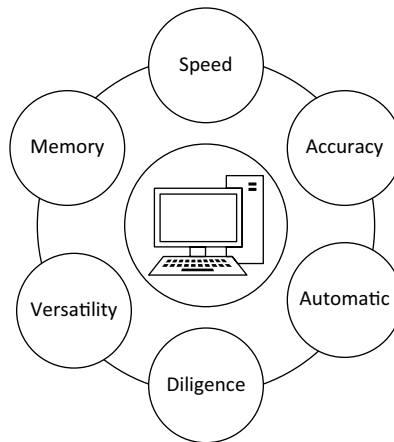


FIGURE 1.1
Characteristics of a computer.

Analyzing the definition, you may notice four crucial components of a computer system: (1) input device, (2) output device, (3) storage device, and (4) the processing device. The overall organization of all these components is discussed later in this chapter. The above defines a computer system as a data processing machine. But it is not only a data processing unit. Rather, it has several capabilities that need to be discussed here in the form of its characteristics. We could not complete the definition without knowing the characteristics of a computer system. Figure 1.1 shows the essential characteristics of a computer system collected from [1].

- *Speed*: A computer is a high-speed electronic device which takes a negligible amount of time to perform any task compared to the speed of any human being.
- *Accuracy*: Computers are very accurate in producing the correct output. A computer produces the wrong result only when the user has made a mistake.
- *Automatic*: Computers execute the task assigned to them without any intervention until the job gets finished.
- *Diligence*: Computers never tire. They can work continuously and produce correct and consistent results every time.
- *Versatility*: A computer is capable of doing different tasks. We all know that nowadays computers are used everywhere.
- *Memory*: A computer is potent at remembering things and never forgets them. Whatever we store in computer memory will be there throughout its lifetime.

1.3 History of the Computer

The computer has a long history of development. The objective of this book is something different. So in this section, we only provide an overview of how the computer came into existence by highlighting some inventions. The origin of the entire development is not



(a)



(b)

FIGURE 1.2

(a) Charles Babbage; (b) John von Neumann.

essential to us, but I would like to start with Charles Babbage's (see Figure 1.2a) contribution. He is known as the *father of the digital programmable computer*.

The work of Babbage describing a "difference engine" was used by many researchers as inspiration for further development [2], leading to several working models but with the disadvantage that the programs were hardwired and challenging to change. A final and major contribution was made during the 1940s by John von Neumann (see Figure 1.2b), known as the "stored program." With a stored program, we can control a computer system's activity, and this program is usually stored inside the computer's memory [2]. In today's modern world, digital computers are all built based on this stored-program concept. A brief contribution to computer development in chronological order is [3]:

- The German philosopher and mathematician, Gottfried Leibniz (1646–1716), built the first calculator to perform multiplication and division. It was not reliable due to the inaccuracy of its parts.
- Charles Babbage (1792–1872) (Figure 1.1a) was a British inventor who designed his difference engine in 1822 and, in 1842, came up with an "analytical engine" incorporating the ideas of a memory and card input/output for data and instructions. But he was not able to build the system. Babbage is mostly remembered for and considered as the father of digital computers.
- Howard Aiken (1900–73), a Harvard professor with IBM's backing, built the Harvard Mark I computer (51 ft long) in 1944. It required three seconds to perform multiplication.
- John Vincent Atanasoff built a specialized computer in 1941 and was visited by Willaim Mauchly before constructing the Electronic Numerical Integrator and Calculator (ENIAC).

- J. Presper Eckert and Mauchly designed and built the ENIAC in 1946 for military computations. It used vacuum tubes (valves), which were totally electronic (and operated in microseconds), instead of the electromechanical relay.
- Von Neumann was a scientific genius and a consultant on the ENIAC project. In 1950, he formulated plans with Mauchly and Eckert for a new computer, the Electronic Discrete Variable Automatic Computer (EDVAC), which was to store programs as well as data.
- At the same time (1950), another computer named the Electronic Delay Storage Automatic Calculator (EDSAC) was developed by Maurice Wilkes at Cambridge University in England.
- After the above inventions, every computer built followed the von Neumann architecture. Several generations of computers have been developed, but the overall architecture remains the same.

The reader of this book is encouraged to find out more on computer generations and present computer system scenarios.

1.4 Basic Computer Organization

Before we can understand basic computer organization, see Figure 1.3 of a desktop computer system and its components. There are five components that we usually see: (1) the keyboard; (2) the mouse (3); the monitor; (4) the cabinet; (5) the speaker. We also see some other components, such as the Uninterrupted Power Supply (UPS) and the joystick. Every component is not essential, but we connect them for ease of use.

Let us look inside the cabinet. Figure 1.4 shows what we can find inside, and we unveil only those components that help us to explain the basic organization of a computer system:

- A Central Processing Unit (CPU);
- Two types of memory (primary and secondary);
- A Switched-Mode Power Supply (SMPS);
- A motherboard that provides ports for connecting all other components.

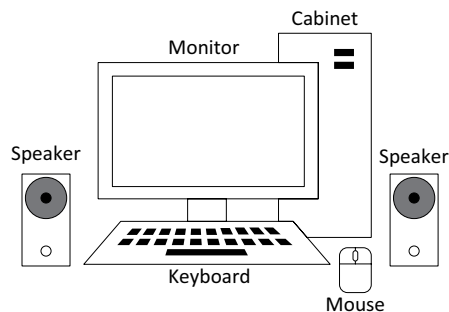


FIGURE 1.3
A desktop computer system.

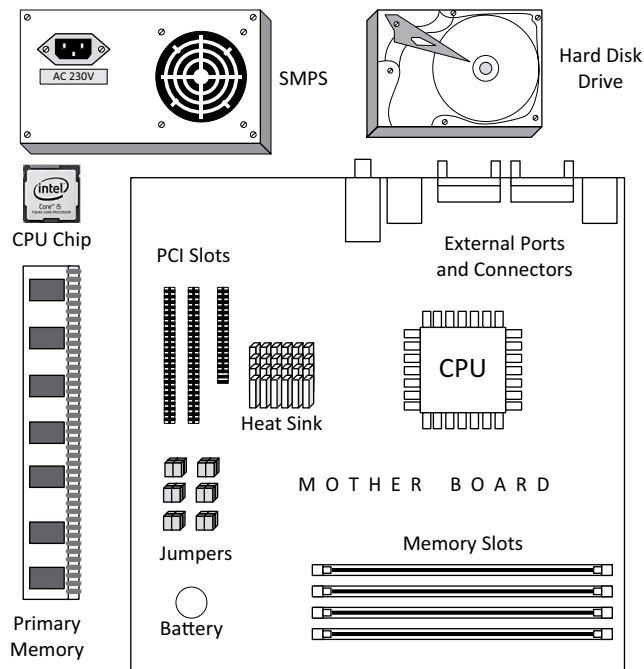


FIGURE 1.4
Major components inside the cabinet.

Readers are encouraged to explore more about these components and write an essay that will enhance their knowledge of a computer system.

According to [4], a simple digital computer should have five essential parts: (1) memory, (2) Arithmetic Logic Unit (ALU), (3) Control Unit (CU), (4) input devices, and (5) output devices. Among these components, the ALU and CU belong to the CPU. The CU of the CPU plays a significant role in executing the user's task. The CU controls the overall activity of the computer system and manages communication among the other components. To understand the overall execution of a task, look at Figure 1.5, which shows the basic organization of these components. The description of each component is given below.

1.4.1 Input Devices

This unit helps in supplying data and instruction to the computer system. For example, if we wish to instruct the computer to play a song, then we search for a particular song and click the play button using the mouse. In this scenario, the mouse acts as the input unit.

Similarly, if we wish to write a computer program, we use the keyboard as our input device. There are several input devices connected to our computer through which we give commands: mouse, keyboard, joysticks, and others.

An input device reads data or instructions from the user and sends it to the computer system for further processing.

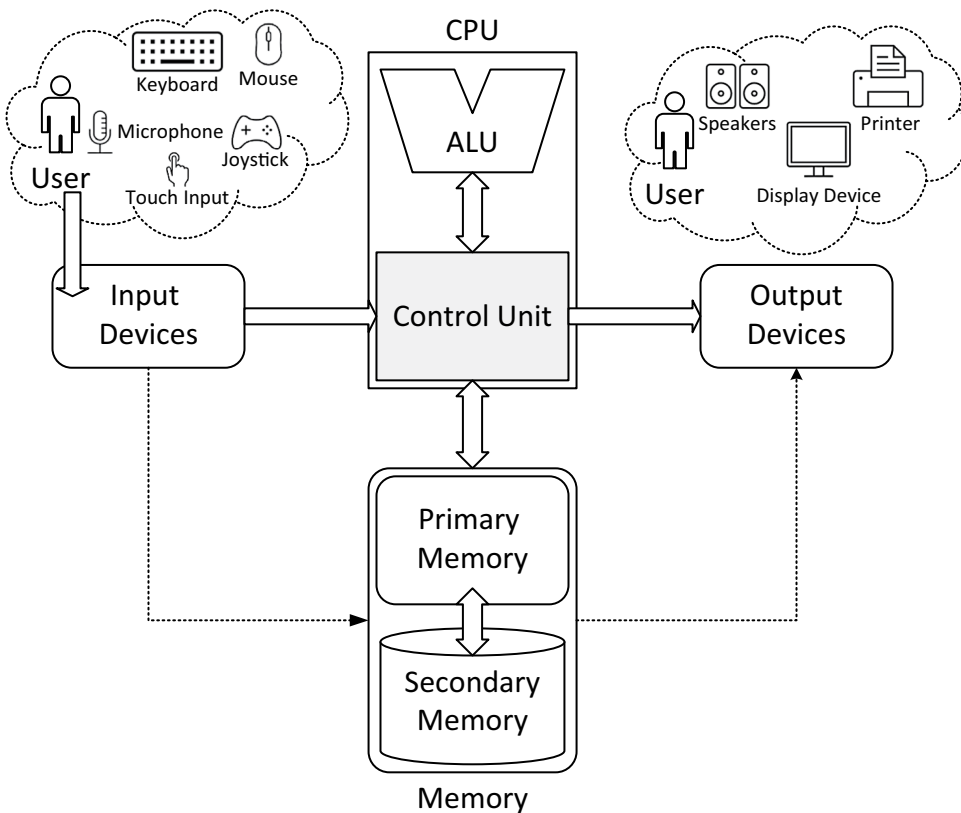


FIGURE 1.5
Basic organization of a computer system.

1.4.2 Memory

Every computer has a memory unit primarily categorized into two types: *primary memory* and *secondary memory*. We will further discuss the classification and description of these memory units in Section 1.5. The primary job of this unit is to store everything, and that includes the data or instructions read by the input unit, the intermediate result produced by the processor, and the final result.

Memory is used to store input data or instructions, intermediate results yielded during processing, and the final result produced by the processor.

1.4.3 Central Processing Unit

The CPU is called the brain of the computer system. It has two main components: the ALU and the CU. It reads the data from the storage unit, performs calculations as per the user instructions, stores the final result in the storage device, and displays the result to the user. The ALU is used to execute all the logical and arithmetic operations. The CU is used to control the overall activity of the computer system and manage the interaction among the different components.

The CPU is called the brain of the computer system. It controls and coordinates the interaction among different components and handles all the arithmetic and logical operations.

1.4.4 Output Devices

These help in displaying the results of a computation. For example, when we execute a program, the output is shown on the monitor screen. So, in this case, the monitor acts as the output device. Similarly, a speaker is also an output device.

An output device obtains the result produced by the CPU, converts it into a human-readable form, and displays it to the user of the computer system.

A programmer needs to understand how the instructions are executed inside the CPU. This requires a detailed exploration of the memory unit. The following section introduces the idea of memory-processor integration and its communication technique.

1.5 Computer Memory

A modern digital computer has several memory subsystems organized in a hierarchy, starting from the smallest high-speed registers to a high-capacity hard disk drive. A computer has the following main memory subsystems:

1. Registers;
2. Cache memory;
3. Primary memory;
4. Secondary memory.

Figure 1.6 shows how these memories are arranged with the CPU for easy communication and program execution. The main reason behind so many categories of memory lies in the two requirements: *speed* and *capacity*. The CPU needs high-speed memory because the execution speed of a CPU is relatively high compared to the data supplying capability (reading the content from memory and submitting it to the CPU for processing) of any memory. On the other hand, a user always needs a high-volume memory to permanently keep all their data and programs. The capacity of the secondary memory is relatively high compared to the main memory, cache, and registers. In contrast, fetching data from the register is faster compared to any other memory subsystem. It is evident that building high-speed memory incurs more *cost* as compared to low-speed memory.

A programmer needs to understand memory organization to write efficient programs. Let us describe a few points about each category of all memory types.

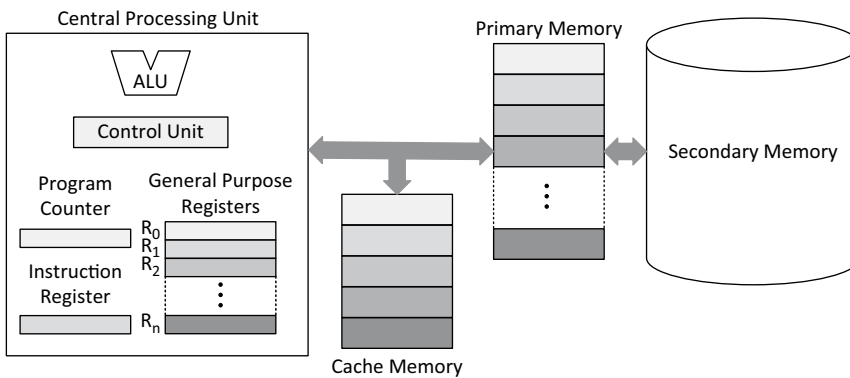


FIGURE 1.6
Memory organization.

1.5.1 Registers

These are the smallest, high-speed, volatile memories available inside the CPU for easy access while processing or executing a task. The total number of registers present inside a CPU varies from architecture to architecture. The most common ones are general purpose registers, program counters, and instruction registers. A *general purpose register* stores current and intermediate data of an executed instruction. There may be several general purpose registers available inside the CPU. A *program counter* keeps the information regarding the instruction about to run next. It is like a counter and holds an address that keeps increasing after the execution of the current instruction. The *instruction register* contains the instruction that is currently executing. The CPU fetches the instruction from the primary memory, places it inside the instruction register, decodes it, and finally executes it.

1.5.2 Cache Memory

Cache memory is a volatile memory present in between the main memory and the CPU register. The speed of this memory is higher than main memory and slower than the CPU registers. Every time the CPU requests the next instruction, the control unit first searches for it inside the cache. If it is not there, then the control brings a set of instructions (along with the required one) from the main memory and keep it inside the cache. The benefits of bringing the whole group lie in the principle of *locality of reference*: the CPU always fetches the instructions that are adjacent to each other, location wise.

1.5.3 Primary Memory

Primary memory, commonly referred to as the main memory, is a significant component of any computing device. It is an intermediate memory between the secondary memory and the CPU of the computer system. On request, the CPU needs to load an application in the main memory for execution. When a programmer finishes writing a program, he or she stores it in secondary memory. To execute that program, the CPU loads it from secondary memory to the primary memory. The processor is now directly interacting with the main memory and runs the program. At this point, it is necessary to understand the internal architecture of the main memory subsystem.

TABLE 1.1

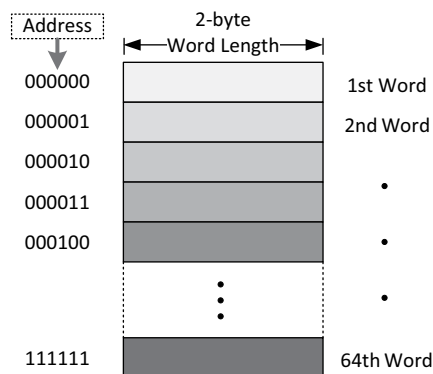
Units of Storage in Computer

Serial No.	Units	Description
1	bit	Refers to either 0 or 1
2	1 Nibble	4 bits
3	1 Byte	8 bits
4	1 Kilobyte (KB)	1024 Bytes (2^{10} Bytes)
5	1 Megabyte (MB)	1024 Kilobyte (2^{10} KB)
6	1 Gigabyte (GB)	1024 Megabyte (2^{10} MB)
7	1 Terabyte (TB)	1024 Gigabyte (2^{10} GB)
8	1 Petabyte (PB)	1024 Terabyte (2^{10} TB)
9	1 Exabyte (EB)	1024 Petabyte (2^{10} PB)
10	1 Zettabyte (ZB)	1024 Exabyte (2^{10} EB)

The main memory consists of several locations, and we can identify each location with an *address*. A location holds data or an instruction in the form of binary bits. A *bit* is the smallest unit of storage, refers to either 0 or 1, though it is not much use as a single unit. To be useful, a collection of bits is required: a group of four bits forms a *nibble*, a group of eight bits forms a *byte*; Table 1.1 shows the other units made from collections of bits.

Data stored in memory is in the form of a *word*, and the *word size* (*word length*) depends on the computer architecture. A modern computer system can have 16-bit (2-byte word), 32-bit (4-byte word), or 64-bit (8-byte word) word sizes. An example of a simple main memory is shown in Figure 1.7. The *word length* is 2 bytes, and there are 64 locations. Hence to identify each location uniquely, we need 6 bits ($2^6 = 64$) starting from the address 000000 to 111111.

Two types of primary memory exist in a computer system: Random Access Memory (RAM) and Read Only Memory (ROM). ROM has several variations: Programmable ROM (PROM), Erasable Programmable ROM (EPROM), Electrically Erasable Programmable

**FIGURE 1.7**

An example of a main memory subsystem.

ROM (EEPROM), and so on; and the description of each type is not included in the scope of this book. Similarly, RAM is also of two types: static RAM and dynamic RAM.

1.5.4 Random Access Memory

RAM is also known as the main memory of a computer system. In RAM, we can retrieve and store information randomly from any location:

1. RAM is the read–write memory of the computer;
2. RAM is volatile memory: the content of the RAM is erased when we switch off the computer system;
3. In RAM, it is possible to select any memory location randomly to store and retrieve information.

1.5.5 Read Only Memory

The information present in this memory is permanent, and we cannot modify the content. Hence it is a read only memory:

1. Information can be read only.
2. It is a non-volatile memory: the data present in this memory are permanent.
3. Such memories are also called permanent stores or dead stores.
4. It is generally used to store bootable information.
5. Generally, the computer manufacturer provides a ROM chip.

1.5.6 Secondary Memory

Before executing a program, you should write it and store it somewhere inside your computer memory. Primary memory will not store your program permanently; it can help your CPU execute the program by providing memory space. When your program finishes running, the allocated memory will be taken back – that’s why you need permanent storage to store your programs. Secondary memory does that job for you.

There are two types of secondary memory in today’s computers: a Hard Disk Drive (HDD) and a Solid-State Drive (SSD). SSDs are much faster compared to HDDs. SSDs have no mechanical moving parts, but HDDs have built-in platters called magnetic disks and a movable head that travels back and forth between the disks to access the data. On the other hand, SSDs are quite expensive compared to HDDs and have limited storage capacity.

1.5.7 Hard Disk Drive

Figure 1.8 shows the internal architecture of the currently available HDDs. As you can see, an HDD has a mechanical moving read–write head that moves back and forth through the platters. The figure shows the organization of the read–write head and its direction of movement. The platters are coated with a thin magnetic material and are used to record data. Each platter is divided into several *tracks* and *sectors* to store data, as shown in Figure 1.8. While accessing the data recorded on each platter’s surface, read–write heads move forward and backward, and, the platters rotate to bring the desired sector under the read–write head.

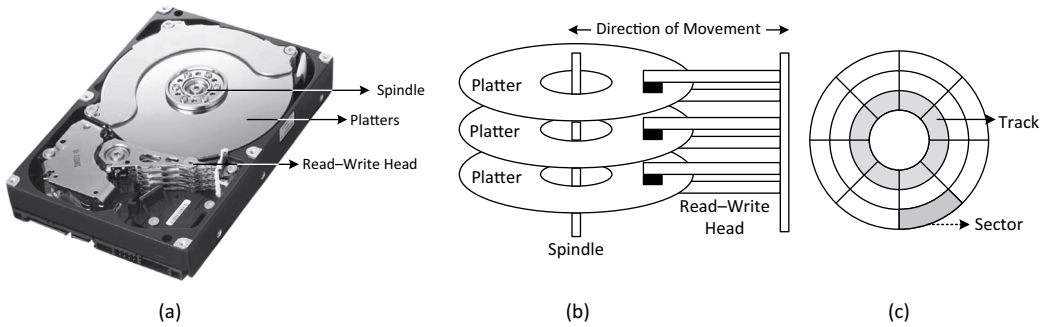


FIGURE 1.8
Internal architecture of an HDD.

1.5.8 Solid State Drive

Figure 1.9 shows a typical SSD and its block diagram [5–7]. An SSD uses solid-state memories to store data. The main component of an SSD is the NAND flash memory, a controller, a RAM, and a host interface. Figure 19.9b shows the organization of these components and their communication path. The *NAND flash memory*, sometimes called flash memory, is a non-volatile semiconductor device used to store data. There are several so-called flash blocks, consisting of 64 to 128 pages, to record user data. These pages are further subdivided into subpages, which is equal to the typical size of a sector. Each subpage also has two areas: a data area and a spare area. The data area stores the user’s data, and the spare area stores management information, such as bad sectors and error correction codes. Flash memories are arranged in the form of a package sharing an eight-bit-wide common input/output (I/O) bus. The *host interface* provides the connection to the host through interfaces such as a serial advanced technology attachment (SATA), a parallel advanced technology attachment (PATA), or a universal serial bus (USB). The *SSD controller* handles the read/write request coming from the user with the help of a RAM – as a temporary buffer. The RAM buffers the data through a read/write request before actually reading/writing from/to the flash memory.

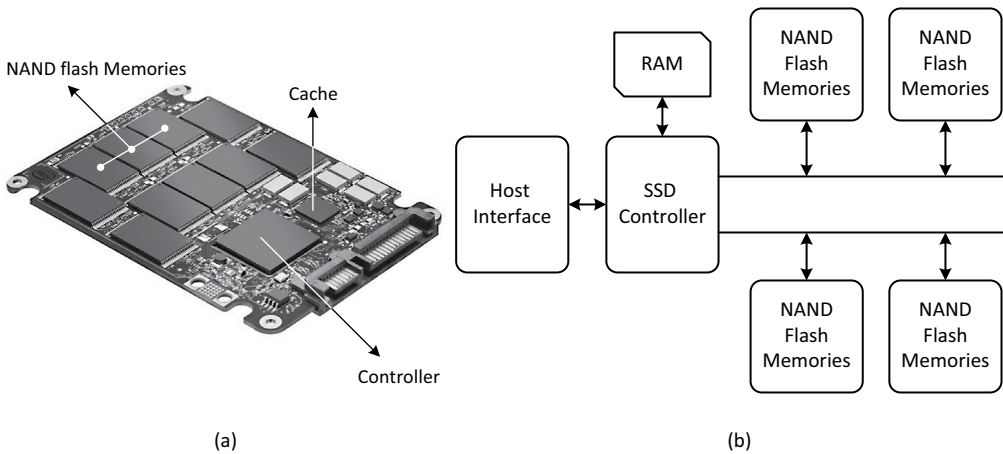


FIGURE 1.9
(a) A typical SSD; (b) Block diagram of an SSD.

1.6 Introduction to the Operating System

1.6.1 Hardware and Software

Before I introduce what an operating system is, I would like to discuss the concept of software (s/w) and hardware (h/w). The h/w is a physical device connected to your computer. S/w is a collection of programs, and we install them on a computer system to instruct the h/w to do something. That means, to interact with the h/w, we need the help of s/w. Buying a computer from the market and connecting it to a power supply does not help you interact with the computer. The only way you can instruct the computer to do anything is to install s/w and give commands through it. Hence, the s/w will act as an interface between the user and the computer system. Figure 1.10 shows the user and h/w interaction through s/w. The figure shows a three-layer communication: the h/w layer, s/w layer, and user layer. The user layer provides instruction to the s/w layer, which in turn instructs the h/w to execute the instruction.

S/w is mainly of two types: application s/w and system s/w (Figure 1.10). *Application s/w* is built for a specific task or specific application. For instance, Microsoft word, VLC media player, or any browser s/w like Google chrome is application s/w. On the other hand, the *system s/w* allows access to the h/w. It acts as the interface between the user and the h/w as well as application s/w and h/w. The relationship can be seen in Figure 1.11.

1.6.2 Operating System

An Operating System (OS) is system s/w that handles all the instructions coming from the user or application s/w and instructs the h/w to perform accordingly (Figure 1.11). Several OSs are available on the market, like Microsoft Windows 10, Ubuntu, Linux, Unix, and iOS. After buying a new computer, the first task is to install an OS on it; only then will the computer be usable. Later, we can install application s/w above the OS to do other specific tasks. If we want to listen to music, we can install s/w like Windows Media Player or VLC media player. Similarly, for word processing, we can install Microsoft Word or LibreOffice. Figure 1.12 shows the position of an OS in a layering architecture.

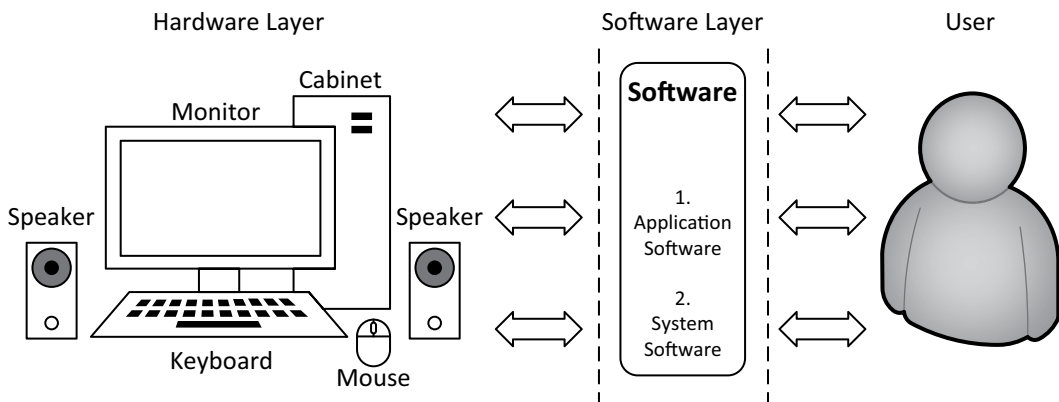


FIGURE 1.10
Hardware interaction through software.

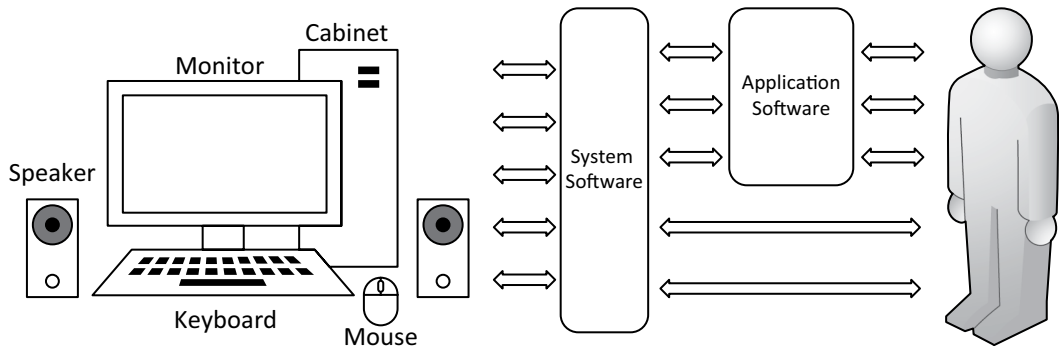


FIGURE 1.11
System software providing access to hardware.

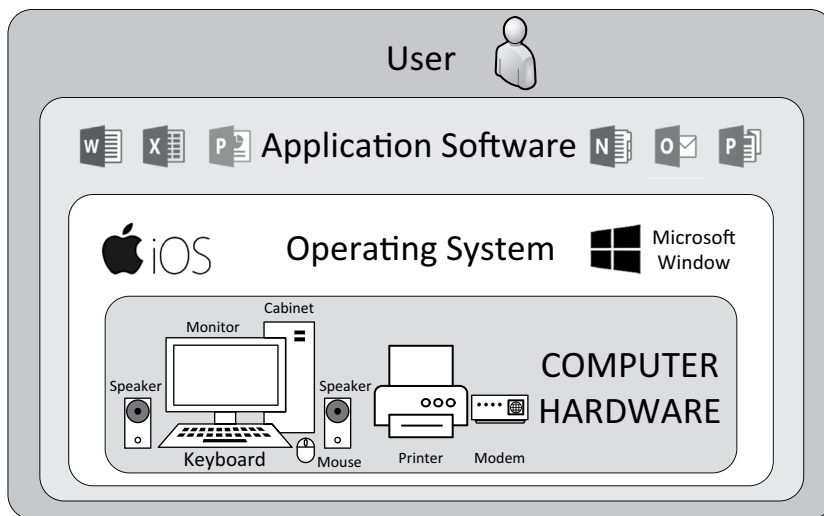


FIGURE 1.12
Operating System layer.

1.6.3 Functions of an Operating System

An OS manages several resources and handles all the requests that are coming from users and the application s/w. The broad classification of the OS functions includes:

1. Resource management;
2. Memory management;
3. Process management;
4. File management.

A detailed description of these functions is beyond the scope of this book. Readers are encouraged to explore more about these functions in any standard OS book.

1.7 Review Questions

1.7.1 Long Answers

1. What is a computer system, and what are its characteristics?
2. What are the basic components of a computer system? Explain its organization with a proper diagram.
3. What is a central processing unit (CPU)? Explain all the components of a CPU with an appropriate diagram.
4. What is computer memory? Explain the categories of several memory subsystems and their organization found in a computer system.
5. What are CPU registers? Explain the standard registers found inside a CPU irrespective of any architecture.
6. What is cache memory? Explain the need for cache memory and how it works.
7. Elaborate the working principle of cache memory to reduce the speed mismatch between the main memory and the CPU.
8. Explain the categories of the memory subsystem and state the difference between primary memory and secondary memory.
9. Draw the block diagram of the main memory subsystem and explain the detail of address space and word size.
10. What is the difference between RAM and ROM?
11. Explain the internal architecture of a hard disk drive (HDD) with its working principle.
12. Explain the internal architecture of a solid-state drive (SSD) with its working principle.
13. What are hardware and software? Explain the difference between them.
14. How does a user access/instruct hardware to execute an instruction? Explain the hardware–software communication required to accomplish a task.
15. What is the difference between application software and system software? Explain using an appropriate example.
16. What is an operating system (OS)? Explain the layered architecture of a system showing the OS layer.
17. Explain the various functions of an operating system.
18. Explore your computer and list all the application software and system software installed in your system.
19. Explore 32-bit and 64-bit operating systems. Which OS is installed in your computer system and why?
20. Explain the booting process of a computer system.

1.7.2 Short Answers

1. Who is known as the father of computer systems and why?
2. Who introduced the concept of the “stored program”? Explain it.

3. What is the full form of ENIAC, EDSAC, and EDVAC? Which one was developed first?
4. What are SMPS and UPS? Why are they necessary?
5. What is a control unit, where is it present, and what does it do?
6. What is the role of an ALU, and what operations does it perform?
7. Give some examples of input devices and output devices.
8. What is the meaning of locality-of-reference?
9. Explain the memory hierarchy, with respect to size, cost, and speed, of several memory subsystems.
10. What is a bit? How many bits form one byte?
11. List out the different variations of read only memory (ROM).
12. What is the difference between volatile and non-volatile memory?
13. What is NAND-flash memory, and where is it used?
14. What is word size or word length?
15. What is a program counter?
16. What is an instruction register?
17. What is a general purpose register?
18. What is the full form of SATA and PATA? What is the difference between them?
19. What is a bootstrap loader program?
20. What is the typical rotational speed of a hard drive in a modern computer system?

1.7.3 Practical Exercises

1. What should you know before purchasing a computer (whether a desktop or a laptop) from the market?
2. Do you think a bigger RAM capacity increases the speed of your computer system? Explore and explain.
3. What are x64 and x86 based operating systems? What is the difference between both of them? Try to find out which type of operating system is installed on your computer.
4. Explore your system and try to find out the following information about your computer:
 - a. What is the RAM size?
 - b. Which OS are you using?
 - c. What is the size of your secondary memory?
 - d. Is the secondary memory connected to your computer an HDD or SSD?
 - e. How can you find out how many processes are running on your system at any instant of time?
 - f. What is the speed of your processor?
 - g. How many cores does your processor have?
 - h. What is the size of your CPU register?
5. Study different operating systems, their types, and consider which one is best, and why.

6. Three major players in operating system design are Microsoft Windows, Linux, and Apple ios. Prepare a report on these OSs and learn how they are different from each other.
7. What is a pen drive? To which category of memory system, primary or secondary, does it belong?
8. Explore several keyboard shortcuts for your computer, create a new file, save a file, minimize the opened windows, toggle through windows, and other shortcuts that ease your operating capability.
9. When you shut down your computer, three options are shown in a Windows machine: shut down, restart, and sleep. What is the use of sleep?
10. Go to your computer settings and explore the detail of several components, such as system, personalize, account, and network & Internet.

References

1. Sinha, Pradeep K., and Priti Sinha. *Computer Fundamentals*. BPB Publications, 2010.
2. Randell, Brian, ed. *The Origins of Digital Computers: Selected Papers*. Springer, 2013.
3. Forouzan, Behrouz A., and Firouz Mosharraf. *Foundations of Computer Science*. Thomson, 2008.
4. Van Der Poel, W.L. A Simple Electronic Digital Computer. *Appl. Sci. Res.* 2, 367–400 (1952).
5. Kim, J., S. Seo, D. Jung, J. Kim, and J. Huh, "Parameter-Aware I/O Management for Solid State Disks (SSDs)," *IEEE Trans. Comp.*, vol. 61, no. 5, pp. 636–649, May 2012, doi: 10.1109/TC.2011.76.
6. Chen, F., R. Lee, and X. Zhang, "Essential Roles of Exploiting Internal Parallelism of Flash Memory Based Solid State Drives in High-speed Data Processing," *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, San Antonio, TX, 2011, pp. 266–277, doi: 10.1109/HPCA.2011.5749735.
7. Hu, Y., H. Jiang, D. Feng, L. Tian, H. Luo, and C. Ren, "Exploring and Exploiting the Multilevel Parallelism Inside SSDs for Improved Performance and Endurance," *IEEE Trans. Comp.*, vol. 62, no. 6, pp. 1141–1155, June 2013, doi: 10.1109/TC.2012.60.

2

Number Systems

2.1 Introduction

This chapter introduces several number systems used by computer systems. Readers can skip this chapter if they already familiar with this topic. The content of this chapter will help a programmer to understand several concepts that appear in future chapters. So if you are a beginner and learning this subject for the first time, I recommend reading the entire chapter thoroughly. After completing this chapter, the reader will know the following:

1. The various types of number systems that a computer uses to perform computations;
2. Conversion among number systems.

In our day-to-day life, we use different symbols, numbers, characters, letters, and so on. Where a numbering system is concerned, we are familiar with the *decimal number system*, which uses ten different digits (0, 1, 2, ..., 9) to represent a number. But, a computer system only understands two digits: either 0 or 1. When we make a number with these two digits (0 or 1), we call it a *binary number*. Other number systems also exist, and we classify them as shown in Figure 2.1. This chapter will introduce number systems and will show you how to convert a number from one type to another.

2.1.1 Non-positional Number Systems

In former times, humans counted on their fingers. When ten fingers were not adequate, stones, pebbles, and sticks were used to indicate the values. This type of counting is known as a non-positional number system.

2.1.2 Positional Number Systems

In a positional number system, every digit holds a position inside the number. For example, consider the decimal number 387, where 7 belongs to 0th place, 8 is in 1st place, and 3 is in 2nd place. Not only the position but also its type and base matter. In general, each digit of a number is determined by three factors:

1. The digit itself;
2. The position of the digit in the number;
3. The base of the number system.

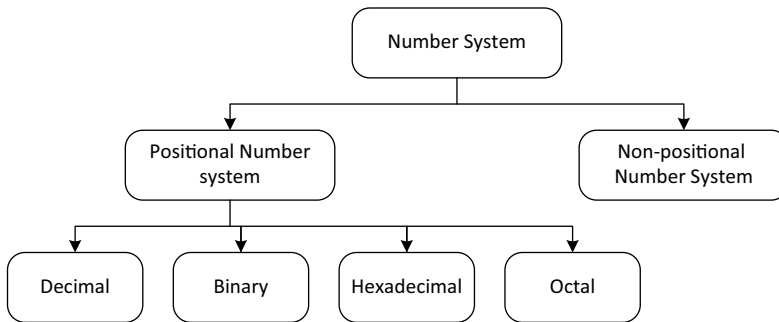


FIGURE 2.1
Categories of number systems.

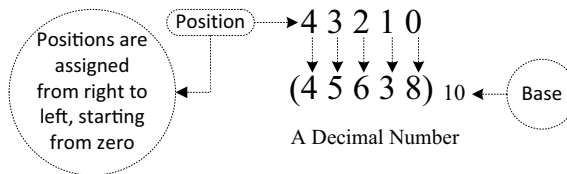


FIGURE 2.2
A positional number.

Figure 2.2 shows an example by taking a decimal number 45638, where the base is 10, and the positions of each digit are assigned from right to left, starting from 0 (zero).

There are several benefits in representing a number in this form. In this chapter, you will find a description and representation of all positional numbers and the position of each digit. We will start with a description of the decimal number system because we use it in everyday calculations.

2.2 Positional Number Systems

In this section, we will describe the detail of each positional number system, starting with the decimal number system. The description of each number system follows a simple flow. First, we will talk about the symbols (digits) used, and then we will determine the position of each digit, and finally show how each number system is related to the decimal number system.

2.2.1 Decimal Number System

Today, most people use decimal number representation for counting. In the decimal number system, there are ten digits:

0,1,2,3,4,5,6,7,8,9

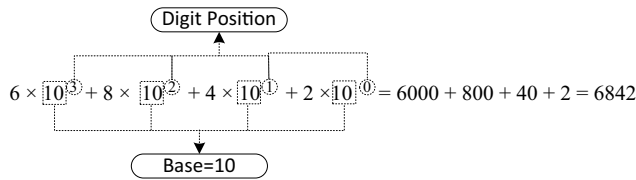


FIGURE 2.3
Place value of each digit in a decimal integer number.

These digits can represent any value, for example: 6842 (read as six thousand eight hundred and forty-two). Figure 2.3 shows the place value of each digit in a decimal number. We are concentrating on *integer numbers* now; later, we will show you how to deal with real numbers.

We first multiply each digit with the base to the power of its position. After that, we add the result of the above operation to get the value (see Figure 2.3). In a decimal number system, the *base* is 10. In general, in any number system, each digit’s position is assigned from the left starting from 0. In our example, we have four digits, 6, 8, 4, and 2, with positions 3, 2, 1, and 0, respectively. The digit 6 is multiplied by the base 10 to the power of its position 3, and we get 6000. Similarly, the digits 8, 4, and 2 are multiplied by 10 to the power of their corresponding positions (2, 1, and 0) to get 400, 40, and 2, respectively. Finally, all the results are added to get the value.

NOTE

Any number to the power 0 is 1. Even 0 to the power of 0 is also 1.

$$10^0 = 1$$

$$5^0 = 1$$

$$0^0 = 1$$

Now consider a *real number* description where the number contains a fractional part. The positions of the digits present before the decimal point follow a similar representation, like an integer. But, the positions are slightly different for the digits present after the decimal point. For the latter case, the positions are assigned from left to right, starting with -1 .

Consider one example 68.732 (read as sixty-eight point seven three two). In this example, 68 is considered as an integer and treated in a similar manner as discussed above. The digits present after the decimal point (i.e., 732) are assigned in a sequence starting from -1 . Figure 2.4 shows this assignment and the calculation involved in generating the value. Here, 1, 0, -1 , -2 , -3 are positions assigned to 6, 8, 7, 3, 2, respectively. The digits in the given number are multiplied by the base to the power of its respective position and finally added to get the value.

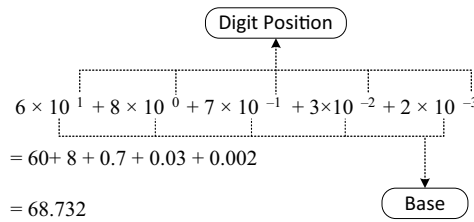


FIGURE 2.4
Place value of each digit in a decimal real number.

2.2.2 Binary Number System

The binary number system is the one that is actually used in a computer system. A computer is made up of electronic components and can have only two states: ON or OFF. We can represent ON as 0 and OFF as 1. So, any number made up of 0 and 1 we call a binary number. Computers use the binary number system to represent everything. The *base* in a binary number system is 2. Each digit in a binary number system is called a bit; 8 bits form a byte.

Like a decimal number system, a binary number also follows the same rule of assigning a position to each digit. The difference is in their bases: in a decimal number, the base is 10, and in binary, it is 2. Figure 2.5 shows our example binary number, its base, and the position of each digit in the binary number. Let us see what happens when the digits in a given binary number are multiplied by the base to the power of its respective position and finally added. We take our example binary number to perform this operation. Figure 2.5 shows this operation, and we notice that it produces 53 as a result, which is nothing but the decimal equivalent of the given binary number.

Let's think about the above explanation the other way. The equivalent decimal value of the binary number $(110101)_2$ is $(53)_{10}$. How do we calculate this? We can find it by the sum of each digit multiplied by the base to the power of its position (Figure 2.5).

Now, consider a binary number that contains a decimal point. We can treat the number in the same way as the decimal real number. We assign position numbers to the digits

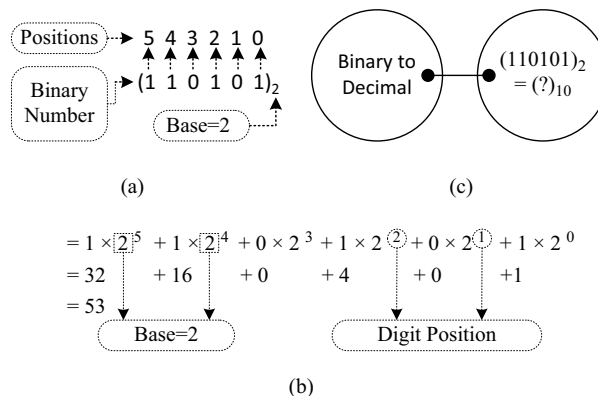


FIGURE 2.5
Binary number system and its associated operations.

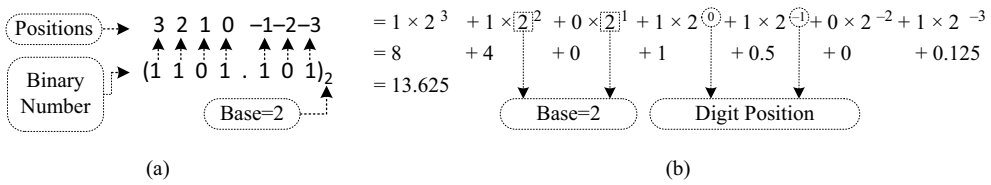


FIGURE 2.6 Binary number with a decimal point and its conversion.

present after the decimal point from left to right, starting from -1 . The digits present before the decimal point do not require any special treatment. Figure 2.6 shows an example of a binary number with a decimal point and its position assignment, and shows the calculation steps to find its equivalent decimal number.

2.2.3 Hexadecimal Number System

The hexadecimal number system uses 16 symbols (called *hexadecimal digits*) to represent a number:

$$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F$$

The hexadecimal digits A, B, C, D, E, F refer to numerical values 10, 11, 12, 13, 14, 15, respectively. The *base* is 16 because it uses 16 hexadecimal digits. The benefit of using a hexadecimal number system over a binary number system is its small length in representing a number. A four-bit binary number requires only one hexadecimal digit to represent the same number, as we will see shortly.

We follow the same rule to assign positions to each digit of a hexadecimal number, starting from 0, assigned from right to left. Figure 2.7 shows a hexadecimal number with its assigned position numbers. We can calculate the decimal equivalent of that given hexadecimal number using the same principle followed in the other number systems discussed above. We first multiply each digit by its base to the power of its corresponding position and finally add it. The calculation procedure is shown in Figure 2.7. Hence, we can say the hexadecimal number 24B4 is equivalent to decimal number 9396.

Hexadecimal numbers with a decimal point are not so common, but we can represent and process it the same way as binary numbers with a decimal point. We will discuss the conversion process later in this chapter.

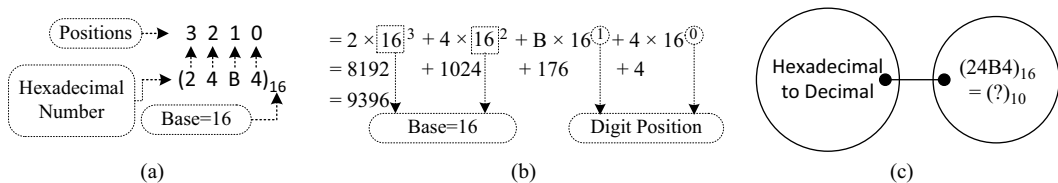


FIGURE 2.7 Hexadecimal number system and its associated operation.

2.2.4 Octal Number System

The octal number system uses eight symbols (*octal digits*) to represent a number:

$$0, 1, 2, 3, 4, 5, 6, 7$$

The *base* is 8 because it uses eight octal digits to represent a number in the octal number system. The benefit of using an octal number system over a binary number system is its small length in representing a number. A three-bit binary number requires only one octal digit to represent the same number, as we will see shortly.

We follow the same rule to assign positions to each digit of an octal number, starting from 0, assigned from right to left. Figure 2.8 shows an octal number with its assigned position numbers. We can calculate the decimal equivalent of that given octal number using the same principle followed in the other number systems discussed above. We first multiply each digit by its base to the power of its corresponding position and finally add it. The calculation by procedure is shown in Figure 2.8. Hence, we can say the octal number 435 is equivalent to decimal number 285.

Octal numbers with a decimal point are also not so common, but we can represent and process it the same way as binary numbers with a decimal point. We will discuss the conversion process later in this chapter.

From the above discussion, we can derive a generalized process of converting a number with any base to its decimal equivalent (Figure 2.9). The following description refers to the

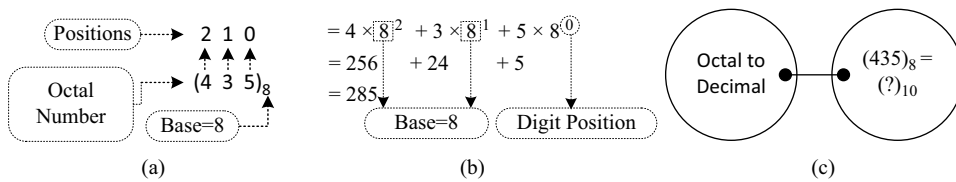


FIGURE 2.8
 Octal number system and its associated operation.

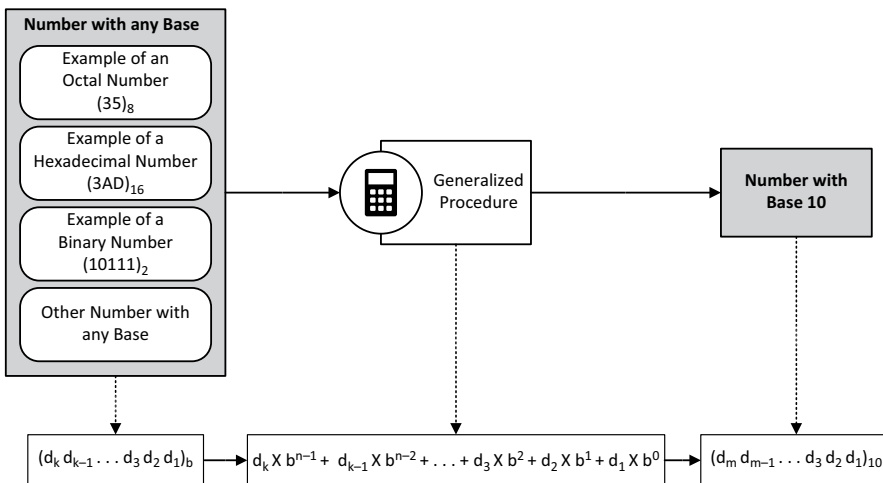


FIGURE 2.9
 Generalized procedure to convert a number with any base to its decimal equivalent.

integer numbers only. To describe the process, we need to represent a given number (any base) with the following notation:

$$(d_k d_{k-1} \dots d_3 d_2 d_1)_b$$

where,

d_i represents a digit in the given number, and i ranges from 1 to k ;

b represents the base of the given number.

The digits of the given number are assigned by its position number starting with 0 from right to left. That means the position 0, 1, 2, ..., $n-1$ is assigned to $d_1, d_2, d_3, \dots, d_k$, respectively. To calculate the decimal equivalent, each digit (d_i) of the given number should be multiplied by its base (b) to the power of its position:

$$d_k \times b^{n-1} + d_{k-1} \times b^{n-2} + \dots + d_3 \times b^2 + d_2 \times b^1 + d_1 \times b^0$$

Performing the above operation, we can find its corresponding decimal equivalent:

$$(d_m d_{m-1} \dots d_3 d_2 d_1)_{10}$$

where,

d_i represents a digit in the given decimal number, and i ranges from 1 to m ;

10 represents the base of the given number, $b = 10$.

Figure 2.9 shows the entire conversion process.

2.3 Number Conversion

It is essential to know the conversion of numbers from one base to another base. We know that computers only understand binary numbers, and what we know are decimal numbers. Hexadecimal numbers are used to represent everything in a compact form so that it is easy to express and is more readable. As programmers, we need to know the details of conversion among these numbers. In this section we will follow the question–answer pattern to show each conversion. A description is given where necessary.

2.3.1 Binary to Decimal

There are two approaches to converting binary to decimal. The first method has already been discussed in Section 2.2.2. With this method, we will assign a position number to each digit starting from 0, right to left. Then, we multiply the digit with the base to the power of its position number. Finally, we add all the results obtained in the last step to get its decimal equivalent. The second approach is a shortcut method to obtain the decimal equivalent. We will discuss both these approaches using appropriate examples.

2.3.1.1 Approach 1

Given number: $(10011)_2$. The approach is shown in Figure 2.10. It is a three-step process:

1. Assign a position number to each digit of the given binary number;
2. Multiply the digit with the base to the power of its position number;
3. Add the result to find its decimal equivalent.

2.3.1.2 Approach 2

To find the decimal equivalent with this approach, we need to remember a formula. The formula is built by listing the powers of 2 starting from 0, i.e., $2^0, 2^1, 2^2, \dots$, and so on. If we write their values, they will be 1, 2, 4, 8, 16, \dots , etc.; the number is doubled whenever you write the next term. The number of terms depends on the number of digits present in the given binary number. For instance, if you want to find out the decimal equivalent of $(1011)_2$, then four terms are required, because there are four digits present in the given binary number, and the terms are 1, 2, 4, and 8. Similarly, if the given number is $(1101101)_2$, then we require seven terms and they will be 1, 2, 4, 8, 16, 32, 64. The term must be assigned from the right to the left corresponding to each digit present in the given binary number.

The next step with this approach is quite simple. We need to add the terms corresponds to digit 1 to get its decimal equivalent. The terms that are assigned to the digit 0 will be ignored and do not take part in the decimal calculation. Figure 2.11 shows the steps of this approach using an example. The given number is $(10011)_2$.

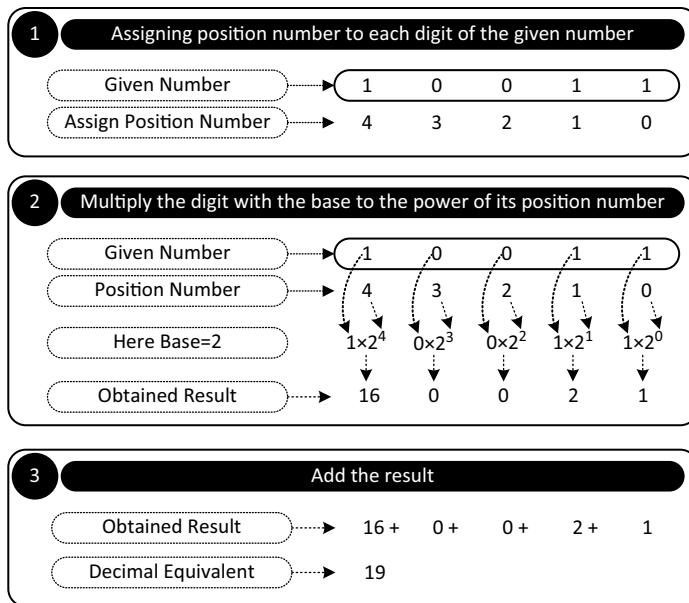


FIGURE 2.10 Binary to decimal conversion steps.

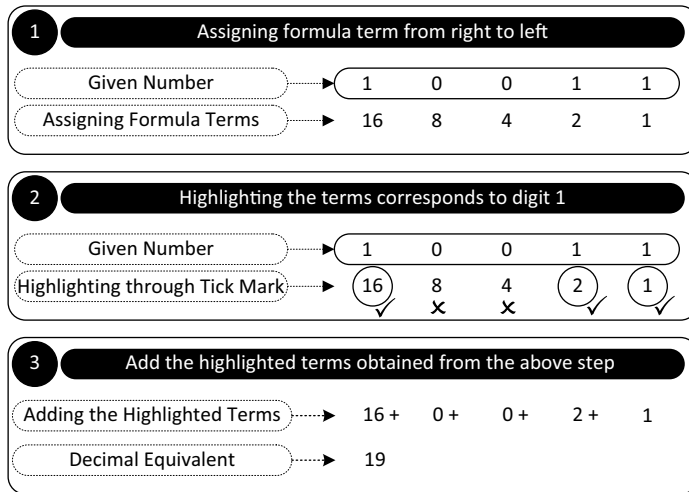


FIGURE 2.11
Binary to decimal conversion (short cut).

SHORT-CUT METHOD FORMULA TO CONVERT BINARY TO DECIMAL

Assign the terms from right to left, starting from 1, double the term value as you proceed towards the left.

Let us take another example. Suppose you want to find the decimal equivalent of $(110011011)_2$. The short cut is given below (see Figure 2.12).

2.3.2 Binary Fraction to Decimal Conversion

We can convert a given binary fraction to its equivalent decimal fraction. The approach is straightforward and does not require any explanation. The process of conversion has already been discussed in the last section. Learners are advised to go through the examples and try to understand the procedure.

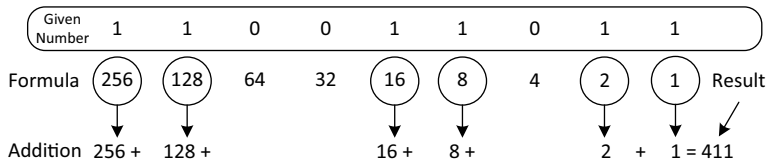


FIGURE 2.12
Another example showing the conversion from binary to decimal.

EXAMPLES

1. Convert $(.111)_2$ to its decimal equivalent.

Solution

$$\begin{aligned}
 \text{Given } (.111)_2 & \\
 &= 1 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} \\
 &= \frac{1}{2} + \frac{1}{4} + \frac{1}{8} \\
 &= 0.5 + 0.25 + 0.125 \\
 &= (0.875)_{10}
 \end{aligned}$$

2. Convert $(1110.1001)_2$ to its decimal equivalent.

Solution

$$\begin{aligned}
 \text{Given } (1110.1001)_2 & \\
 &= 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} \\
 &= 8 + 4 + 2 + 0 + \frac{1}{2} + 0 + 0 + \frac{1}{16} \\
 &= 8 + 4 + 2 + 0 + 0.5 + 0 + 0 + 0.0625 \\
 &= (14.5625)_{10}
 \end{aligned}$$

2.3.3 Binary to Decimal Conversion

When we want to convert a decimal number to its equivalent binary number, we need to divide the decimal number by 2 until the quotient becomes 0 and no further division is possible. During the division process, we must note down the remainders. Arranging the remainders using the bottom-up approach gives us the binary equivalent. Let us take an example to understand the process of conversion.

EXAMPLE

Convert the decimal number $(156)_{10}$ to its binary equivalent. We can follow the steps given below to convert any decimal number to its binary equivalent (refer to Figure 2.13).

- Step 1: Write the decimal number as the dividend inside an upside-down “long division” symbol. Write the base of the destination system (in our case, “2” for binary) as the divisor outside the curve of the division symbol.

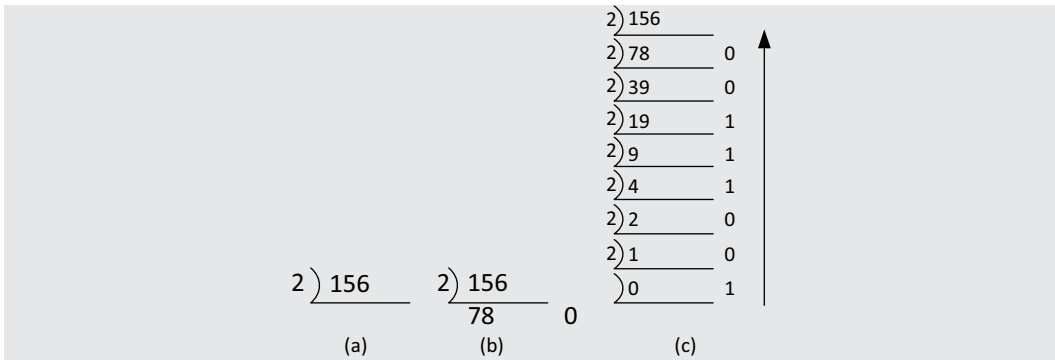


FIGURE 2.13
 Decimal number to binary number conversion process.

- Step 2: Write the integer answer (quotient) under the long division symbol and write the remainder (0 or 1) to the right of the dividend.
- Step 3: Continue downwards, dividing each new quotient by 2 and writing the remainders to the right of each dividend. Stop when the quotient is 0.
- Step 4: Starting with the bottom 1, read the sequence of 1’s and 0’s upwards to the top. You should have 10011100. This is the binary equivalent of the decimal number 156. Hence, $(156)_{10} = (10011100)_2$.

2.3.4 Decimal Fraction to Binary Fraction

In a decimal fractional number, the number appears before the decimal point is converted to binary following the techniques discussed in the previous section; the number which appears after the decimal point will be converted by repeatedly multiplying by 2. During the multiplication, the digit which appears before the decimal point is collected from top to bottom. Let us take some examples to show the process of conversion.

EXAMPLES

1. Convert $(0.8125)_{10}$ to binary.
 As the given number has no number that appears before the decimal point, the process is straightforward. Multiply the given number by 2 and record the numbers that appear before the decimal point. Collect them from top to bottom. Figure 2.14 shows the entire process of conversion. Hence $(0.8125)_{10} = (0.1101)_2$.
2. Convert $(8.8125)_{10}$ to its binary equivalent.
 The given problem has two parts; the number that appears before the decimal point will be solved as per the approaches discussed in the last section and again as described in Figure 2.14. The number that appears after the decimal point will follow the procedure discussed in the previous example shown in Figure 2.14. Hence, $(8.8128)_{10} = (1000.1101)_2$.

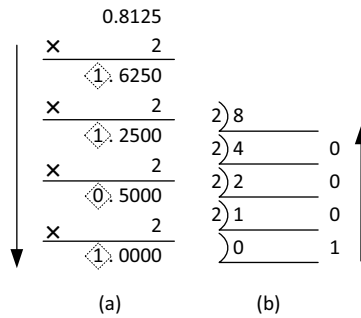


FIGURE 2.14
Decimal fraction to binary fraction.

2.3.5 Decimal to Octal Conversion

Conversion from a decimal number to an octal number is quite simple, like the decimal to binary conversion discussed above. The only difference is, there you divide it by 2, but here you divide the given decimal by 8. The following steps explain the process of conversion.

- Step 1: Divide the decimal number by 8 and obtain the quotient and remainder;
- Step 2: Divide the quotient by 8 and obtained the new quotient and remainder;
- Step 3: Repeat step 2 until the quotient is equal to 0;
- Step 4: Take the remainder from bottom to top for the answer.

EXAMPLE

Convert $(359)_{10}$ to its equivalent octal number.

Solution

The solution is shown in Figure 2.15 (a) and is self-explanatory.
Hence, $359_{10} = 547_8$.

It is also possible to convert a given *decimal fractional number to its octal equivalent*. The process is quite similar to the previous approaches. The numbers are treated separately; numbers that appear before the decimal place will be converted as the process discussed

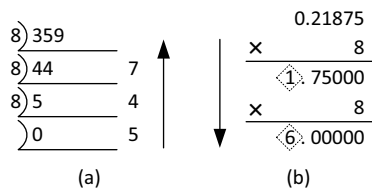


FIGURE 2.15
Decimal to octal conversion.

above, and the number that appears after the decimal will be multiplied by 8. The digit that appears before the decimal point will be collected from top to bottom.

EXAMPLE

Convert $(0.21875)_{10}$ to its octal equivalent.

Solution

The solution is shown in Figure 2.15 (b) and is self-explanatory. Hence, $(0.21875)_{10} = (0.16)_8$.

2.3.6 Octal to Decimal Conversion

We can find this by summing each digit multiplied by the base (8 here) to the power of its position. This has already been discussed. This section will show you the conversion of the octal fractional number to its equivalent decimal fractional number. The process is quite simple; observe the following example to understand it.

EXAMPLE

Convert $(0.34)_8$ to its equivalent decimal number.

Solution

Given number: $(0.34)_8$

$$= 3 \times 8^{-1} + 4 \times 8^{-2}$$

$$= 3 \times \frac{1}{8} + 4 \times \frac{1}{64}$$

$$= 0.375 + 0.0625$$

$$= (0.4375)_{10}$$

2.3.7 Octal to Binary Conversion

To convert a number from octal to binary, we can follow two procedures.

2.3.7.1 Procedure 1

With procedure 1, we follow a two-step process and require two conversions.

Step 1: Convert the octal number to its equivalent decimal number;

Step 2: Convert the decimal number obtained above to binary.

The above procedure is quite lengthy. Hence, we follow Procedure 2 here.

2.3.7.2 Procedure 2

This is a short-cut procedure. We need to represent each digit of an octal number as a three-bit binary. Why? Because we know that an octal number can be formed using digits from 1 to 7, and if you convert any digit to its binary equivalent, it requires only three bits. Observe Table 2.1 to see the binary equivalent of all these digits.

Let us take an example to explain the conversion procedure. We want to convert $(7263)_8$ to its equivalent binary; we need the following steps:

- Step 1: Write the given octal number as shown in Figure 2.16.
- Step 2: Extract the binary equivalent of each digit of the given octal number from Table 2.1 and write it as shown in Figure 2.16.
- Step 3: Read the binary number from left to right to obtain the binary equivalent. Hence, $(7263)_8 = (111010110011)_2$.

2.3.8 Binary to Octal Conversion

To convert a number from binary to octal, we can follow two procedures.

2.3.8.1 Procedure 1

With Procedure 1, we follow a two-step process and require two conversions:

- Step 1: Convert the binary number to its equivalent decimal number;
- Step 2: Convert the decimal number obtained above to octal.

The above procedure is quite lengthy. Hence, we follow Procedure 2 here.

TABLE 2.1

Binary Equivalent of Octal Digits

Octal Digit	Binary Equivalent	Octal Digit	Binary Equivalent
0	000	4	100
1	001	5	101
2	010	6	110
3	011	7	111

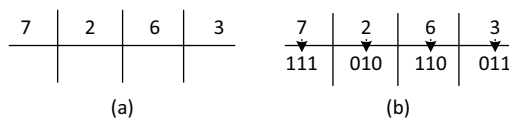


FIGURE 2.16

Short-cut method to convert octal to binary.

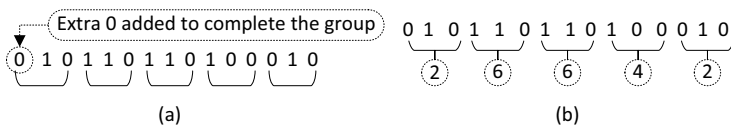


FIGURE 2.17
Short-cut method to convert binary to octal.

2.3.8.2 Procedure 2

This procedure is quite simple and easy to follow. Let us take an example to understand how this procedure works.

EXAMPLE

Convert $(10110110100010)_2$ to its octal equivalent.

Solution

We need the following steps to find the octal equivalent:

- Step 1: Figure 2.17 shows the first step. Divide the binary number into groups of three bits each (from right to left) by adding 0 bits for completing the groups (if needed).
- Step 2: Figure 2.17 shows the next step and is the final one to obtain our octal equivalent. Replace each group by its octal equivalent following Table 2.1. Hence, $(10110110100010)_2 = (26642)_8$.

Now consider a fractional binary number. To convert it to octal, we use the following steps:

- Step 1: Divide the binary number before the binary point into groups of three bits each (from right to left) and after the binary point into groups of three bits each (from left to right) by adding 0 bits for completing the groups (if needed);
- Step 2: Replace each group with its octal equivalent following Table 2.1.

EXAMPLE

Convert $(101010011011.10100011)_2$ to its octal equivalent.

Solution

The solution procedure is shown in Figure 2.18, and is self-explanatory. Hence, $(101010011011.10100011)_2 = (5233.506)_8$.

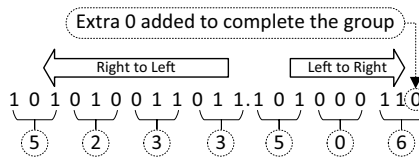


FIGURE 2.18
Fractional binary number to octal conversion.

2.3.9 Decimal to Hexadecimal Conversion

Conversion from a decimal number to a hexadecimal number is quite simple, like the decimal to binary conversion discussed above. The only difference is, there you divide it by 2, but here you divide the given decimal by 16. The following steps explain the process of conversion.

- Step 1: Divide the decimal number by 16 and obtain the quotient and remainder;
- Step 2: Divide the quotient by 16 and obtained the new quotient and remainder;
- Step 3: Repeat step 2 until the quotient is equal to 0;
- Step 4: Take the remainder from the bottom to top for the answer.

EXAMPLE

Convert $(58)_{10}$ to its equivalent hexadecimal number.

Solution

The solution is shown in Figure 2.19(a) and is self-explanatory.

Hence, $58_{10} = 3A_{16}$.

It is also possible to convert a given *decimal fractional number to its hexadecimal equivalent*. The process is quite similar to the previous approaches. The numbers are treated separately; numbers that appear before the decimal place will be converted as the process discussed above, and the number that appears after the decimal will be multiplied by 16. The digit that appears before the decimal point will be collected from top to bottom.

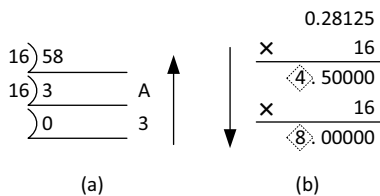


FIGURE 2.19
Decimal to hexadecimal conversion.

EXAMPLE

Convert $(0.28125)_{10}$ to its hexadecimal equivalent.

Solution

The solution is shown in Figure 2.19 and is self-explanatory. Hence, $(0.28125)_{10} = (0.48)_{16}$.

2.3.10 Hexadecimal to Decimal Conversion

We can find this by summing each digit multiplied by the base (16 here) to the power of its position. This has already been discussed in the previous section. In this section, we will show you the conversion of the hexadecimal fractional number to its equivalent decimal fractional number. The process is quite simple; observe the following example to understand it.

EXAMPLE

Convert $(0.48)_{16}$ to its decimal equivalent.

Solution

Given number: $(0.48)_{16}$

$$\begin{aligned}
 &= 4 \times 16^{-1} + 8 \times 16^{-2} \\
 &= 4 \times \frac{1}{16} + 8 \times \frac{1}{256} \\
 &= 4 \times 0.0625 + 8 \times 0.00390625 \\
 &= (0.28125000)_{10}
 \end{aligned}$$

2.3.11 Hexadecimal to Binary Conversion

To convert a number from hexadecimal to binary, we can follow two procedures.

2.3.11.1 Procedure 1

With Procedure 1, we follow a two-step process and require two conversions:

- Step 1: Convert the given hexadecimal number to its equivalent decimal number;
- Step 2: Convert the decimal number obtained above to binary.

The above procedure is quite lengthy. Hence, we follow Procedure 2 here.

TABLE 2.2

Binary Equivalents of Hexadecimal Numbers

Hexadecimal Number	Binary Equivalent	Hexadecimal Number	Binary Equivalent
0	0000	8	1000
1	0001	9	1001
2	0010	A (10)	1010
3	0011	B (11)	1011
4	0100	C (12)	1100
5	0101	D (13)	1101
6	0110	E (14)	1110
7	0111	F (15)	1111

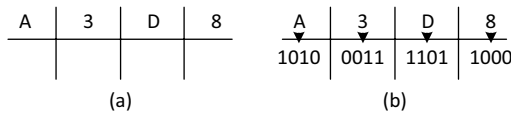


FIGURE 2.20
Short-cut method to convert hexadecimal to binary.

2.3.11.2 Procedure 2

This is a short-cut procedure. We need to represent each digit of a hexadecimal number as a four-bit binary. Why? Because we know that a hexadecimal number can be formed using digits from 1 to 9 and then A, B, C, D, E, and F. Where, A = 10, B = 11, C = 12, D = 13, E = 14, F = 15. If you convert any digit to its binary equivalent, it requires only four bits. Observe Table 2.2 to see the binary equivalent of all these digits.

Let us take an example to explain the conversion procedure. If we want to convert $(A3D8)_{16}$ to its equivalent binary; we need to follow these steps:

- Step 1: Write the given hexadecimal number as shown in Figure 2.20.
- Step 2: Extract the binary equivalent of each digit of the given hexadecimal number from Table 2.2 and write it as shown in Figure 2.20.
- Step 3: Read the binary number from left to right to obtain the binary equivalent. Hence, $(A3D8)_{16} = (1010001111011000)_2$.

2.3.12 Binary to Hexadecimal Conversion

To convert a number from binary to hexadecimal, we can follow two procedures.

2.3.12.1 Procedure 1

With Procedure 1, we follow a two-step process and require two conversions:

- Step 1: Convert the binary number to its equivalent decimal number;
- Step 2: Convert the decimal number obtained above to hexadecimal.

The above procedure is quite lengthy. Hence, we follow Procedure 2 here.

2.3.12.2 Procedure 2

This procedure is quite simple and easy to follow. Let us take an example to understand how this procedure works.

EXAMPLE

Convert $(10110110100010)_2$ to its hexadecimal equivalent.

Solution

We need the following steps to find the hexadecimal equivalent.

- Step 1: Figure 2.21 shows the first step. Divide the binary number into groups of four bits each (from right to left) by adding 0 bits for completing the groups (if needed).
- Step 2: Figure 2.21 shows the next step and is the final one to get our hexadecimal equivalent. Replace each group by its hexadecimal equivalent following Table 2.1. Hence, $(10110110100010)_2 = (2DA2)_{16}$.

Now consider a fractional binary number. To convert it to hexadecimal, use the following steps:

- Step 1: Divide the binary number before the binary point into groups of four bits each (from right to left) and after the binary point into groups of four bits each (from left to right) by adding 0 bits for completing the groups (if needed);
- Step 2: Replace each group with its hexadecimal equivalent following Table 2.2.

EXAMPLE

Convert $(101010011011.1010001)_2$ to its hexadecimal equivalent.

Solution

The solution procedure is shown in Figure 2.22 and is self-explanatory. Hence, $(101010011011.1010001)_2 = (A9B.A2)_2$.

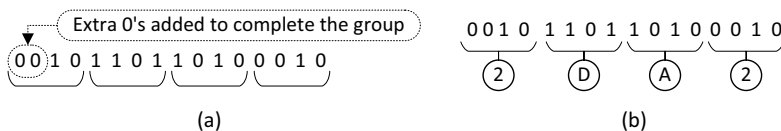


FIGURE 2.21 Short-cut method to convert binary to hexadecimal.

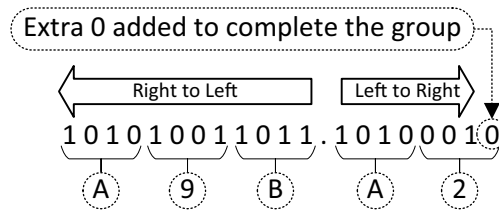


FIGURE 2.22
Fractional binary number to hexadecimal conversion.

2.4 Review Questions

1. What is the difference between a positional number system and a non-positional number system?
2. Why does the computer need a binary number system to represent numbers?
3. If a computer uses a binary number system, what is the need for hexadecimal and octal number systems?
4. Write short notes on:
 - a. The binary number system;
 - b. The octal number system;
 - c. The hexadecimal number system.
5. Describe the generalized procedure for converting a number with any base to its decimal equivalent. Explain with an example.
6. Can you derive a generalized procedure to convert a decimal number to any other number with a diverse base? If yes, then outline the procedure and explain it with an example.
7. Can you design a number system with base 6? What are the different symbols required to make this number system? How can you convert a number with base 6 to decimal and vice versa?

2.4.1 Conversion Questions

1. Convert the following binary numbers to their equivalent decimal numbers.
 - a. 10001001_2
 - b. 10101_2
 - c. 1101_2
 - d. 110.110_2
 - e. 11.001_2
 - f. 1100.101_2
2. Convert the following decimal numbers to their equivalent binary numbers.
 - a. 78_{10}
 - b. 653_{10}
 - c. 8792_{10}
 - d. 98.23_{10}
 - e. 987.34_{10}
 - f. 9832.356_{10}

3. Convert the following binary numbers to their equivalent octal numbers.
 - a. 110110110111_2
 - b. 1010101110011_2
 - c. 101011100_2
 - d. 10101.1100100_2
 - e. 10101.110110_2
 - f. 1101010.110101110_2
4. Convert the following octal numbers to their equivalent binary numbers.
 - a. 1324_8
 - b. 765_8
 - c. 52712_8
 - d. 43.456_8
 - e. 32.776_8
 - f. 254.366_8
5. Convert the following binary numbers to their equivalent hexadecimal numbers.
 - a. 110010010100_2
 - b. 1010101001_2
 - c. 11000100101_2
 - d. 10001.10101_2
 - e. 1111.010101_2
 - f. 1101001.101010111_2
6. Convert the following hexadecimal numbers to their equivalent binary numbers.
 - a. $A2B5_{16}$
 - b. $D9C34_{16}$
 - c. $FF2A3B9_{16}$
 - d. $2FE.34_{16}$
 - e. $7A3.AD2_{16}$
 - f. $786A.AA2E_{16}$
7. Convert the following decimal numbers to their equivalent octal numbers.
 - a. 657_{10}
 - b. 7578_{10}
 - c. 543543_{10}
 - d. 987.85_{10}
 - e. 5454.2323_{10}
 - f. 4568.534_{10}
8. Convert the following octal numbers to their equivalent decimal numbers.
 - a. 3232_8
 - b. 14_8
 - c. 6453_8
 - d. 443.7564_8
 - e. 34243.75_8
 - f. 33545.3232_8
9. Convert the following decimal numbers to their equivalent hexadecimal numbers.
 - a. 37_{10}
 - b. 7448_{10}
 - c. 94323_{10}
 - d. 977.85_{10}

- e. 5324.2323_{10}
 - f. 4348.124_{10}
10. Convert the following hexadecimal numbers to their equivalent decimal numbers.
- a. $A23_{16}$
 - b. FAB_{16}
 - c. $AB43_{16}$
 - d. $AA.34B_{16}$
 - e. $EA3.56F_{16}$
 - f. $EA67.9E_{16}$
11. Convert the following octal numbers to their equivalent hexadecimal numbers.
- a. 323_8
 - b. 7544_8
 - c. 7263_8
 - d. 241.66_8
 - e. 644.132_8
 - f. 22.7554_8
12. Convert the following hexadecimal numbers to their equivalent octal numbers.
- a. $F3A_{16}$
 - b. $FEDAB_{16}$
 - c. $F45BA_{16}$
 - d. $FE3.45A_{16}$
 - e. $BC.459A_{16}$
 - f. $BBDA.456_{16}$

3

Problem Solving through Flowcharts and Algorithms

3.1 Introduction

Now that we understand what a computer is, it's time to solve problems using the computer. We will begin by introducing the problem-solving techniques progressed through algorithm writing and flowchart drawing. To learn how to code always starts with understanding the problem well, drafting it on paper through a step-by-step method called an algorithm, and finally, drawing the flowchart to show how the input flows through the solution steps and produces the output. Figure 3.1 shows the essential steps needed before we write C programming code. Hence, we recommend the reader, before proceeding to write programs, to follow these steps.

This chapter provides a detailed description of algorithm writing and flowchart drawing. I have tried to include a sufficient number of examples that describe all the concepts involved in writing algorithms. Once the reader knows how to write algorithms, flowchart drawing is so much easier. After completing this chapter, the reader will be able to answer the following:

1. What is problem solving, and how to approach it?
2. What is an algorithm, and how to write an algorithm to solve a given problem?
3. What is a flowchart, and how do you draw one?
4. What are the various symbols used in drawing a flowchart?
5. What is the relationship between an algorithm and a flowchart?

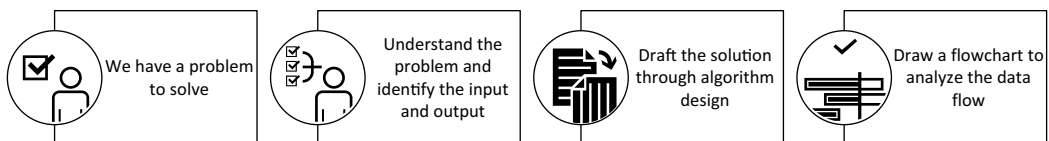


FIGURE 3.1
Essential steps before writing a C program.

3.2 Problem-solving Approach

A programming problem is always challenging to solve because the computer is a powerful yet feeble-minded machine. It can do effortless tasks, but with a proper accuracy and faster than any human being. Suppose a problem is given to us to find out the factorial of a number. Any normal human being will require a certain amount of time to provide you with the answer, but if you solve the same task using a computer, it instantly yields the answer. Interestingly a computer cannot do this on its own. So we need to describe with proper steps how to find the factorial of a number. Because a computer does not know how to find the factorial, but knows how to perform basic arithmetic operations like addition, subtraction, multiplication, and division, it is the programmer's job to write the step-by-step instructions that need to be followed by the computer to find the required result (the factorial of a number).

From the above discussion, we can see that we should adequately plan the solution for a given problem using problem-solving techniques. A problem-solving method is constituted of five phases. Figure 3.2 shows the steps that include all these five phases, and we should follow these to solve any given problem.

1. *Understanding the problem*: Unambiguously describe the problem for easy understanding.
2. *Identifying the problem requirement*: Identifying the inputs to be given and what the expected output is.
3. *Making a plan of the solution*: To design an algorithm and flowchart.
4. *Implement the plan*: Convert the algorithm into a program for implementation using any programming language.
5. *Test and verify*: Execute the program and inspect the output.

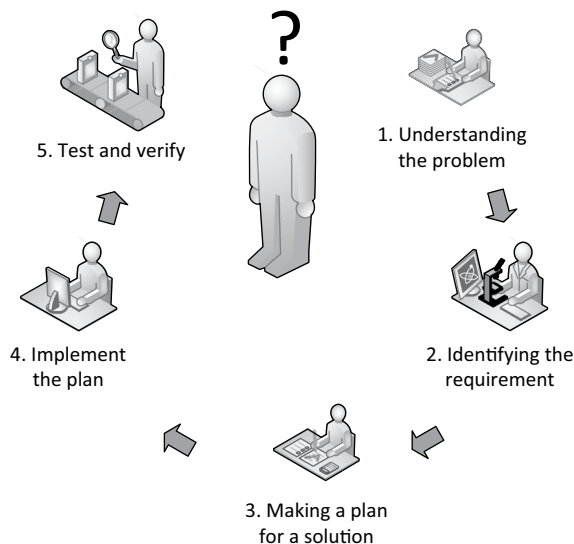


FIGURE 3.2
Problem-solving phases.

In the next section, we are going to explain the detail of algorithm design followed by drawing flowcharts. During the algorithm design, phases 1 and 2 automatically come into the discussion. Section 3.5 will introduce the concept of programming and the different components involved in executing them.

3.3 Algorithm Design

Before we jump into the definition of an algorithm, let us look at the image shown in Figure 3.3. We have taken this image from a cook book by Sanjeev Kapoor. The objective of using this image is to introduce you to different features of an algorithm. We are not here to learn cooking; instead, the picture will help you to understand how to write unambiguous statements in making a recipe. The procedure written in the image has a high resemblance to algorithm design.

If you analyze the recipe, you can easily see that the steps are well written, and whoever follows the steps will be able to cook a tasty “crisp pohe”. No actions are ambiguous, and the procedure mentions all the ingredients required to prepare the recipe. We can say that the method specified in the image is a step-by-step method to make the food, and the ingredients are known as the input for processing. Finally, the “crisp pohe” will be the output of this process.

Now we are ready to explain the algorithm. By following the similar analogy discussed above, we can define an algorithm as follows:

Crisp Pohe

(Kurkuri Pohe)

Ingredients	
4 cups	Nylon Poha
2 tbsp	Ghee
1 cup	Coconut (scraped)
3	Green chillies (finely chopped)
½ tsp	Cumin seeds (coarsely powdered)
1 tsp	Coriander seeds (coarsely powdered)
1 tbsp	Sugar
Salt to taste	

Method

- Heat ghee in a kadai and gently roast the nylon poha over low heat till they turn crisp.
- Mix together scraped coconut, chopped green chillies, powdered cumin and coriander seeds, sugar and salt well.
- Add this to the roasted poha and toss gently.
- Serve immediately as the poha becomes soggy if left for long.

FIGURE 3.3
A cooking recipe.

An algorithm is a sequence of unambiguous instructions for solving a problem in a finite amount of time.

According to the programmer's perspective, we write the algorithm to solve a given programming task that can easily be converted into a program statement irrespective of any programming language. For example, let the job be to add two numbers; then how do we write an algorithm for that? Following the problem-solving technique, we need to analyze and identify the input to our algorithm and what should be the output produced by it. In our case, the algorithm needs two numbers, and it will provide their addition as the result. After that, we need to write the step-by-step procedure to perform the addition operation. Every step must instruct the computer to do something, and upon completion of all the steps, the machine should produce the required result. The complete algorithm to solve the addition of two numbers will be discussed in the subsequent section.

3.3.1 Characteristics of an Algorithm

According to [1], every algorithm must conform to the following characteristics:

1. *Algorithms are well-ordered*: Every solution needs a specific order of execution, and so the algorithm must preserve this order in its steps.
2. *Algorithms have unambiguous operations*: Every step of an algorithm performs a distinct operation and is unambiguous.
3. *Algorithms have effectively computable operations*: Suppose we want to write steps for a multiplication operation and assume that our computer doesn't have a multiplier unit, then we should not use multiplication symbols in our steps. We may use a repetitive addition operation to perform multiplication. That means every step in the algorithm contains an action that is possible to do.
4. *Algorithms produce a result*: This is evident because we are writing an algorithm that processes inputs to produce an output.
5. *Algorithms stop after a finite amount of time*: Every algorithm eventually stops execution after it produces an output.

Every algorithm must satisfy the following criteria:

1. *Finiteness*: This implies that the algorithm must have a finite number of steps.
2. *Definiteness*: Each step must be clear and unambiguous.
3. *Input*: An algorithm must receive some inputs for its operation.
4. *Output*: At least one output must be produced from the algorithm.
5. *Effectiveness*: This implies that all the operations involved in an algorithm must be sufficiently basic in nature so as to be carried out manually in a finite interval of time.

3.4 Basics of an Algorithm

There is no strict rule followed during algorithm writing, and algorithms written by two people may differ from each other, even if they both write it to solve the same problem. In this book, we will use some basic symbols and control structures to make the writing style uniform:

1. *Assignment symbol* (\leftarrow): This is used to assign a value to a variable. Sometimes the “=” symbol is also used for this purpose.
2. *Relational symbols, arithmetic symbols, and comments*: see Table 3.1.

For beginners, these symbols may be unusual, but they are used every day by C programmers. Most of the symbols are self-explanatory, but note:

- For “double equal to”, suppose $a = 5$ and $b = 5$, then $a == b$ evaluates to true. That means the value of a and b are the same.
- As we are about to convert our algorithms to C programs, we need to understand the difference between the divide ($/$) and the remainder ($\%$) symbol. Suppose $a = 10$ and $b = 3$, then a/b evaluates to 3, and $a\%b$ evaluates to 1.
- Sometimes we need a comment line for documentation, and we write our comment following the $//$ symbol.

EXAMPLE

- Step 1: START // Start of Algorithm
 Step 2: Set $N \leftarrow N + 1$ // Increase the value of N by 1

3. *Control statements*: While writing an algorithm, we need the following three necessary control structures. Generally speaking, a control structure controls the flow of instruction execution depending on certain conditions. During the process of problem

TABLE 3.1
 Symbols Used in Algorithm Writing

Symbol	Descriptions	Symbol	Descriptions
<	Less than	==	Double equal to (similarity checking)
<=	Less than or equal to	+	Plus
>	Greater than	-	Minus
>=	Greater than or equal to	*	Multiplication
≠OR !=	Not equal to	/	Division
//	Comment line	%	Remainder

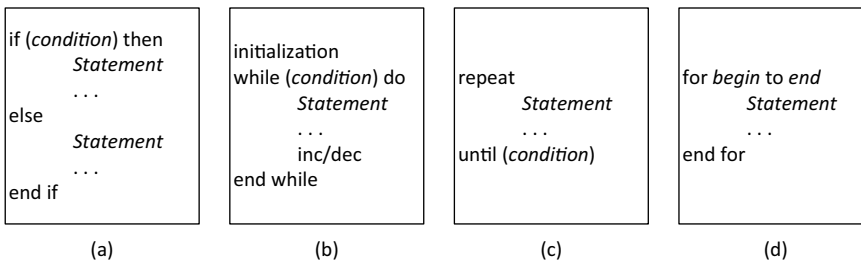


FIGURE 3.4
Syntax of control statements.

solving and algorithm writing, we may come across situations where the execution of a particular step relies on the satisfiability of other conditions. Then we need a control structure:

- *Selection*(if-then-else): used for executing a statement or a set of statements whenever a particular statement is true. Figure 3.4 shows the syntax for writing if-then-else control statements.
- *Looping*(while-do, repeat-until, for): used for executing a statement or a set of statements multiple times while a condition evaluates to true. Refer to Figure 3.4 for the syntax of writing these..

3.4.1 Advantages of Using an Algorithm

1. It provides a step-by-step solution to a problem and is easy to understand.
2. It provides a better way to approach the solution of a problem and makes it easy to find an error in solution steps.
3. It is independent of any programming language. After writing the algorithm, we can easily convert it into program statements irrespective of any specific computer language.
4. It provides better documentation.

In summary, an algorithm is a collection of well-ordered computational steps that take an input or a set of inputs and produces an output or a set of outputs. Let us take an example problem and write an algorithm to solve it.

3.4.2 Example: Write an Algorithm to Add Two Numbers and Produce the Sum

The first step is to analyze the problem and identify the inputs and the possible outputs. Our algorithm requires two numbers as input. Naturally, after addition, it will produce their sum. Thus, the output is also another number. Let *a* and *b* represent the input that holds two numbers and *r* be another variable that stores the sum of *a* and *b*.

Input: *a* and *b*

Output: *r*

Hence the following algorithm shows the solution steps. As our book focuses on the C programming language, we are trying to write the algorithm in such a manner that we can replace each step of the algorithm with the corresponding C code.

Algorithm 3.1

ADD-TWO-NUMBERS

1. Start
2. Read `a, b` // Read two numbers from the user and store it in `a` and `b`
3. Set `r ← a+b` // Add the value of `a` and `b` and store it in `r`
4. Print `r` // Show or print the result on screen
5. Stop

We will follow some conventions while writing the algorithm for easy reading. We know that, later, we will use this algorithm to draw a flowchart and convert each step to its corresponding C code. Hence, these conventions will help.

1. Every algorithm begins with a “start” step and ends with a “stop” step.
2. Before writing the algorithm, we must identify its input(s) and output(s), which are later used by our C code.
3. We should write the algorithm in such a manner that we can easily convert it into its corresponding program statements.

3.5 Flowcharts

A flowchart is a kind of diagram that represents the process flow or the sequence of operations needed to solve a given problem. It provides a more straightforward method to understand how the data flows through the solution steps and produces an output. The symbols used in the flowchart are simple and easy to learn. A programmer always prefers to draw a flowchart before writing a computer program. There are several definitions that describe a flowchart, but we will provide you with the programmer’s view.

A flowchart is a structured approach to represent the solution steps outlined in an algorithm with the help of simple and easy-to-learn symbols that show how the input(s) is/are processed through the system and produces the output(s).

3.5.1 Advantages of Using a Flowchart

1. *Easier to understand the program logic:* It provides a better way to understand the program logic.

2. *Provides better analysis and maintenance:* A flowchart provides an effortless analysis of the sequence of steps, so more effort can be put into considering the logic of the whole process.
3. *Proper documentation:* Like an algorithm, a flowchart also serves as an appropriate paper document and helps in documenting the entire flow.
4. *Better coding:* A flowchart acts as a guide or a blueprint during actual program writing.
5. *Proper debugging:* The flowchart helps in the debugging and error finding processes.

3.5.2 Flowchart Symbols

Figure 3.5 shows the basic flowchart symbols used in drawing the flow of execution of any algorithm. A description of all these symbols is presented below.

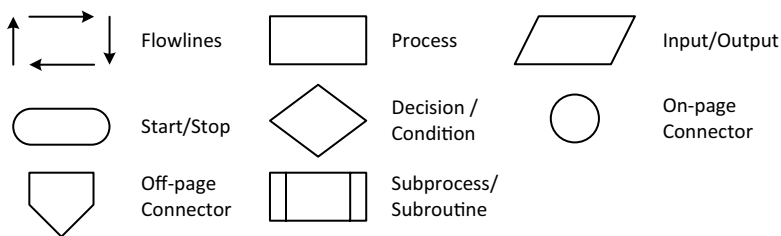


FIGURE 3.5
Flowchart symbols.

FLOW LINES: Show the direction of flow of data or control.

PROCESS: This is used to represent all the processing statements.

INPUT/OUTPUT: This symbol is used when anything is given as an input to the flowchart and anything that is produced as an output from the flowchart.

START/STOP: This symbol is used to specify the start and end of a flowchart. Only one start and end symbol can be on the flowchart.

CONDITION: This symbol is used to specify the different conditions for decision making.

ON-PAGE CONNECTOR: Used to connect remote flowchart portions on the same page.

OFF-PAGE CONNECTOR: Used to connect remote flowchart portions on different pages.

SUBPROCESS/SUBROUTINES: This symbol identifies a separate flowchart segment (module).

3.5.3 Flowchart Drawing Guidelines

1. In drawing a proper flowchart, all necessary requirements should be listed out in logical order.

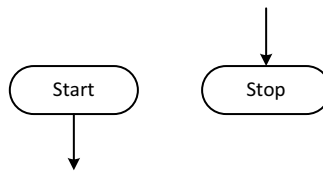


FIGURE 3.6
Flowlines with terminal symbols.

2. The flowchart should be clear, neat, and easy to follow. There should not be any room for ambiguity in understanding the flowchart.
3. The usual direction of the flow of a procedure or system is from left to right or top to bottom.
4. Only one flow line (see Figure 3.6) is used in conjunction with the terminal symbol.
5. If the flowchart becomes complex, it is better to use connector symbols to reduce the number of flow lines. Avoid the intersection of flow lines if you want to make it a more effective and better way of communication.
6. Ensure that the flowchart has a logical *start* and *finish*.

It is useful to test the validity of the flowchart by passing simple test data through it.

Now that we understand the detailed procedure of how to produce a flowchart, let us take one example problem and draw a flowchart for it. We will take the example problem from Section 3.4.2: the addition of two numbers and production of the sum. The detailed algorithm is known to us. Our objective is to sketch the steps using the symbols of the flowchart. Before drawing, we need to identify which step corresponds to which symbol (see Figure 3.7).

In the next section, we will take a set of potential problems and solve them using algorithms and flowcharts to understand the other features of this problem-solving technique (algorithms and flowcharts).

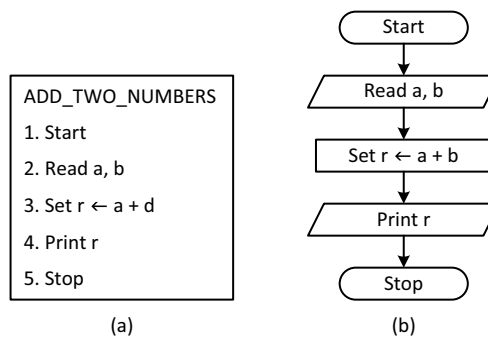


FIGURE 3.7
Algorithm and flowchart for the addition of two numbers.

3.6 Example Problems

EXAMPLE 1

Write an algorithm to find out the area of a rectangle, display the result, and draw its corresponding flowchart.

Solution

Algorithm 3.2

Input: Length (l) and breadth (b) of the rectangle

Output: Area (area) of the rectangle

AREA-RECTANGLE

1. Start
2. Read l, b
3. Set $\text{area} \leftarrow l \times b$
4. Print area
5. Stop

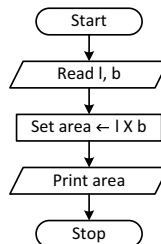


FIGURE 3.8

Flowchart to find the area of a rectangle.

Figure 3.8 shows the corresponding flowchart.

You can see that the above example is very similar to the example discussed in Section 3.4.2.

EXAMPLE 2

Write an algorithm to find the largest number of two numbers and draw its flowchart.

Solution

To solve this type of problem, we need a relational operator like "<" or ">" and a decision-making statement of the type *if-then-else*.

Algorithm 3.3

Input: Two numbers a and b

Output: Display the bigger number, either a or b

BIGGER-AMONG-TWO

1. Start
2. Read a, b
3. **if** a > b **then**
4. Print "a is Big"
5. **else**
6. Print "b is Big"
7. **end if**
8. Stop

Steps 1 and 2 of the algorithm do not require any explanation. In step 3, we compare the value of a and b with an if-else construct. If the statements evaluate to true, then step 4 will be executed; otherwise, step 6 will be executed. In this algorithm, we use the if-else decision control statement, and we have a flowchart symbol that exists for the same, described in section 3.5.2. Figure 3.9 shows the corresponding flowchart.

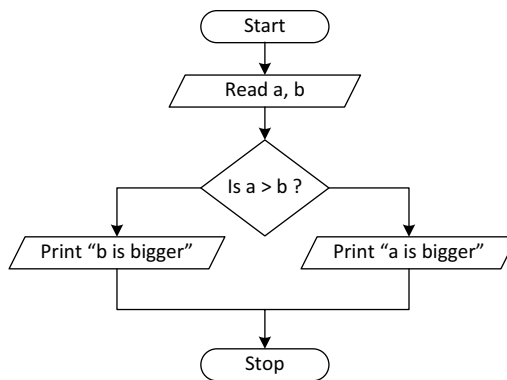


FIGURE 3.9
Flowchart to find the largest number of two numbers.

EXAMPLE 3

Write an algorithm and draw a flowchart to print the book name five times.

Solution

To solve the problem, we can adopt one of two approaches. We can write five print statements to print the book name in the first approach; Algorithm 3.4 shows the solution steps for this.

Algorithm 3.4*Input:* No input required*Output:* Display the book name five times

PRINT-BOOK-NAME

1. Start
2. Print "C Programming Learn to Code"
3. Print "C Programming Learn to Code"
4. Print "C Programming Learn to Code"
5. Print "C Programming Learn to Code"
6. Print "C Programming Learn to Code"
7. Stop

Algorithm 3.4 is correct, and it produces the result as expected. But, what if you are asked to print the book name 100 times? Then we need to write the same step (Print "C Programming Learn to code") 100 times, which is cumbersome. Hence the alternative is to use control statements like *while-do* or *for*. Algorithms 3.5 and 3.6 show the solution steps that use a *while-do* and a *for* control statements, respectively. Figure 3.10 shows the corresponding algorithm.

Algorithm 3.5*Input:* No input required*Output:* Display the book name five times

PRINT-BOOK-NAME

1. Start
2. Set $i \leftarrow 1$ //Initialize a variable i with value 1
3. **while** $i \leq 5$ **do**
4. Print "C Programming Learn to Code"
5. Set $i \leftarrow i+1$ //Increasing the value of i
6. **end while**
7. Stop

Algorithm 3.6*Input:* No input required*Output:* Display the book name five times

PRINT-BOOK-NAME

1. Start
2. Set $i \leftarrow 1$ //Initialize a variable i with value 1
3. **for** $i \leftarrow 1$ **to** 5
4. Print "C Programming Learn to Code"
5. **end for**
6. Stop

EXAMPLE 4

Write an algorithm and draw a flowchart to find out the sum of the digits of a number. (Hint: suppose the number given is 365, then the sum of the digits are $3 + 6 + 5 = 14$.)

Solution

To solve the above problem, we need three different variables: N, R, and Sum. N is used to store the number. We use R to store individual digits, and the variable Sum will hold the result of the addition. Now the question is how to extract individual digits from the given number. To do that, we will use the remainder operator (%) and divide the given number by 10. That will provide us with the last digit.

Initially, we will divide the given number by 10 and store the remainder for further calculation. We will use the quotient of the previous division as the input for the next iteration. Continue in this way, and we will stop when the quotient becomes 0. We will add the remainder collected from each iteration to get the answer. The entire process is shown in Figure 3.11. Algorithm 3.7 shows the solution steps, and Figure 3.12 shows its corresponding flowchart.

Algorithm 3.7

Input: A random number (N)

Output: Addition result (sum) of all the digits of the given number N

ADD-DIGIT

1. Start
2. Read N
3. Set $\text{sum} \leftarrow 0$
4. **while** $N \neq 0$ **do**
5. Set $R \leftarrow N \% 10$
6. Set $\text{Sum} \leftarrow \text{Sum} + R$
7. Set $N \leftarrow N / 10$
8. **end while**
9. Print sum
10. Stop

The algorithm requires a number as input, which is N here (Algorithm 3.7, step 2). The number may constitute one or more digits. This will produce the sum of all the digits of N, which is the output of this algorithm (sum). We initialize the sum to 0 in step 3 because, later, our algorithm extracts the digits one-by-one (using the while loop iterations) from N and adds it to the sum. Steps 4 to 8 perform the operation described in Figure 3.11. Every iteration proceeds through a condition checking " $N \neq 0$." The execution of this algorithm stops when the condition becomes false. Initially, step 5 extracts one digit from N and assigns it to R (remainder). Step 6 adds the value obtained in step 4 to the variable sum, which was previously 0. Finally, step 7 reduces N by one digit by dividing the number N by 10. If you analyze these operations, you will notice

that if N is a three-digit number, then there will be three iterations needed to add all the digits. Similarly, a four-digit number requires four iterations, and an n -digit number requires n iterations. After the condition in step 4 evaluates to false, our algorithm will execute step 9 and produce the result, which is the addition of all the digits of N . Figure 3.12 shows the flowchart.

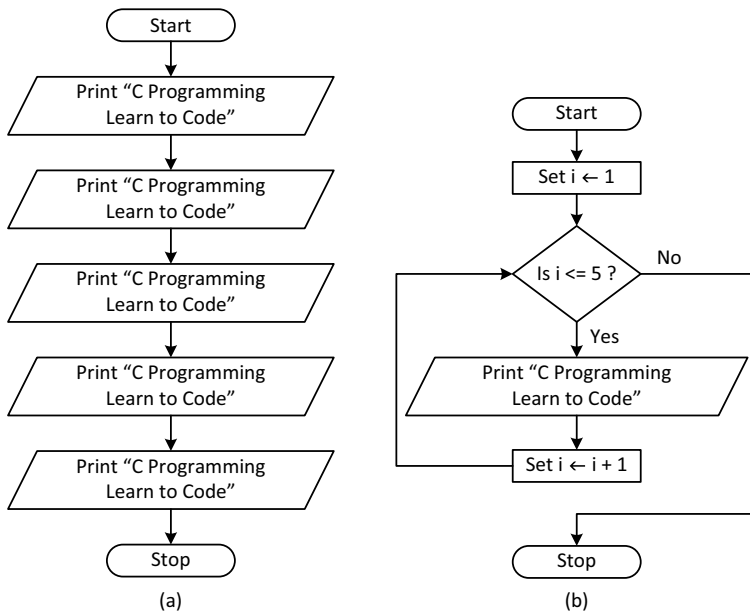


FIGURE 3.10
Flowchart of algorithms 3.5 and 3.6.

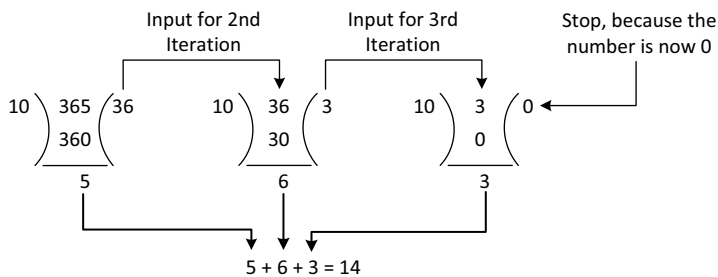


FIGURE 3.11
Analyzing the solution with an example.

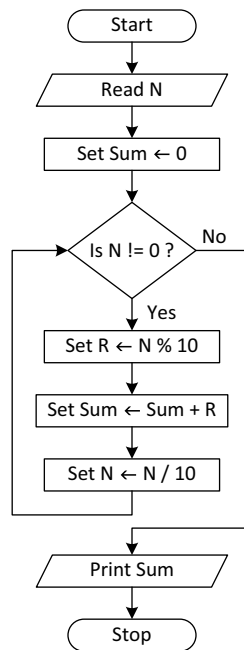


FIGURE 3.12
Flowchart showing the additions of the digits of a number.

3.7 Basics of a Programming Language

According to the phases of problem-solving (Section 3.2), after the development of the algorithm and flowchart, the next stage is implementation using a programming language. This section will discuss the different types of programming languages that can be considered for this purpose.

Let us first introduce what a program is. A *program* is a set of logically related instructions that are arranged in a sequence and guide the computer to solve a problem. The process of writing a program is called *programming*.

A standard programming language is used to write a computer program. Programming languages can be classified into two types:

1. Low-level languages;
2. High-level languages.

3.7.1 Low-level Languages

A computer cannot understand instructions given in high-level language or the language used by humans. The computer only understands the language of 0's and 1's, which is nothing but a low-level language.

There are two types of low-level languages:

1. Machine-level languages;
2. Assembly-level languages.

3.7.1.1 Machine-level Languages

This is a sequence of instructions written in the form of binary numbers consisting of 1's and 0's to which a computer responds directly. Machine languages are the only languages understood by computers.

An instruction in a machine-level language consists of two parts:

1. OPCODE (operation code);
2. OPERANDS (addresses or locations).

Operation code tells the computer what function must be performed, e.g., addition, subtraction. Operands are the memory locations where the values are stored upon which the operation will be done.

EXAMPLE

0000	0111	Load A register with value 7
0010	1010	Load B register with value 10
1000	0010	$A \leftarrow A + B$
0111	0110	Halt Processing

Advantages

The computer can execute programs written in machine language very fast because the instructions are understood directly by the computer, and no translation is required.

Disadvantages

1. Machine dependent: the user must know the internal design of the computer;
2. Difficult to use;
3. Error prone;
4. Difficult to debug.

3.7.1.2 Assembly-level Languages

In the case of assembly-level languages, instead of binary codes, some mnemonic codes are used. A mnemonic code is nothing but a symbolic representation, or we can say it is a combination of letters, digits, or special characters that are used instead of binary codes.

For example, instead of using code 1000 for addition, if we use the symbol "ADD," it will be more readable.

EXAMPLE

```
LDA    A, 7           Load A register with value 7
LDA    B, 10          Load B register with value 10
ADD    A, B           A ← A + B
HLT    Halt Processing
```

The machine cannot directly execute an assembly language program as it is not in binary form. In some way, it should be transferred to binary form (i.e., machine language), which is done by an assembler.

An *assembler* is a program that translates an assembly language program into a machine language program. A program written in assembly language is called source code. The assembler converts the source program into a machine-language program known as object code (Figure 3.13).

Advantages

1. Easy to use and easier to understand as compared to machine-level language;
2. Easy to locate errors and correct them;
3. Assembly language has the same execution efficiency as machine-level language because it is a one-to-one translator between the assembly language program and its corresponding machine language program.

Disadvantages

1. One of the significant drawbacks is that assembly language is machine-dependent. A program written for one computer might not run on other machines with different hardware configurations.
2. A conversion program is needed to convert the assembly language program to a machine language program.

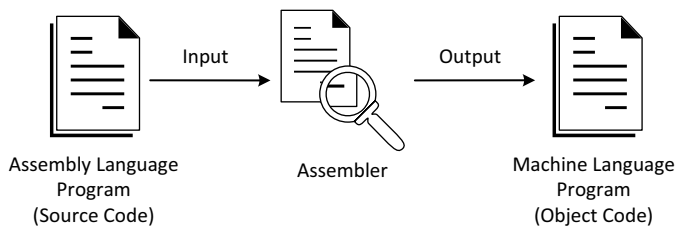


FIGURE 3.13
The operation of an assembler.

3.7.2 High-level Languages

According to the above discussion, assembly languages and machine-level languages require in-depth knowledge of computer hardware, and both languages are machine-dependent. But a high-level language enables the programmer to concentrate on program logic rather than computer hardware.

A high-level language is simply an English-like language that uses some mathematical symbols for program construction. This makes the program easier to read, understand, and manipulate. High-level languages are also called problem-oriented languages because the instructions are suitable for solving a problem. Like assembly-level language, high-level language needs to be translated to machine-level language for understanding by the computer. So, to convert a high-level language to machine-level language, we need a program known as a compiler or an interpreter.

A *compiler* is a program that translates a high-level language program into a machine language program. A program written in a high-level language is called *source code*. The compiler converts the source program into a machine language program known as *object code* (Figure 3.14).

We use several high-level languages today. Some examples are C, C++, JAVA, Python, Cobol, and Fortran. For every language, there is a compiler that translates the high-level code to machine code (Figure 3.15).

An *interpreter* is a program that also translates a high-level language program into a machine language program, but translates it one line at a time. There are many interpreter languages used today, e.g. VB, Perl, BASIC.

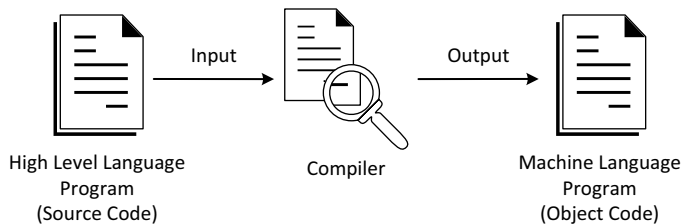


FIGURE 3.14
Function of a compiler.

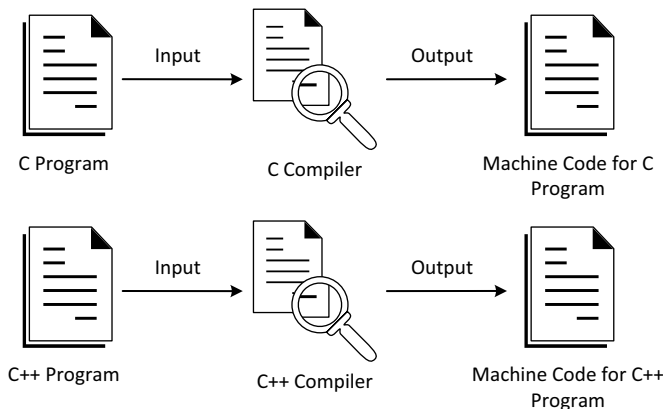


FIGURE 3.15
Different compilers for different languages.

3.7.2.1 Compiler vs. Interpreter

1. A compiler compiles the whole program at once, but an interpreter interprets one line at a time;
2. After compilation, the compiler produces a list of errors, but an interpreter stops after the first error;
3. Program execution is faster with the compiler, but with the interpreter, program execution is slower.

3.7.2.2 Advantages

1. Easier to learn and implement because it is based upon commonly used language like English;
2. Machine independent;
3. Easy to find errors and modify.

3.8 Review Questions

3.8.1 Objective Type Questions

1. An algorithm depends upon a programming language. True/false?
2. _____ is a sequence of unambiguous instructions for solving a problem in a finite amount of time.
3. _____ is a graphical representation showing the flow of control among the steps in a program, people in an organization, or pages of a presentation.
4. _____ symbol is used to specify a condition in a flowchart.
5. Algorithm development differs from person to person. True/false?
6. A high-level language is a machine-independent language. True/false?
7. A _____ compiles the whole program at once but an _____ interprets one line at a time.
8. An instruction in machine-level language consists of two parts: _____ and _____.
9. A _____ is a set of logically related instructions that are arranged in a sequence and guide the computer to solve a problem.
10. The process of writing a program is called _____.

3.8.2 Practice Problems

1. Write an algorithm to find out the area of a circle and draw its corresponding flowchart.
2. Write an algorithm to calculate simple interest and draw its corresponding flowchart.

3. Write an algorithm to find the average of three numbers and draw its flowchart.
4. Write an algorithm to find the area of a right-angled triangle and draw its flowchart.
5. Write an algorithm to swap the value of two variables and draw its flowchart.
6. Write an algorithm to check whether a number is an even or odd number and draw its flowchart. (*Hint*: the number must be divided by 2, and if the remainder is 0, then that number is even, otherwise that number is odd.)
7. Write an algorithm to find the biggest number among three numbers and draw its flowchart.
8. Write an algorithm to check whether a student is a pass or fail. Input marks of five subjects out of 100. Find the average score. If the average score is greater than or equal to 50%, then print "YOU ARE A PASS" or else print "YOU ARE A FAIL". Draw the corresponding flowchart.
9. Write an algorithm and draw a flowchart to find out the biggest number among four numbers.
10. Write an algorithm and draw a flowchart to swap the value of two different numbers.
11. Write an algorithm and draw a flowchart to calculate simple interest.
12. Write an algorithm and draw a flowchart to check whether a number is a palindrome or not.
13. Write an algorithm and draw a flowchart to check whether a number is a prime number or not.
14. Write an algorithm and draw a flowchart to check whether a number is a perfect number or not.
15. Write an algorithm and draw a flowchart to reverse a number.
16. Write an algorithm and draw a flowchart to check whether a number is an Armstrong number or not.
17. Write an algorithm and draw a flowchart to print all the even numbers present within a range. The lower range and the higher range must be supplied to the algorithm.
18. Write an algorithm and draw a flowchart to find out the factorial of a number.
19. Write an algorithm to compute the sum of the squares of integers from 1 to 50 and also draw the corresponding flowchart.
20. Write an algorithm to read a number N from the user and print all its divisors.
21. Write an algorithm and draw a flowchart to find out the summation of $1 + 2 + 3 + \dots + n$.

3.8.3 Subjective Questions

1. What is an algorithm?
2. What is a flowchart?
3. What are the characteristics of an algorithm?
4. Describe the different symbols used in a flowchart.
5. What is the difference between a compiler and interpreter?
6. What is the difference between a high-level language and a low-level language?

7. Define the following terms:
 - a. Program;
 - b. Assembler;
 - c. Compiler;
 - d. Interpreter.

Reference

1. Schneider, G. Michael, and Judith Gersting. Invitation to computer science. *Cengage Learning*, 2018.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

4

Introduction to C Programming

4.1 Introduction

This chapter introduces the overall features of the C programming language, why we should use this language, what its characteristics are, and a short description of its history. I will also explain what the structure of a C program is, and how to execute it in different environments.

The Tiobe programming community index is a measure of the popularity of programming languages. According to the index for August 2021, C programming (with a rating of 12.57%) is at the top of the list and is treated as the most popular language. I believe that learning C programming acts as a building block for learning other high-level languages. C will always stay ahead of other popular languages like Java and Python due to its completeness and low-level programming capability.

We can use C to solve every type of programming problem. It can facilitate the low-level requirements of a programmer or high-level specifications. There are several discussions on whether C is a high-level language or a low-level language. It has the capability to access the system's low-level functions as well as enabling us to code most of the high-level specifications. Hence, many programmers assume that C is a middle-level language. But you can find many books and much literature where C is mentioned as a high-level language. We are not here to decide this issue; rather, we will focus on the different features of this language and learn how to code with it.

- Most high-level languages (e.g., FORTRAN) provide everything the programmer might want to do already build into the language;
- A low-level language provides nothing other than access to the machine's basic instruction set;
- A middle-level language, such as C, probably doesn't supply all the constructs found in high-languages, but it provides you with all the building blocks that you will need to produce the results you want.

The advantages of using C are:

- C is a real-world language, widely available and popular with professionals;
- C is a small, efficient, powerful, and flexible language;
- C has been standardized, making it more portable than some other languages;
- C is close to the computer hardware, revealing the underlying architecture;

- C provides enough low-level access to be suitable for embedded systems;
- C is a high-level language allowing complex systems to be constructed with minimum effort;
- C's modular approach suits large, multi-programmer projects;
- C's use of libraries makes it adaptable to many different application areas;
- The Unix operating system was written in C and supports C;
- C gave birth to C++, widely used for application programming, and, more recently, Java, which is based upon C++;
- Many other languages borrow from C's syntax: for example, Java, JavaScript, and Perl.

4.2 History of C

C came into being in the years 1969–73, in parallel with the early development of the Unix operating system; the most creative period occurred during 1972. The C programming language often called the “white book” or “K&R” [1]. Finally, in the mid-1980s, the language was officially standardized by the ANSI X3J11 committee, which made further changes. Until the early 1980s, although compilers existed for a variety of machine architectures and operating systems, the language was almost exclusively associated with Unix; more recently, its use has spread much more widely, and today it is among the languages most commonly used throughout the computer industry [2, 3].

By 1960 a number of computer languages had come into existence, almost all for a specific purpose. For example, COBOL was being used for commercial applications, and FORTRAN for engineering and scientific applications. At this point people started thinking about developing a common language which could program for all possible applications.

Several languages preceded the development of C. In 1967, Martin Richards developed a language called Basic Combined Programming Language (BCPL). In 1970 Ken Thompson developed a similar language called B. Finally, in 1972 Dennis Ritchie (Figure 4.1)

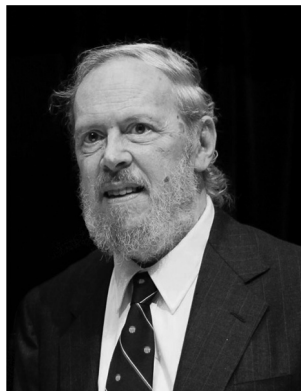


FIGURE 4.1
Dennis Ritchie.

developed C, which took many concepts from BCPL and B and added the concept of data types.

BCPL, B, and C differ syntactically in many details, but broadly they are similar. Programs consist of a sequence of global declarations and function (procedure) declarations. Procedures can be nested in BCPL. B and C avoid this by imposing a more severe restriction: there are no nested procedures at all.

A brief summary of this development is as follows.

1960: ALGOL

- Developed by: international committee;
- Remarks: too general, too abstract.

1963: CPL (Combined Programming Language)

- Developed by: Cambridge University;
- Remarks: hard to learn, difficult to implement.

1967: BCPL (Basic Combined Programming Language)

- Developed by: Martin Richards at Cambridge University;
- Remarks: too specific, could deal with only specific problems.

1970: B

- Developed by: Ken Thompson at AT&T Bell Labs
- Remarks: too specific, could deal with only specific problems.

1972: C

- Developed by: Dennis Ritchie at AT&T Bell Labs
- Remarks: combination of both BCPL and B.

In 1983, the American National Standards Institute (ANSI) established a committee to provide a modern, comprehensive C definition. The result, the ANSI standard, or “ANSI C,” was completed in 1989. ANSI C is sometimes called the C89 standard. In 1990, the ISO (International Organization for Standardization) adopted ANSI C as ISO/IEC 9899:1990, which is sometimes called C90. Therefore, C89 and C90 refer to the same standard. The development did not stop there; several standards were published with new features in subsequent years. A list of C standards and the years they were standardized is shown in Table 4.1 [4–6].

TABLE 4.1
C Standards and Their Year-wise Development

Year	C Standards
1972	Birth of C language
1978	K&R C
1989/90	ANSI C (C89)/ISO C (C90)
1999	C99
2011	C11
2017	C17

4.3 Executing a C Program

When a task is assigned to a programmer, he or she needs to analyze it, prepare an algorithm, and draw flow charts to solve the given task. The next step is to convert the algorithm into a program and execute it to check the results. This section describes the execution process of a program. Developing and executing a program in C requires at least four steps, shown in Figure 4.2:

1. Editing (or writing) the program;
2. Compiling it;
3. Linking it (with functions that are needed from the C library);
4. And finally, executing it.

4.3.1 Editing

You write a computer program with words and symbols that are understandable to human beings. This is the *editing* part of the development cycle. You type the program directly into a window on the screen and save the resulting text as a separate file. This is often referred to as the *source file*. The C program is stored in a file with the extension “.c”.

4.3.2 Compiling

You cannot directly execute the source file. To run on any computer system, the source file must be translated into binary numbers understandable by the computer’s central

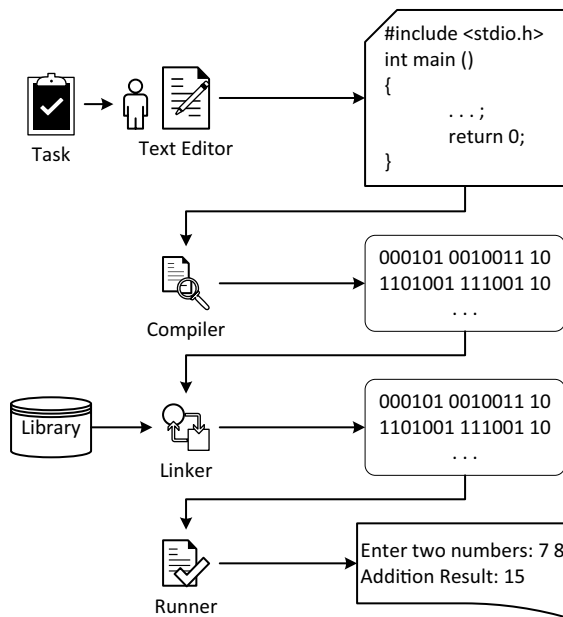


FIGURE 4.2
Executing a C Program.

processing unit. This process produces an intermediate object file – with the extension “.obj”, which stands for “object.”

4.3.3 Linking

The first question that comes to most people’s minds is *why is linking necessary?* The main reason is that many compiled languages come with library routines which can be added to your program. These routines are written by the manufacturer of the compiler to perform a variety of tasks, from input/output to complicated mathematical functions. In the case of C the standard input and output functions are contained in a library (stdio.h), so even the most basic program will require a library function. After linking, the file extension used is “.exe” which stands for executable file.

4.3.4 Executing

Thus the text editor produces “.c” source files, which go to the compiler, which produces “.obj” object files, which go to the linker, which produces “.exe” executable files. You can then run “.exe” files.

4.4 Structure of a C Program

Every C program consists of one or more functions. A function is nothing but a group or sequence of C statements that are executed together to perform a specific task. Each C program function performs a specific task. The entire program will have the structure shown in Figure 4.3.

4.4.1 Documentation

The documentation section consists of a set of comment (remark) lines giving the name of the program, the author, and other details which the programmer would like to use later. Comments may appear anywhere within a program. Such comments are helpful in identifying the program’s principal features or in explaining the underlying logic of various program features. A *single line or multi-line comment* may be specified in C. For single-line comments, we simply use a double forward slash (//) (e.g., //this is a comment). For multi-line comments, we use delimiters starting with /* and ending with */ (e.g., /*this is a comment*/). The C89 standard (ANSI C) only supports multi-line comments. Later on, after the development of the C99 standard, the single line comment was also included.

4.4.2 Header Files

This is also called a link section. These statements instruct the compiler to include C *preprocessors* such as header files and symbolic constants before compiling the C program. Some of the preprocessor statements are:

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

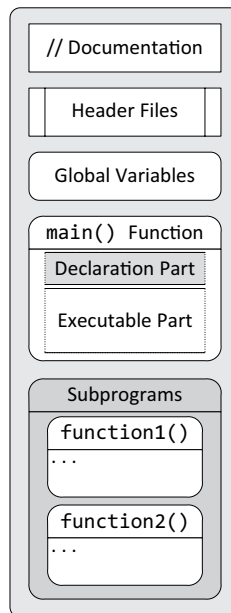


FIGURE 4.3
Structure of a C program.

The lines that begin with # are preprocessed before the compilation starts. It tells the computer to include the contents of all header files like `stdio.h` or `conio.h` to the current program. Then the compilation begins.

4.4.3 Global Variables

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

4.4.4 `main()` Function

Each and every C program should contain only one `main()` function. Execution starts with this `main()` function. The function should be written in small case (lower case) letters, and it should not be terminated by a semicolon. `main()` executes user-defined program statements, library functions, and user-defined functions. All these statements should be enclosed within left and right curly braces

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

4.4.5 Subprograms

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

4.4.6 Your First C Program

We start with a simple C program that prints a line of statements on your computer screen. The code is shown in Program 4.1, followed by the output; an explanation of each line of code is given.

PROGRAM 4.1

```
1. /* Write a program to print a
2. single statement*/
3. #include<stdio.h> //Header File
4. void main()
5. {
6.     printf ("C Programming Learn to Code");
7. } //End of main function
```

Output

C Programming Learn to Code

Lines 1 and 2:

- This line is known as the documentation line. It is also called the comment line.
- Comments are a way of explaining what a program does. They are put between `/*` and `*/`.
- Comments are ignored by the compiler and are used by the user and other people to understand the code.
- You should always put a comment at the top of a program that tells you what it does because one day if you come back and look at it, you might not be able to understand what it does, but the comment will tell you.
- You can also use comments in between your code to explain a piece of code that is very complex. For example, in line 3 we have used a single-line comment starting with `//` to tell the user that it is the header file. Similarly, in line 7, we are showing the end of the main function.

Line 3:

- We know C programs are divided into modules and functions. The user writes some functions, and many others are stored in the C library.

- Library functions are grouped together (category-wise) and stored in a different file known as a header file.
- If you want to access the header file's functions, it is necessary to tell the compiler about the file to be accessed.

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

```
#include<fileName>
```

The file name is the name of the library file that contains the required function definition. For example, one header file `stdio.h` is used in this statement which lets us use certain commands. `stdio` is short for standard input/output, which means it has commands for input, such as reading from the keyboard, and output, such as printing things on the screen.

Lines 4, 5, and 7:

- `main()` is a special function that tells the computer where the program starts. Every program must have exactly one `main()` function.
- The empty parentheses `()` immediately following the `main` indicates that the function `main` has no arguments.
- The opening brace `{` (line 5) indicates the beginning of the function `main`, and the closing brace `}` (line 7) in the last line indicates the end of the function.
- All statements between these two braces `{ }` form the *function body*.
- The word `"void"` before `main` indicates the function `main` does not return anything.
- C permits several forms of the `main` function:
 - o `main()`
 - o `int main()`
 - o `void main()`
 - o `main(void)`
 - o `void main(void)`
 - o `int main(void)`
- We may specify `int` before `main` to indicate the `main` function will return an integer value. When `int` is specified, the last statement in the program must be `"return 0"`.
- The word `"void"` in between the parentheses will indicate the function `main` does not take any argument.

Line 6:

- This is the `printf` command, and it prints text on the screen.
- The data that is to be printed is put inside brackets.

- Also, notice that the words are inside inverted commas because they are what is called a string.
- Each letter is called a character, and a series of characters that are grouped together is called a string.
- Strings must always be put between inverted commas.
- You have to put a semicolon after every command to show that it is the end of the command.

4.5 Compilers and Editors for Executing C Programs

Now that we understand how to write a simple program using C, let us see what is required to execute our program. We need two kinds of software to write and run a program on a computer. The first one is an *editor* by which we type our program in and the second one is a *compiler* that checks our program for errors.

4.5.1 Editors

An editor provides an Integrated Development Environment (IDE) containing several features that help a programmer be more productive. If you don't have an editor installed on your computer, you can use default editors like Notepad (Windows users) or the vi/vim editor (Linux users) to type in your code. You can install any full-featured editors like MS Visual Studio Code, CodeBlocks, SublimeText, or DevC++ on your computer for convenience. These editors provide you with specialized buttons for your program's compilation and execution so that you don't have to type the command on a prompt every time you execute your code.

Figure 4.4 shows the MS Visual Studio Code editor and its components that help a programmer write effective code. It is a source code editor available for Windows, macOS, and Linux. You can find other features of this editor in [7].

4.5.2 Compilers

Without a compiler, we cannot execute a program by an editor only. A compiler is software that translates the source code into machine code. Several compilers have been developed over the years by companies to support the different C standards (C89, C99, C11, C18) proposed so far. A list of compilers and their supported standards are shown in Table 4.2.

If you are a beginner and reading the C programming language for the first time, your first duty is to run Program 4.1 on your machine. You might be thinking about which compiler and editor you need to install on your computer. That depends on what machine you have; is it a Windows, Linux, or Mac system? Depending on your machine, you need to choose a compiler carefully because they are built differently for different machines, and after compilation, they generate environment-specific machine code. Figure 4.5 shows this concept.

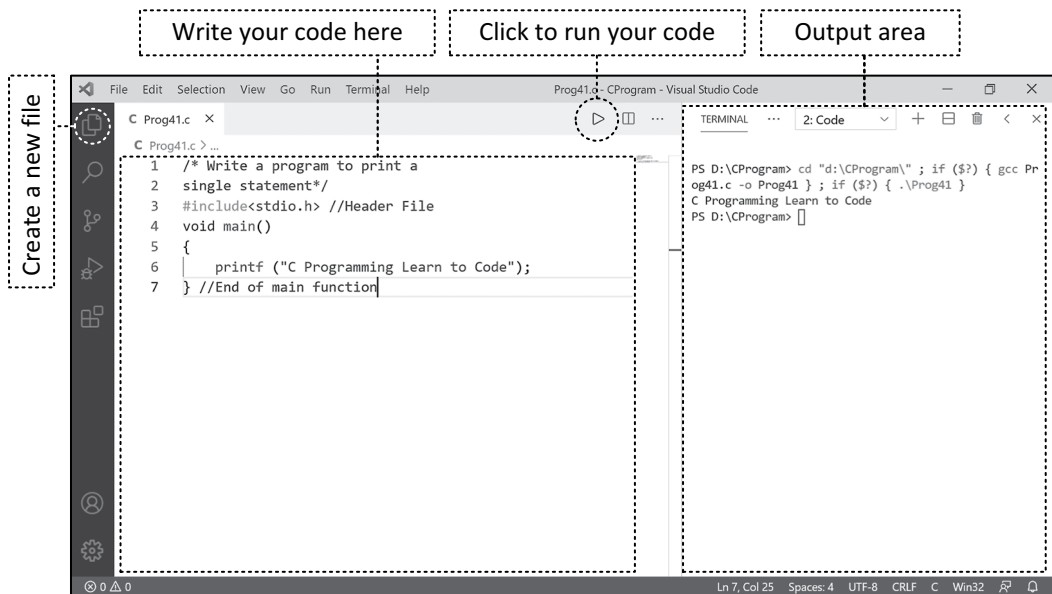


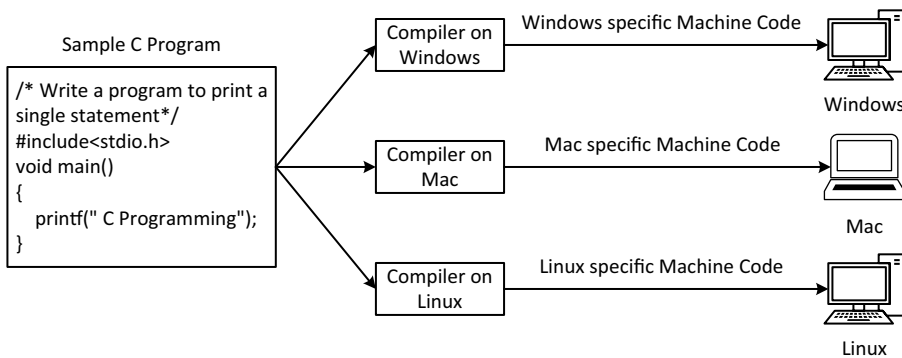
FIGURE 4.4
Microsoft visual studio code editor.

TABLE 4.2

Compilers and Their Supported Standards

Compiler	Supported Standards
Amsterdam Compiler Kit	C K&R and C89/90
Clang, using LLVM backend	C89, C90, C99, and C11
GCC	C89/90, C99, and C11
HP C/ANSI C compiler	C89 and C99
Microsoft Visual C++	C89/90 and C99
Pelles C	C99 and C11 (Windows only)
Vbcc	C89/90 and C99
Tiny C Compiler	C89/90 and some C99

If you are a Windows user, we recommend installing a MinGW compiler on your machine and a Microsoft Visual Studio Code (VS Code) editor to write your program. The installation procedure of the MinGW compiler on the Windows machine is described in Chapter 16. A Linux user does not require any compiler because the GCC compiler is pre-built on these machines. You may need to install VS Code as an editor if you don't want to use the vi editor (preinstalled on a Linux machine). If you are a Mac user, then you need to install the command-line tools. To install it, you need to open a terminal window and type the command "gcc". If your computer has the command-line tool installed already, you will see some text output. If not, you will get a dialog box to install it. Click on the install button and it will take some time to complete the installation process. After the process

**FIGURE 4.5**

Environment-specific machine code generation by different compilers.

completes it will have installed the compiler known as gcc (GNU Compiler Collection). To check for successful installation, you need to type the command “gcc --version” in the terminal window. If you have the version information, then the installation has been successful. You can use any text editor to write your code or install VS Code to edit and execute your program.

4.5.3 Executing Your First C Program

In this section, we will explain how to execute your first C program on your machine. The execution procedure is slightly different from computer to computer. We will consider every machine starting with a Mac.

4.5.3.1 Mac

The precondition is you have already installed a gcc compiler on your machine, as explained in the previous section. You have created a program by typing it in using any default editor and have saved it as prog41.c.

Step 1: Open the terminal window and go to the current directory where your prog41.c file is present.

Step 2: Type the following command to compile your program:

```
gcc -o exec prog41.c
```

Here, “exec” specifies the executable file’s name generated after successfully compiling the program prog41.c. Instead of using exec, you can also use any name of your choice.

Step 3: Type the following command to execute your code:

```
./exec
```

Now you can see the output on your screen.

4.5.3.2 Windows

The precondition is you have already installed the MinGW compiler on your machine, as explained in Chapter 16. You have created a program by typing it in using any default text editor (Notepad) and saved it as prog41.c.

Step 1: Open the command prompt and go to the current directory where your prog41.c file is present.

Step 2: Type the following command to compile your program:

```
gcc -o runfile prog41.c
```

Here, “runfile” specifies the executable file’s name generated after successfully compiling the program prog41.c. Instead of using runfile, you can also use any name of your choice.

Step 3: Type the following command to execute your code:

```
runfile
```

Now you can see the output on your screen.

4.5.3.3 Linux

The preconditions are the same; the gcc compiler is a default compiler in a Linux machine and does not require installation. Create a C program file, type your code, and save it as prog41.c.

Step 1: Open the terminal window and go to the current directory where your prog41.c file is present.

Step 2: Type the following command to compile your program:

```
gcc -o exec prog41.c
```

Here, “exec” specifies the executable file’s name generated after successfully compiling the program prog41.c. Instead of using exec, you can also use any name of your choice.

Step 3: Type the following command to execute your code:

```
./exec
```

Now you can see the output on your screen.

I recommend the reader to install MS Visual Studio Code as an editor so that you don’t have to type the command every time you run your program. VS Code is freely available for all environments, whether Windows, Linux, or Mac.

4.6 Review Questions

4.6.1 Objective Questions

1. The C programming language was developed in the year _____ and by _____.
2. C is basically a combination of two languages that are _____ and _____.
3. The documentation section contains a set of _____ lines.
4. C is a _____ level programming language.
5. Every C program must have one `main()` function. True/false?
6. A program can have how many main functions?
7. The `main()` function doesn't return any value. True/false?
8. In which year was C standardized by ANSI?
9. Name at least two editors for running a C program.
10. Name at least two header files used in a C program.

4.6.2 Short Answer Questions

1. What is meant by object code?
2. Why is linking needed?
3. What is the preprocessor directive?
4. What is the need for a header file in C?
5. What is the use of a `stdio.h` header file?
6. Is comment nesting possible in C? If no, why? If yes, how?
7. What does the `#` symbol specify in the declaration of preprocessor statements?
8. What is a global variable?
9. How can you specify single line and multi-line comments in C?
10. What is the difference between a compiler and an editor?
11. Is it compulsory to include a header file while writing a C program?

4.6.3 Programming Questions

1. Write a C program to display your college name and address.
2. Analyze the following programs and find the errors present in these programs.

```
(a)      /* Write a program to print a single statement
          #include<stdio.h> //Header File
          void main()
          {
              printf ("C Programming Learn to Code");
          } //End of main function
```

- ```
(b) /* Write a program to print a single statement */
 #include<stdio.h> //Header File
 void main()
 {
 printf ("C Programming Learn to Code")
 }

(c) /* Write a program to print a single statement*/
 #include<stdio.h> //Header File
 //void main()
 {
 printf ("C Programming Learn to Code");
 } //End of main function

(d) /* Write a program to print a single statement*/
 #include<stdio.h> //Header File
 void main()
 {
 //printf ("C Programming Learn to Code");
 } //End of main function

(e) /* Write a program to print a single statement*/
 #include<stdio.h> //Header File
 void main(void)
 {
 printf ("C Programming Learn to Code");
 } //End of main function

(f) /* Write a program to print a single statement*/
 #include<stdio.h> //Header File
 int main()
 {
 printf ("C Programming Learn to Code");
 return 0;
 } //End of main function
```

### 3. What output is produced when the following program executes?

```
/* Write a program to print a
single statement*/
#include<stdio.h> //Header File
void main()
{
 printf ("Incredible Country");
 printf(" Great Country");
 printf(" I love my Country");
} //End of main function
```

### 4. Execute the following two programs on your computer and analyze the output difference.

```
/* Write a program to print a single statement*/
#include<stdio.h> //Header File
void main()
{
 printf ("Incredible Country");
 printf(" Great Country");
 printf(" I love my Country");
} //End of main function
```

```
/* Write a program to print a single statement*/
#include<stdio.h> //Header File
void main()
{
 printf ("Incredible Country Great Country I love my Country");
} //End of main function
```

#### 4.6.4 Long Questions

1. Describe the structure of a C Program.
2. What are the advantages of using a C program?
3. Write short notes on the history of the C programming language.
4. How can you execute a C program? Explain each step with an appropriate diagram.
5. What is the difference between a compiler and an editor? List out all the compilers developed so far with their supported standards.
6. Prepare documentation about the installation procedure of the compiler and an editor on your machine. Explain each step with a proper figure starting from the installation to executing the simple program shown in Program 4.1.

---

#### References

1. Ritchie, Dennis M., "The development of the C language," *ACM Sigplan Notices*, vol. 28, no. 3, pp. 201–208, 1993.
2. Ritchie, Dennis M., Brian W. Kernighan, and Michael E. Lesk, *The C programming language*. Prentice Hall, 1988.
3. Stroustrup, Bjarne, "Sibling Rivalry: C and C++," (2002).
4. ISO, LA0, "ISO/IEC 9899: 2018-information technology-programming languages–C," (2018).
5. ISO, ISO, "ISO/IEC 9899: 2011 Information technology—Programming languages—C," (2011).
6. Organisation, I. S., "ISO/IEC 9899: 1999 Programming Languages–C," (1999).
7. <https://code.visualstudio.com/learn>





**Taylor & Francis**

Taylor & Francis Group

<http://taylorandfrancis.com>

# 5

## Constants, Variables, and Data Types

### 5.1 Introduction

Before you learn any language, not explicitly a programming language, you need to learn its character sets. You form words using those character sets, write sentences, and finally write paragraphs, essays, and so on. Besides that, you need to learn about the grammar of that language to form a sentence correctly. The grammar specifies the syntax, semantics, and morphology of a language. Figure 5.1 shows the usual flow of learning any language.

In this chapter, we will learn the fundamental things required to write an error-free program. The C programming language has its tokens for coding and developing efficient programs. The smallest individual units of a C program are known as tokens. A C program can be constructed by using all these tokens and their syntax rules. The whole chapter is dedicated to explaining these C tokens.

After completing this chapter, the student will learn the necessary tokens needed to write a C program, including character sets, keywords, variables, constants, and finally, how to write simple programs using these tokens.

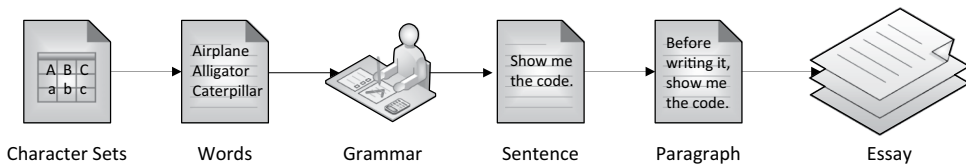
### 5.2 C Character Sets

A character denotes any alphabet, digit, or special character used to form words, numbers, and expressions:

*Alphabets:* A, B, C ....., Z and a, b, c, d, ..., z

*Digits:* 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

*Special characters:* # % ^ & \* ( ) - + = | \ ? / . < > , " ; ' @ \_ [ ] { } etc.



**FIGURE 5.1**  
Learning a language.

### 5.3 Keywords

Keywords are words whose meaning has already been explained in relation to C compilers. These keywords cannot be used as variable names because, if we do so, we will be trying to assign a new meaning to the keyword, which is not allowed.

Keywords are words whose meaning is already known to the compiler.

According to the C89 standard (ANSI C) there are 32 keywords available in C. While we are using keywords in our program writing we must write them in lowercase letters. Table 5.1 lists all the keywords as per the C89 standard:

Besides these keywords, there are several introduced in other C standards. C99 introduces five new keywords, and C11 introduces seven. Table 5.2 shows the list of keywords introduced in C99 and C11.

**TABLE 5.1**

List of Keywords Supported by the C89 Standard

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

**TABLE 5.2**

List of Keywords Introduced in C99 and C11

| C99        | C11            |
|------------|----------------|
| _Bool      | _Alignas       |
| _Complex   | _Alignof       |
| _Imaginary | _Atomic        |
| inline     | _Generic       |
| restrict   | _Noreturn      |
|            | _Static_assert |
|            | _Thread_local  |

### 5.4 Variables and Identifiers

Variables in C are the named memory locations where we can store values, and these values vary or change during program execution. These memory locations (variables) can store integers, characters, and real values. When we store some value in the variable, the previous value (if present) will be deleted.

Variables are named memory locations where we store a value.

Rules for constructing a variable name:

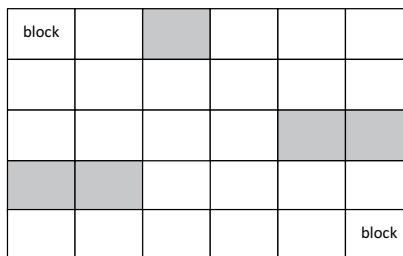
1. A variable name can be any combination of digits, characters, or underscores. The first character in the variable name must be alphabetic.
2. No commas or blank spaces are allowed within a variable name.
3. A variable name should not be a keyword.
4. We cannot use special symbols other than underscores (`_`) in a variable name.
5. Upper case and lower case are significant, e.g., the variable `Var` is not the same as `var` or `VAR`.

Table 5.3 shows some valid and invalid variable names.

What is a named memory location? As we know, every computer has a primary memory, and before executing a program, we must allocate space for every variable present in our program. From the programmer’s view, a primary memory looks as shown in Figure 5.2.

**TABLE 5.3**  
Valid and Invalid Variable Names

| Valid        |              | Invalid                                              |
|--------------|--------------|------------------------------------------------------|
| a            | Basic Salary | Space is not allowed between variable names          |
| Total        | 2age         | A variable name should not start with a number.      |
| Basic_salary | void         | This is a keyword.                                   |
| age1         | sal#         | No special characters allowed other than underscore. |
| sal          | default      | This is a keyword.                                   |



**FIGURE 5.2**  
A memory segment (Programmer’s View).

- A programmer assumes that a primary memory is a collection of blocks, and each block is one byte (eight bits) long.
- Every block has an address.
- To store anything, we might need one or more blocks in a contiguous fashion.
- The shaded region in Figure 5.2 shows the allocated blocks.
- Before we store any value, we need to name the block with some identifier so that we can access that value using its name in the future. Hence, the variables are known by their named memory locations.

*Identifiers* are names given to various program elements, such as variables, functions, arrays, structures, and other user-defined objects. The rules applied for variable declaration are also applicable for identifiers. All variables can be an identifier, but all identifiers may not be a variable.

Identifiers are names given to various program elements, such as variables, functions, arrays, structures, and other user-defined objects.

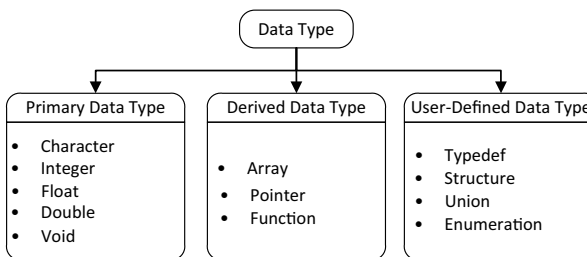
We will see more about identifiers in subsequent chapters.

### 5.5 Data Types

In the real world, we use different types of data, such as integers, reals, and characters. To store data or values, we need variables. In the previous section, we saw that a variable is a named memory location. We have also seen that before assigning value to a variable, we must allocate some memory blocks for it. Now, the question is, how many blocks do we need to store a value? That depends on the type of data you are going to store in that variable. Hence, we must declare a variable with a data type that instructs the compiler to allocate the required number of blocks and give it a name.

C language data type is broadly classified into three groups, as shown in Figure 5.3.

In this chapter, we will discuss primary data type. Derived data type and user-defined data type are discussed as and when needed.



**FIGURE 5.3**  
Data types in C.

### 5.5.1 Primary Data Types

These are also known as basic data types, predefined, and previously known to the compiler. We use some keywords to represent these data types. Table 5.4 shows these data types, and their corresponding keywords.

We use these keywords to declare variables when we want to store values. Each primary data type conveys two crucial pieces of information:

1. It defines the type of data the user wants to store in these variables;
2. It also instructs the compiler to allocate the required amount of space needed to store these values.

The amount of space allocated for each data type is different from others and depends on the platform you are using. For instance, if you are using the keyword “char” to declare a variable, then that variable consumes one byte (eight bits) of storage space in memory, and we can store a single character in it. The C programming language has an operator known as `sizeof`, and we can use it to know the exact amount of memory allocation in bytes. In the following section, I will explain each primary data type in detail and use them to allocate and assign values.

### 5.5.2 Integer Data Types

Integers are whole numbers with a machine-dependent range of values. We use the integer data type to declare variables that can store integers, either positive or negative, with no fractional component. Table 5.5 shows some examples of integer numbers and numbers which are not integers.

**TABLE 5.4**  
Data Types and Keywords Representing Them

| Serial No. | Data Types | Keywords |
|------------|------------|----------|
| 1          | Integer    | int      |
| 2          | Character  | char     |
| 3          | Float      | float    |
| 4          | Double     | double   |
| 5          | Void       | void     |

**TABLE 5.5**  
Examples of Integer Numbers and Non-integer Numbers

| Integers | Non-integers    |
|----------|-----------------|
| +63      | 8.75            |
| -95      | $5\frac{23}{4}$ |
| 9        | $\frac{4}{4}$   |
| 89       | $\sqrt{34}$     |

TABLE 5.6

## Type Modifiers

| Type Modifier | Keyword  | Description                                                                                           |
|---------------|----------|-------------------------------------------------------------------------------------------------------|
| Signed        | signed   | This allows the user to store either positive or negative integers in the allocated memory locations. |
| Unsigned      | unsigned | Only positive integers are allowed.                                                                   |
| Long          | long     | Reduces the amount of memory location to half (compiler dependent).                                   |
| Short         | short    | Space allocation will be doubled (compiler dependent).                                                |

TABLE 5.7

## Data Types, Memory Allocation Size, and the Range

| Type                          | Storage Size        | Min. Value                     | Max. Value                      |
|-------------------------------|---------------------|--------------------------------|---------------------------------|
| int OR signed int             | 2 bytes (Turbo C++) | -32768                         | +32767                          |
|                               | 4-bytes (others)    | -2,147,483,648                 | +2,147,483,647                  |
| unsigned int                  | 2-bytes (Turbo C++) | 0                              | +65,535                         |
|                               | 4-bytes (others)    | 0                              | +4,294,967,295                  |
| short int OR signed short int | 2-bytes             | -32768                         | +32767                          |
| unsigned short int            | 2-bytes             | 0                              | +65,535                         |
| long int OR signed long int   | 4-bytes             | -2,147,483,648                 | +2,147,483,647                  |
| unsigned long int             | 4-bytes             | 0                              | +4,294,967,295                  |
| long long OR signed long long | 8-bytes             | -9,223,372,036,<br>854,775,808 | +9,223,372,036,<br>854,775,807  |
| unsigned long long            | 8-bytes             | 0                              | +18,446,744,073,<br>709,551,615 |

If you are using a Turbo C++ compiler, then an integer takes 2 bytes (16 bits) of memory, and its range is from  $-32768$  to  $+32767$ . Other compilers like gcc or minGW take 4 bytes (32 bits), and the range is from  $-2147483648$  to  $+2147483647$ . Some type modifiers alter the basic characteristics of primary data types and give some flexibility to memory allocation. Those type modifiers are listed in Table 5.6, along with their keywords and their description.

We can associate these type modifiers with the basic data types to modify their size and range. For instance, the basic data type *int* allocates four bytes of memory, but when we associate the *short* keyword with *int* (*short int*), it allocates two bytes of memory. Table 5.7 shows the detail of these data types, memory allocation in bytes, and the range of numbers represented in them.

### 5.5.3 Floating Point Types

The floating point number represents a real number with six-digit precision. To represent a floating point number, we require the keyword *float* to declare a float variable. When the accuracy of the floating point number is insufficient, we can use the keyword *double* to

**TABLE 5.8**

Floating Point Types with Defined Size and Range

| Serial No. | Types       | Size (in Bits) | Range                          |
|------------|-------------|----------------|--------------------------------|
| 1          | float       | 32 (4 byte)    | $3.4e - 38$ to $3.4e + 38$     |
| 2          | double      | 64 (8 byte)    | $1.7e - 308$ to $1.7e + 308$   |
| 3          | long double | 80 (10 byte)   | $3.4e - 4932$ to $1.1e + 4932$ |

**TABLE 5.9**

Character Data Types with Defined Size and Range

| Serial No. | Types               | Size (in Bits) | Range        |
|------------|---------------------|----------------|--------------|
| 1          | char or signed char | 8 (1 byte)     | -128 to +127 |
| 2          | unsigned char       | 8 (1 byte)     | 0 to 255     |

declare our variable. The *double* is the same as *float* but with longer precision. To extend the precision further, we can use *long double*, which consumes 80 bits of memory space. Table 5.8 shows the detail of these data types, space allocation, and their ranges.

#### 5.5.4 Character Data Types

A character in C refers to an alphabet, number, or symbol enclosed within a single quote. For example, 'a', '5', '6', and '&' represents a character constant. To store these characters, we require a variable declared with the `char` keyword. Characters are usually stored in eight bits of internal storage. The type modifier signed or unsigned can be explicitly applied to `char`. While unsigned characters have values between 0 and 255, signed characters have values from -128 to 127. Table 5.9 shows the detail of these data types, size, and their ranges.

A character is stored in memory as an ASCII value. ASCII stands for American Standard Code for Information Interchange. ASCII is a character encoding standard that assigns a numeric value to each character used in a computer. For example, the ASCII value of capital 'A' is 65, and the small 'a' is 97. The entire ASCII table is presented in Appendix A for easy reference.

#### 5.5.5 Void Types

We use the data type `void` to specify an empty set of values. In general, data type `void` is used to specify the return type of a function or to define a pointer that can hold the address of any variable. This is not the right place to discuss further void data types.

Now that we understand the basic concept of variable and data types, it's time to declare a variable using C code syntax. In the next section, we will discuss how to declare a variable, and we will see how memory allocation takes place for our variable, and how we can assign some value to this variable.



### 5.6 Declaration of Variables

In C programming, we need to declare every variable before we use it inside our program code. As we know, a variable denotes a memory location, and naming the variable is purely user-specific. As we have discussed in the previous section, to declare a variable, we need the help of a data type. The syntax is shown in Figure 5.4.

We need to declare every variable before we use it inside our program code.

We should name the variable in such a manner that it conveys the sense of what it contains. For example, suppose we want to store the average of five numbers in a variable, then we may choose a name like `avg`, `avg5`, `avg_five`, or `average`. But remember that it is not a mandatory condition to choose a meaningful name; instead, it is a suggestion. Let us take some examples of declaring a variable (see Figure 5.5).

Multiple variable declaration specifies the variables with a comma-separated list (Figure 5.5b). A variable declaration conveys the following things:

1. Name of the variable(s);
2. Type of data the variable(s) holds;
3. The size and the range of the variable(s).

Now let us discuss how the compiler deals with this declaration and what changes are made by the compiler in memory allocation. The discussion is based on the single variable declaration statement shown in Figure 5.5a.

- When the declaration line is executed, the compiler allocates a memory block and names it `avg`, as shown in Figure 5.6a.
- The blocks with patterns specify some allocated blocks of memory.

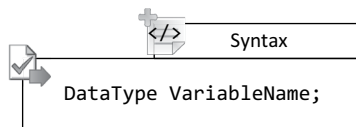


FIGURE 5.4 Declaration of a variable.

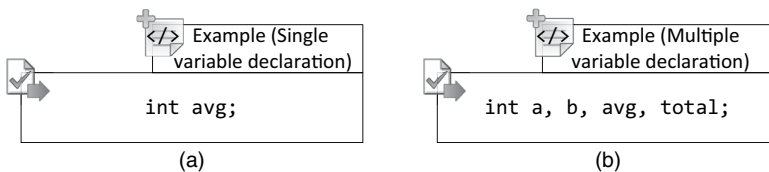
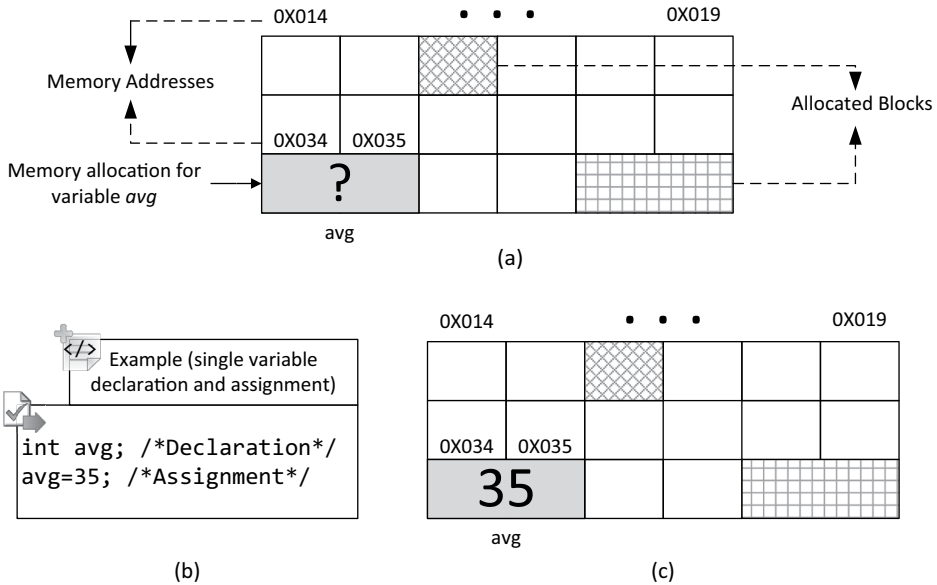


FIGURE 5.5 Examples of single and multiple variable declarations.

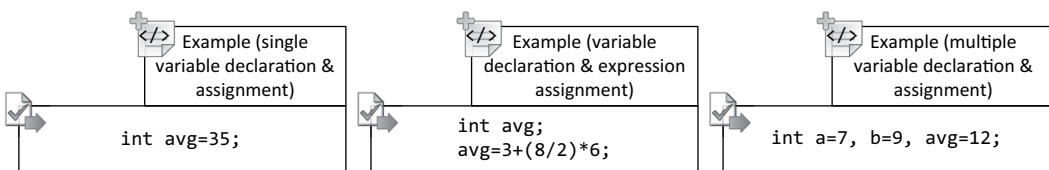


**FIGURE 5.6** Variable declaration, assignment, and memory allocation.

- Every block has an address starting with 0X. The addresses shown in the figure are imaginary and different from the actual address.
- As an integer takes two bytes (we are assuming each block stores one byte of information), so two blocks get allocated for avg, and the addresses are 0X034, 0X035.
- The question mark inside the allocated block indicates no value yet assigned to avg. As no value is assigned, it is called an *uninitialized variable*.

Figure 5.6c shows what happens when we assign a value to avg. To assign a value, we need an assignment operator (=), and the code is shown in Figure 5.6b. The line enclosed with /\* and \*/ is known as the *comment line*, and the compiler ignores these lines during execution. We use these lines for documentation purposes.

The variable declaration can take various forms. We can combine the declaration and the assignment line to form a single line. We can also assign a whole expression to a variable. Figure 5.7 shows all the different kinds of representation possible with variable declaration and assignment.



**FIGURE 5.7** Different types of variable declarations and assignments.

## 5.7 Constants

A constant represents a value that does not change. We can use a variable to store a constant, or we can use the constant directly in our expression. Note that a variable is a name given to the memory location.

For example, observe the following expression:

$$3x + 2y = 38$$

Here:

- The values 3, 2, and 38 are constants, and x and y are variables;
- If you consider the example shown in Figure 5.6, avg is a variable and the value 35 is a constant assigned to avg.

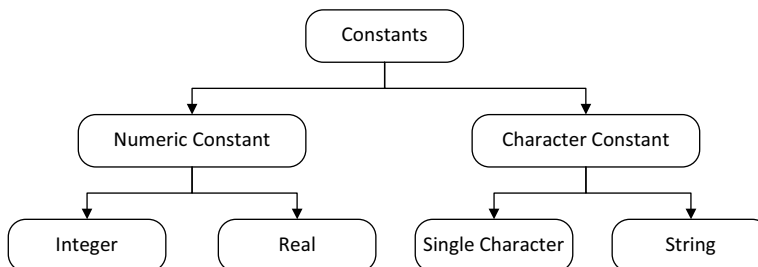
There are several types of constants we can use, and in C programming, the constants are classified as shown in Figure 5.8.

### 5.7.1 Integer Constants

These represent integer values that contain a collection of digits. The following points explain the more detailed representation of an integer constant:

- An integer constant must have at least one digit;
- It must not have a decimal point;
- It could be either positive or negative;
- If no sign precedes an integer constant, it is assumed to be positive;
- No blank spaces are allowed within an integer constant.

Examples: 435, 76, +98, -43, 789.



**FIGURE 5.8**  
Constants in C.

### 5.7.2 Real Constants

Real constants are also called floating-point constants. They represent values that contain a decimal point. There are two ways to represent a real constant:

1. Fractional form;
2. Exponential form.

### 5.7.3 Fractional Form

- A real constant must have at least one digit.
- It must have a decimal point.
- It could be either positive or negative. The default sign is positive.
- No commas or blank spaces are allowed within a real constant.

Examples: +324.56, -568.9, 32.4, 748.00.

### 5.7.4 Exponential Form

In this case, the real constants are represented in two parts:

- The part appearing before 'e' is called the mantissa. The part following 'e' is called the exponent.
- The mantissa part and the exponent part must be separated by the letter 'e' or 'E'.
- The mantissa part may have a positive or negative sign.
- The default sign of the mantissa part is positive.
- The exponent must have at least one digit, which must be a positive or a negative integer.
- The default sign is positive.

Examples: +3.2e-5, 4.1e5, -0.5e45.

### 5.7.5 Character Constants

A character constant is either a single alphabet/digit/special symbol enclosed within a pair of single quotes:

- The maximum length of a character constant is 1;
- Character constants have an integer value known as an ASCII value.

Examples: 'a', '5', '&'.

### 5.7.6 String Constants

A string constant is a sequence of characters enclosed in a double quote. The characters may be letters, numbers, or special symbols.

Examples: "program", "235", "5+3", "x".

### 5.8 Learn to Code Examples

In this section, we will introduce some programming examples that will help us to understand the writing style of C code.

#### EXAMPLE 1

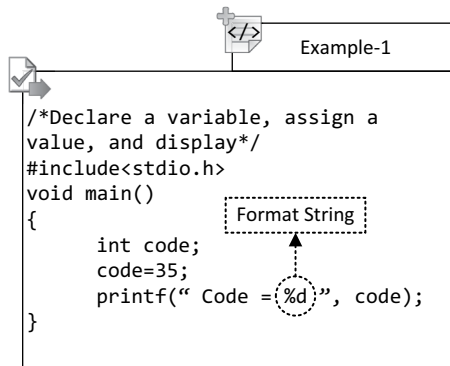
In the first example, let us take a variable, assign a value, and display the value on the screen. Figure 5.9 shows the code segment.

The explanation of the above program is as follows:

- The line `int code;` declares the variable.
- The line `code=35;` assigns 35 to the variable `code`.
- The last line that displays the results needs a detailed explanation. We will discuss the `printf()` statement in the following section.
- Every statement must end with a semicolon.

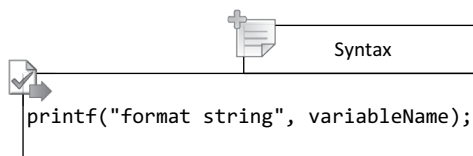
The `printf()` is a function used to print the value contained in a variable. The general form is as shown in Figure 5.10.

The format string or the format specifier starts with a `%` symbol followed by a character such as `d`, `c`, or `f`. The purpose of writing a format string is to tell the compiler about the



```
Example-1
/*Declare a variable, assign a
value, and display*/
#include<stdio.h>
void main()
{
 int code;
 code=35;
 printf(" Code =%d", code);
}
```

**FIGURE 5.9**  
A simple C program to display the value of a variable.



```
Syntax
printf("format string", variableName);
```

**FIGURE 5.10**  
Syntax of `printf()` function.

**TABLE 5.10**  
List of Format Strings for Different Data Types

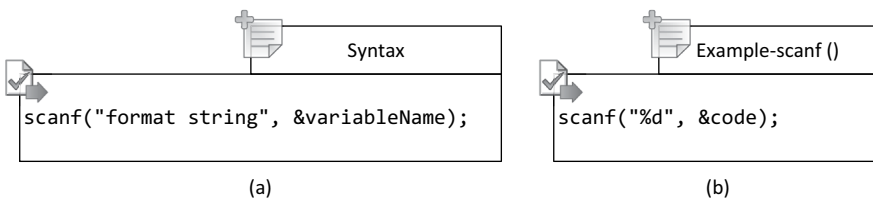
| Serial No. | Data Type          | Format String |
|------------|--------------------|---------------|
| 1          | int                | %d or %i      |
| 2          | char               | %c            |
| 3          | float              | %f            |
| 4          | double             | %lf           |
| 5          | short int          | %hd           |
| 6          | long int           | %ld or %li    |
| 7          | signed int         | %hi or %hd    |
| 8          | unsigned int       | %u            |
| 9          | signed short int   | %hi           |
| 10         | unsigned short int | %hu           |
| 11         | signed long int    | %ld or %li    |
| 12         | unsigned long int  | %lu           |
| 13         | long double        | %Lf           |
| 14         | long               | %lld or %lli  |
| 15         | unsigned long long | %llu          |
| 16         | Octal              | %o            |
| 17         | Hexadecimal        | %x or %X      |
| 18         | String             | %s            |

type of data we are using. For each data type, there exists a separate format string. The format string for all data types is shown in Table 5.10.

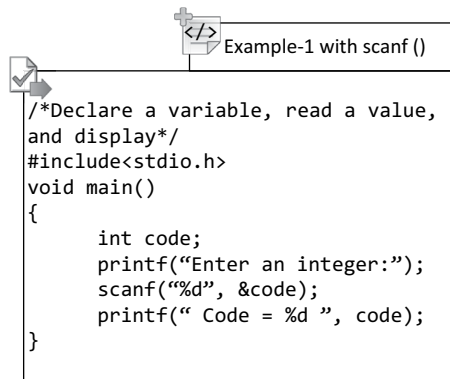
In the above program, we have assigned a number to the variable code. We can also ask the user to input any number. We can do this by using the `scanf()` function (see Figure 5.11a).

- The format string contains the format of the data being received;
- The ampersand (&) symbol before the variable name is an operator that specifies the address of the variable.

Figure 5.11b shows an example. When this statement is encountered by the compiler, the execution stops, and the compiler waits for an integer value to be typed in. The %d



**FIGURE 5.11**  
Syntax of the `scanf()` function and an example.



```

Example-1 with scanf ()
/*Declare a variable, read a value,
and display*/
#include<stdio.h>
void main()
{
 int code;
 printf("Enter an integer:");
 scanf("%d", &code);
 printf(" Code = %d ", code);
}

```

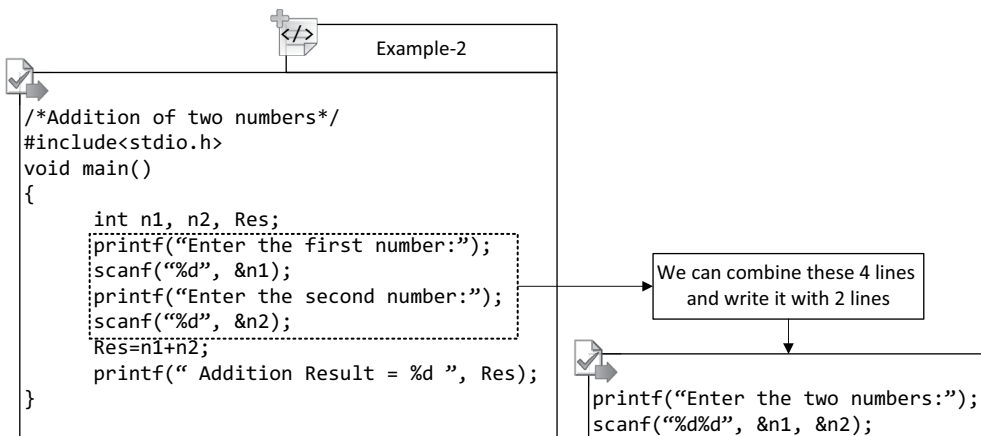
**FIGURE 5.12**  
Rewriting Example 1 to take input from the user and display it.

indicates that the value typed must be an integer. When we type a value, it will automatically be assigned to the variable *code*.

Now let us rewrite the above code (Example 1) where the programmer has to declare a variable of integer type, use the `scanf ()` function to read the data from the user, and display it on the screen. Figure 5.12 shows the code.

## EXAMPLE 2

Let us take another example that adds two numbers. The programmer has to declare three variables: two variables to store two numbers and the third variable to store the addition result. Our program must provide the facility to enter any number the user wants and finally show the result on the screen. Figure 5.13 shows the C code.



```

Example-2
/*Addition of two numbers*/
#include<stdio.h>
void main()
{
 int n1, n2, Res;
 printf("Enter the first number:");
 scanf("%d", &n1);
 printf("Enter the second number:");
 scanf("%d", &n2);
 Res=n1+n2;
 printf(" Addition Result = %d ", Res);
}

```

We can combine these 4 lines and write it with 2 lines

```

printf("Enter the two numbers:");
scanf("%d%d", &n1, &n2);

```

**FIGURE 5.13**  
Example 2: Addition of two numbers.

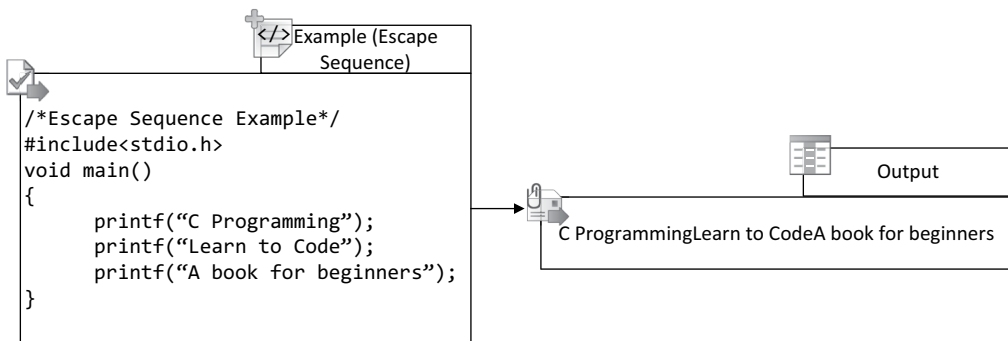
### 5.9 Escape Sequences

Escape sequences are special characters used for formatting. They can be recognized in the code by their special backslash followed by a character that does a specific task. These specialized printing characters are used to make the output readable when printing characters to the screen, file, or other devices.

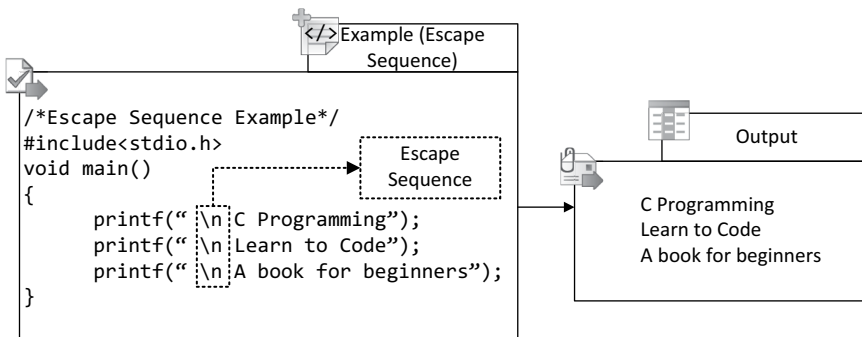
Let us write one example to show you the importance of an escape sequence (see the following code and its corresponding output in Figure 5.14). When we execute this code, the output will appear like this. If you observe the output, everything is displayed in a single line, with no space between the sentences.

If we want the information to be displayed in multiple lines, then we have an escape sequence `\n`. By using this, we can display the contents over multiple lines. We will rewrite the above code (Figure 5.14) as shown in Figure 5.15.

There are several escape sequences supported by the C compiler. A list of all these escape sequences is shown in Table 5.11 along with their meanings. The reader is encouraged to write at least one programming example to understand the working of these escape sequences.



**FIGURE 5.14**  
Understanding the importance of escape sequences.



**FIGURE 5.15**  
Using an escape sequence to format output.



TABLE 5.11

Escape Sequences and Their Meanings

| Serial No. | Escape Sequence | Name            | Meaning                                                    |
|------------|-----------------|-----------------|------------------------------------------------------------|
| 1          | \a              | Alert           | Produce an audible or visible alert                        |
| 2          | \b              | Backspace       | Moves the cursor back one position (non-destructive)       |
| 3          | \f              | Form Feed       | Moves the cursor to the first position of the next page    |
| 4          | \n              | New Line        | Moves the cursor to the first position of the next line    |
| 5          | \r              | Carriage Return | Moves the cursor to the first position of the current line |
| 6          | \t              | Horizontal Tab  | Moves the cursor to the next horizontal tabular position   |
| 7          | \v              | Vertical Tab    | Moves the cursor to the next vertical tabular position     |
| 8          | \'              |                 | Produces a single quote                                    |
| 9          | \"              |                 | Produces a double quote                                    |
| 10         | \?              |                 | Produces a question mark                                   |
| 11         | \\              |                 | Produces a single backslash                                |
| 12         | \0              |                 | Produces a null character                                  |

## 5.10 Review Questions

### 5.10.1 Objective Questions

1. There are \_\_\_\_\_ keywords available in C as per the C89 standard.
2. \_\_\_\_\_ in C are the named memory locations where we can store values, and these values vary or change during program execution.
3. The range of an integer variable is \_\_\_\_\_, and the range of a character variable is \_\_\_\_\_. (Assume that integers take 2 bytes and characters take 1 byte in memory.)
4. A \_\_\_\_\_ does not change its value during the entire execution of the program.
5. \_\_\_\_\_ is used as a format string for *double* variables.
6. All variables are identifiers, but all identifiers are not variables. True/false?

### 5.10.2 Programming Questions

1. Write a program to find out the average of three numbers.
2. Write a program to find the area of a rectangle.
3. Write a program to calculate simple interest.
4. Write a program to find out the area of a right-angle triangle.
5. Write a program to swap two numbers.
6. Write a program to swap two numbers without using a third variable.
7. Write a program to find the ASCII value of the character 'g'.

8. Write a program to multiply two floating-point numbers by using float and double variables.
9. Predict the output or find the error in the following program code:

```
#include<stdio.h>
void main()
{
 int goto=25;
 printf("%d", goto);
}
```

```
#include<stdio.h>
void main()
{
 int Float=25;
 printf("%d", Float);
}
```

```
#include<stdio.h>
void main()
{
 int basic;
 printf("\n%d", sizeof(basic));
 printf("\n%d", sizeof(int));
}
```

```
#include<stdio.h>
void main()
{
 unsigned int basic=-25;
 printf("%u", basic);
}
```

```
#include<stdio.h>
void main()
{
 char symbol='p';
 printf("%d", symbol);
}
```

```
#include<stdio.h>
void main()
{
 char symbol='p';
 printf("%c %c", symbol, symbol+5);
}
```

```
#include<stdio.h>
void main()
{
 int x;
 x=printf("Program");
 printf("%d", x);
}
```

```
#include<stdio.h>
void main()
{
 printf("%d", printf("C
Programming"));
}
```

```
#include<stdio.h>
void main()
{
 int a;
 printf("%d", a);
}
```

```
#include<stdio.h>
void main()
{
 printf("%d", printf("C
Programming")+5);
}
```

```
void main()
{
 int a;
 printf("%d", a+5);
}
```

```
#include<stdio.h>
void main()
{
 printf("Quote me the Risk\rLearn");
}
```

```
#include<stdio.h>
void main()
{
 int a=5;
 a=7;
 a=9;
 printf("%d", a);
}
```

```
#include<stdio.h>
void main()
{
 printf("Correct Me\b\bYourself");
}
```

```
#include<stdio.h>
void main()
{
 int a=5;
 int a=7;
 int a=9;
 printf("%d",a);
}
```

```
#include<stdio.h>
void main()
{
 int a=3277;
 printf("%d",printf("%d",a));
}
```

### 5.10.3 Subjective Questions

1. What are variables, and what are the rules for constructing a variable name?
2. What is the difference between variables and identifiers?
3. What is a keyword?
4. What is a constant? What are the different types of constants in C?
5. C is a strongly typed language. Justify.
6. Write the syntax of the `printf()` and `scanf()` functions.
7. What is an escape sequence? Explain the need for escape sequences with an appropriate programming example.
8. What is the difference between declaration and initialization? Explain with an example.
9. Write short notes on (a) keywords, (b) format strings, (c) identifiers.
10. What is a preprocessor directive? What is the use of the preprocessor directive?

# 6

---

## *Operators and Expressions*

---

### 6.1 Introduction

This chapter will discuss the two most essential parts of any programming language: operators and expressions. An operator is a symbol that tells the computer to perform specific mathematical or logical operations. Operators are used in programs to manipulate data and variables. C is very rich in built-in operators.

An operator is a symbol that tells the computer to perform specific mathematical or logical operations.

Operators can be either binary or unary. *Binary operators* are the operators where one operator will act upon two operands.

For example:

$$12 + 14$$

In this example, the operator + is acted upon two operands 12 and 14. Hence + is a binary operator here. Similarly, a *unary operator* has one operator and one operand.

For example:

$$-25$$

In this example, the operator – is acted upon one operand 25. Hence – is a unary operator here.

The following are the operators used in C. These operators may be used as binary or unary operators.

1. Arithmetic operators;
2. Relational operators;
3. Assignment operators;
4. Logical operators;
5. Increment or decrement operators;
6. Conditional operators;

7. Bitwise operators;
8. Special operators.

In the subsequent section, we will discuss the feature of all these operators, how to form expressions, how they are executed, and in what order. Finally, we will introduce the precedence of operators.

On completion of this chapter, students will have learnt the working of different operators, the precedence of operators, and the way the C compiler executes an expression. Some new operators like increment, decrement, ternary, and other special operators are also a part of this chapter.

## 6.2 Arithmetic Operators

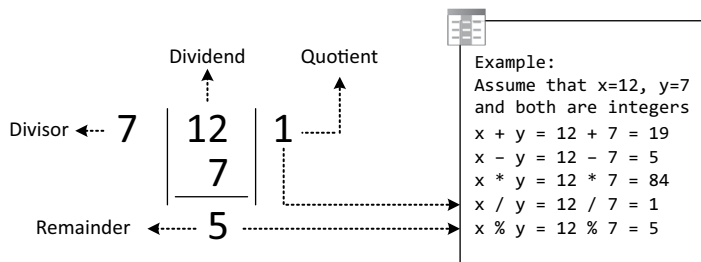
We all know what an arithmetic operator is, and the C language does support all these operators with slight deviations with division and modulo operators. Table 6.1 shows the complete set of arithmetic operators and their symbols.

We all know the result produced by these operators. For the sake of completeness, Figure 6.1 shows the results obtained by these arithmetic operators. You can observe that the division operation returns the quotient, and the modulus operator returns the remainder.

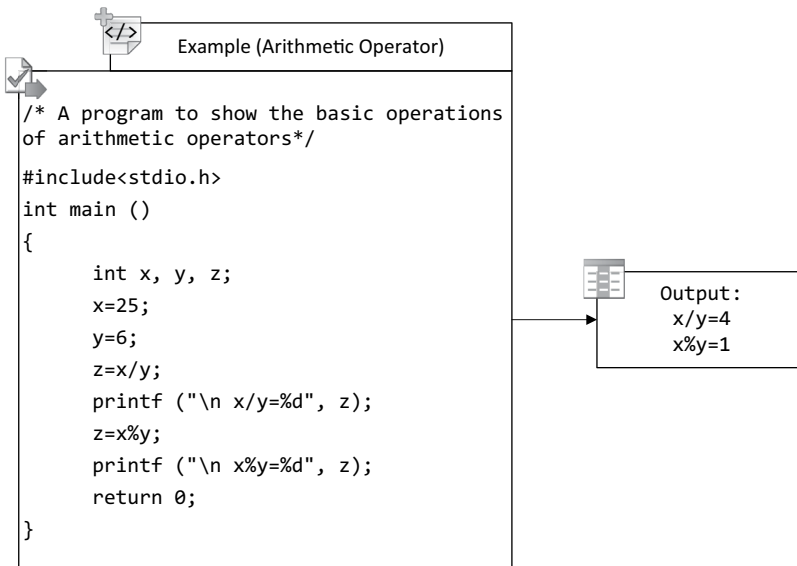
**TABLE 6.1**

Arithmetic Operators

| Serial No. | Operator Name       | Symbol |
|------------|---------------------|--------|
| 1          | Addition            | +      |
| 2          | Subtraction         | -      |
| 3          | Multiplication      | *      |
| 4          | Division            | /      |
| 5          | Modulus (remainder) | %      |



**FIGURE 6.1**  
Arithmetic operations.



**FIGURE 6.2**  
C program showing operations of arithmetic operators.

Let us write an example program that shows the result of the divide and modulo operations. Figure 6.2 shows the C program code. You can see that when 6 divides 25, we obtain 4 (quotient), and when we perform the modulo operation, we get the remainder, which is 1.

### 6.3 Relational Operators

There are situations where we need to compare two values and make decisions. The C language provides several comparison operators for this purpose. The work of these operators is to compare two items for their equality, inequality, or whether one is greater/smaller than the other.

- These are binary operators and work upon two operands;
- The result of these operators is either “true” or “false”;
- If the comparison result is true, then it returns 1, else it returns 0.

Table 6.2 shows the complete set of relational operators and their symbols for your reference. For beginners, the symbols may look uncommon but they do have a similar meaning compared to mathematical operators.

The double equal to operator (`==`) checks the similarity of two operands. For example, if  $a = 5$  and  $b = 5$ , then  $a == b$  returns 1, which means both contain the same value. Similarly, for the same value of  $a$  and  $b$ ,  $a != b$  returns 0. Let us take some programming examples (Figure 6.3) to understand the use and features of these relational operators.

TABLE 6.2

Relational Operators

| Sl. No. | Relational Operator Name  | Symbol Used in C | Similar Mathematical Symbol |
|---------|---------------------------|------------------|-----------------------------|
| 1       | Less than                 | <                | <                           |
| 2       | Greater than              | >                | >                           |
| 3       | Less than or equal to     | <=               | ≤                           |
| 4       | Greater than or equal to  | >=               | ≥                           |
| 5       | Not equal to              | !=               | ≠                           |
| 6       | Double equal to (similar) | ==               |                             |

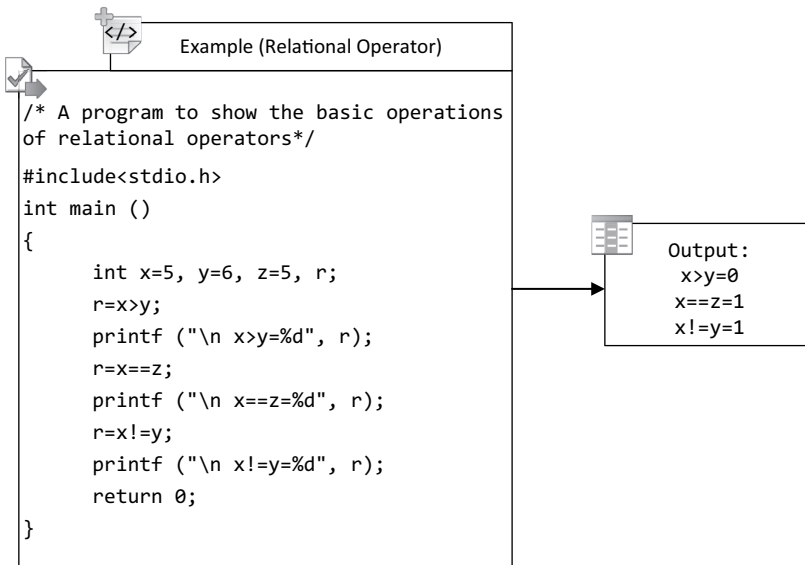


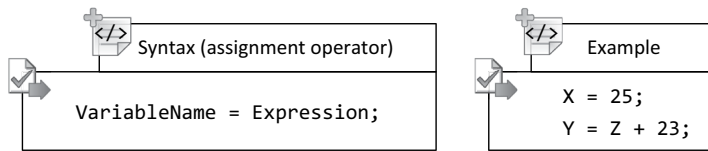
FIGURE 6.3

Program showing operations of relational operators.

## 6.4 Assignment Operators

The symbol = is known as the assignment operator in C. The value (or expression) on the right side of = is assigned to the left side variable. The general form of the assignment operator is shown in Figure 6.4.

- The assignment operator is used to assign a value or the result of an expression to a variable;
- Always the right-hand-side value (in C it is called *rvalue*) is assigned to the left-hand (*lvalue*) variable;



**FIGURE 6.4**  
Assignment operator.

**TABLE 6.3**

Shorthand Assignment Operators

| Statement with Simple Assignment Operator | Equivalent Shorthand Assignment Operator |
|-------------------------------------------|------------------------------------------|
| A=A+1                                     | A+=1                                     |
| A=A-1                                     | A-=1                                     |
| A=A*(N-1)                                 | A*=(N-1)                                 |
| A=A/(N-1)                                 | A/=(N-1)                                 |
| A=A%B                                     | A%=B                                     |

- In the above example, X and Y are known as the *lvalue*, and 25 and Z+23 are known as the *rvalue*.

C has a set of *shorthand assignment operators*. Table 6.3 shows statements with a *simple assignment operator* and their equivalent *shorthand assignment operator*.

The advantage of using a shorthand assignment operator is:

- What appears on the left-hand side need not be repeated;
- The statement is more concise.

## 6.5 Logical Operators

We use logical operators to check the truth and falsity between two expressions or two operands. The C language provides three logical operators, as shown in Table 6.4.

**TABLE 6.4**

Logical Operators

| Sl. No. | Operators | Meaning   |
|---------|-----------|-----------|
| 1       | &&        | Logic AND |
| 2       |           | Logic OR  |
| 3       | !         | Logic NOT |



**TABLE 6.5**

Truth Table of the (a) AND, (b) OR, and (c) NOT Operators

| <b>(a)</b> |           |                       |
|------------|-----------|-----------------------|
| <b>A</b>   | <b>B</b>  | <b>A &amp;&amp; B</b> |
| 0          | 0         | 0                     |
| 0          | 1         | 0                     |
| 1          | 0         | 0                     |
| 1          | 1         | 1                     |
| <b>(b)</b> |           |                       |
| <b>A</b>   | <b>B</b>  | <b>A    B</b>         |
| 0          | 0         | 0                     |
| 0          | 1         | 1                     |
| 1          | 0         | 1                     |
| 1          | 1         | 1                     |
| <b>(c)</b> |           |                       |
| <b>A</b>   | <b>!A</b> |                       |
| 0          | 1         |                       |
| 1          | 0         |                       |

- The logical operator && and || act upon two operands, but the logical NOT operator acts upon one operand;
- The results of these operators are either true or false.

The truth table of the AND, OR, and NOT operators is shown in Table 6.5.

- In logical AND, when both the inputs are 1, the output is 1, and 0 otherwise;
- In logical OR, when both the inputs are 0, the output is 0, and 1 otherwise;
- For the NOT operator, when the input is 1, the output is 0, and vice versa.

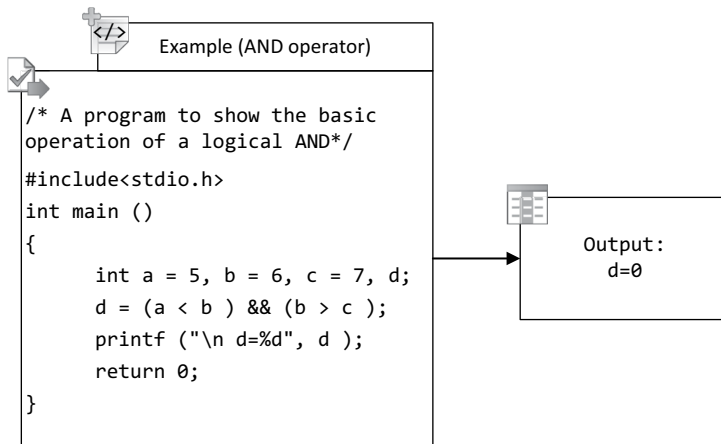
Let us take one programming example and analyze how the compiler executes the code. Figure 6.5 shows an example program to understand the working of the && operator.

The output of the above program will be 0 (zero). Because,  $a < b$  it will yield 1 and  $b > c$  will yield 0. Now  $a \&\& b$  will be 0. Finally, 0 is assigned to d. Hence, we get the output  $d = 0$ . See Figure 6.6 for easy understanding. The numbers 1 through 4 represent the sequence of execution.

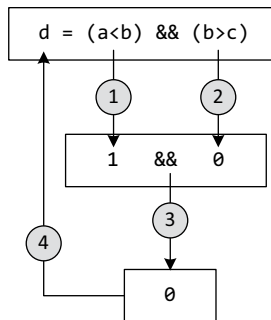
---

## 6.6 Increment and Decrement Operators

The C language provides two special operators ++ and --, called increment and decrement operators. These are unary operators because they operate only on one operand. The operand must be a variable and not a constant.



**FIGURE 6.5**  
Program showing the operation of Logical AND.



**FIGURE 6.6**  
Execution steps of example program.

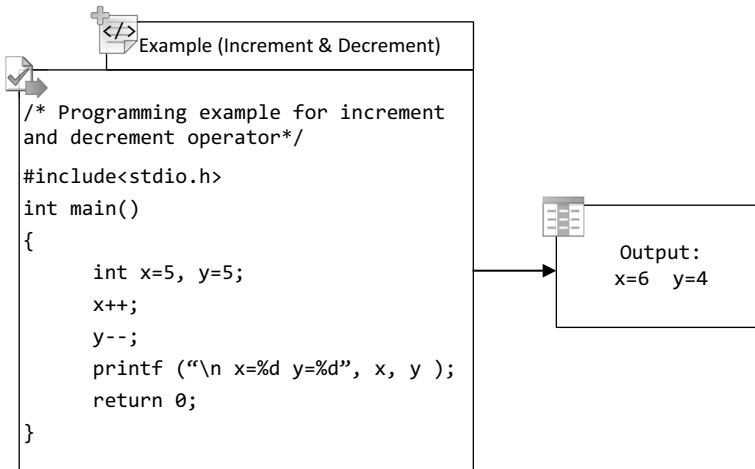
Example:

1. `x++` is valid but `5++` is not valid;
2. `--y` is valid but `--7` is not valid.

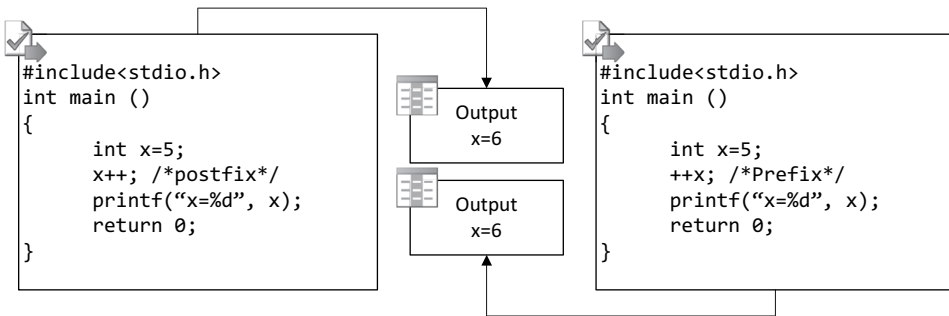
The `++` operator will increment the value of a variable by 1, and the `--` operator will decrement the value of a variable by 1. Let us take a simple example to understand how it works (see Figure 6.7).

In the above example, the line `x++` is executed as `x = x + 1` and the line `y--` is executed as `y = y - 1`. So, the output is `x = 6, y = 4`. These operators can be used either before (prefix) or after (postfix) their operands. So, we can have `x++` (postfix) and `++x` (prefix).

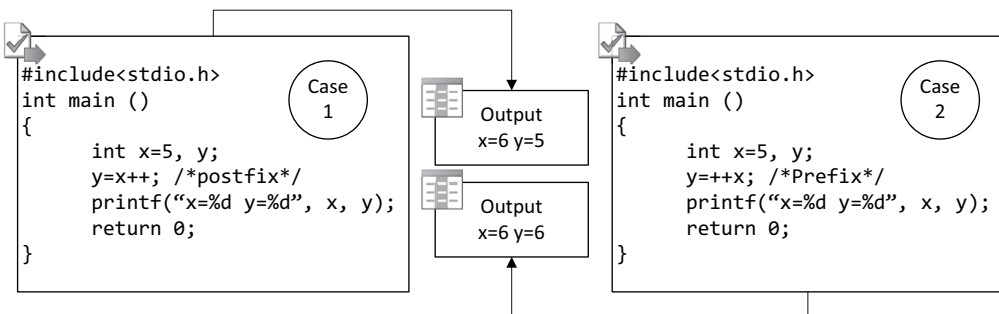
- Prefix and postfix operators have the same effect if they are used in a separate C statement: in Figure 6.8, `x++` and `++x` both execute as `x = x + 1`, and produce 6 as the output.
- Prefix and postfix operators have different effects when used in association with some other variable. For instance, in Figure 6.9, `y` represents a variable that holds the value of `x++` in case 1, and `++x` in case 2. The execution process is explained below.



**FIGURE 6.7**  
Program showing increment and decrement operators.



**FIGURE 6.8**  
Increment or decrement operator executed separately.



**FIGURE 6.9**  
Increment and decrement operator executed in association with variable y.

```

Quiz 1
#include<stdio.h>
int main ()
{
 int x=5, y=5;
 printf(“%d”, ++x);
 printf(“%d”, y++);
 return 0;
}

Quiz 2
#include<stdio.h>
int main ()
{
 int x=5, y=5;
 printf(“%d %d”, ++x, x);
 printf(“%d %d”, y++, y);
 return 0;
}

Quiz 3
#include<stdio.h>
int main ()
{
 int x=5, y;
 y=x++ + ++x + ++x + x++;
 printf(“%d %d”, x, y);
 return 0;
}

```

**FIGURE 6.10**  
Quiz questions.

Case 1 ( $y=x++;$ ): as it is postfix, first, the value of  $x$  is assigned to  $y$ , then  $x$  gets incremented. Hence  $y = 5$  and  $x = 6$ .

Case 2 ( $y=++x;$ ): as it is prefix, first, the value of  $x$  is incremented and  $x$  becomes 6, then the incremented value of  $x$  is assigned to  $y$ . Hence,  $y = 6$  and  $x = 6$ .

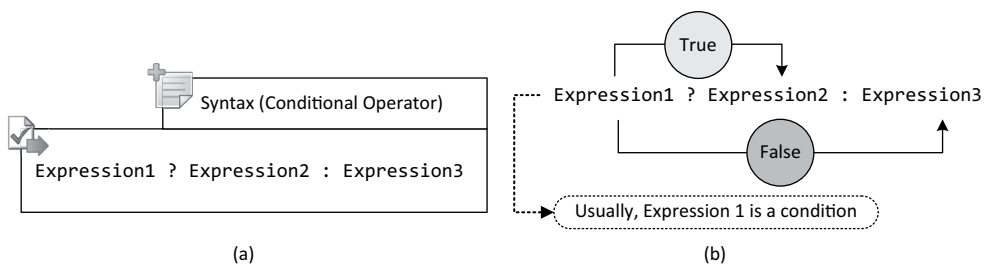
The decremented operators are used in a similar way, except of course the values of  $x$  and  $y$  are decremented.

**Quiz:** Analyze the programs shown in Figure 6.10 and find the outputs.

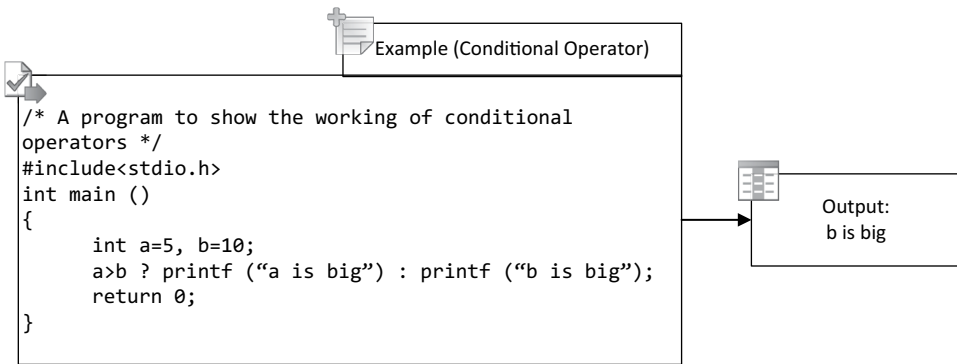
## 6.7 Conditional Operators

The conditional operator is also known as a *ternary operator* because it has two operators and can take three operands. The general form of the conditional operator is shown in Figure 6.11(a).

In the syntax above, “?” and “:” are two operators, and Expression1, Expression2, and Expression3 are the operands. Generally, Expression1 is a condition. If the condition is true, Expression2 gets executed, else Expression3 gets executed. Let us take some programming examples to understand the working of conditional operators. Figure 6.12 shows the first example: to find the bigger among two numbers.



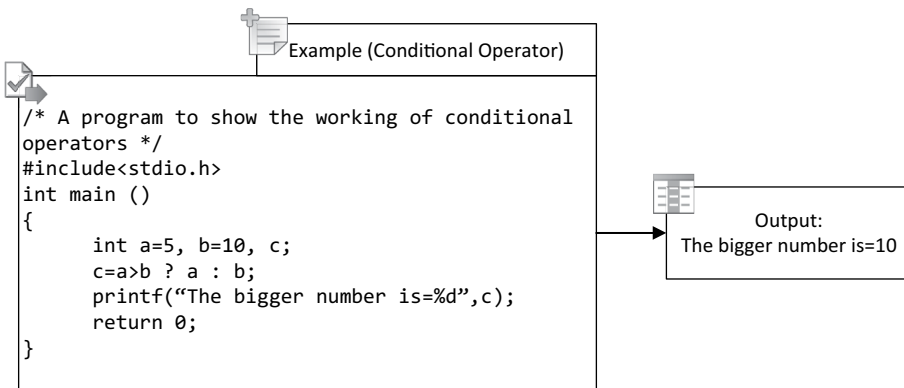
**FIGURE 6.11**  
(a) Syntax of Conditional operator; (b) Working of Conditional operator



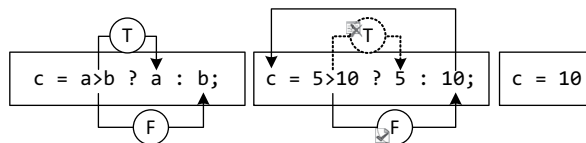
**FIGURE 6.12**  
Finding the bigger number among two numbers using conditional operators.

In this example,  $a > b$  represents a condition, and it is false because 5 is not greater than 10. So, `printf ("b is big")` gets executed. Hence, the output is: b is big.

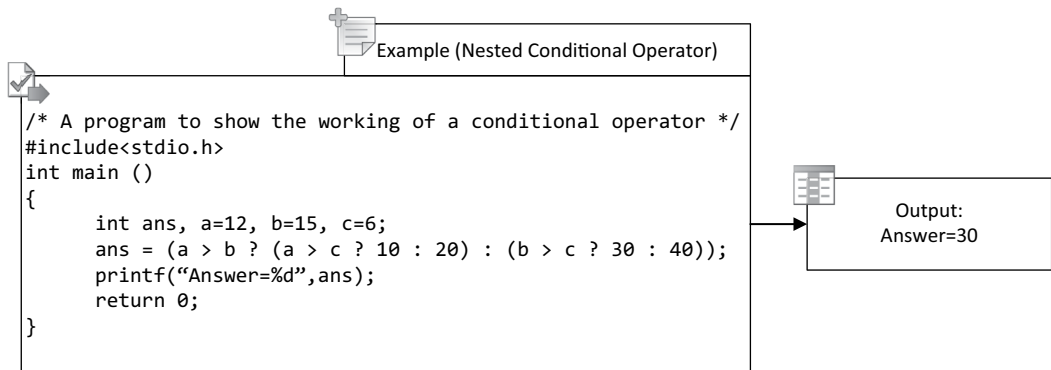
Let us write the same program in a different way as shown in Figure 6.13. In this example, first, the condition  $a > b$  is executed, and the compiler takes the decision. As  $a > b$  is false, so the value of  $b$  is returned and is assigned to  $c$ . Hence,  $c$  has the value 10. In the next line, the value of  $c$  gets printed, which is the bigger number. Figure 6.14 shows the working steps.



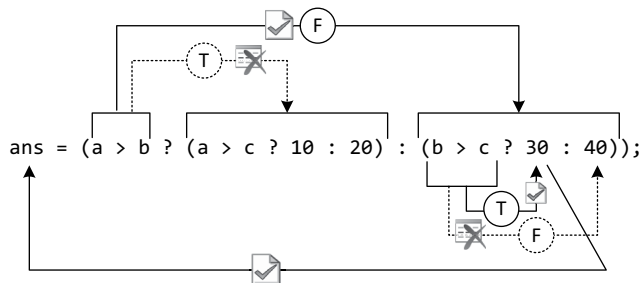
**FIGURE 6.13**  
Finding the bigger number among two numbers using conditional operators (alternative way).



**FIGURE 6.14**  
Execution procedure of the example shown in Figure 6.13.



**FIGURE 6.15**  
Nested conditional operator example.



**FIGURE 6.16**  
Execution process of the example in Figure 6.15.

### 6.7.1 Nested Conditional Operators

The conditional operator can be nested, that is, we can declare one conditional operator statement within another conditional operator. We can easily see this with a programming example. Figure 6.15 shows an example that uses the concept of a nested conditional operator. The execution and working of the program are shown in Figure 6.16.

According to the value of  $a$  and  $b$ , the statement  $a > b$  becomes false, so the false part of the statement is executed as shown in Figure 6.16. Again  $b > c$  becomes true so 30 will be returned and hence is assigned to the variable  $ans$ . So, the output will be 30.

## 6.8 Bitwise Operators

Each digit in a binary number system is called a bit, and is either 0 or 1. As the name suggests, the computer uses a bitwise operator to operate on binary numbers. The C language provides six bitwise operators; Table 6.6 shows their symbols and meanings.

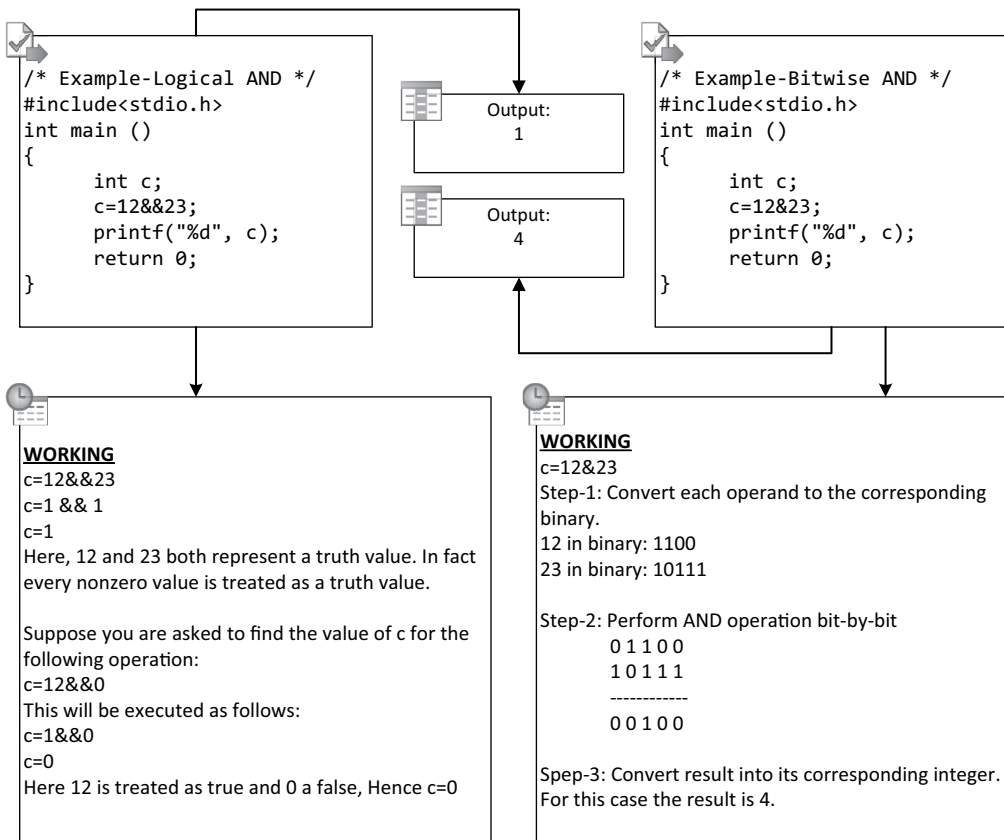
**TABLE 6.6**  
Bitwise Operators

| Sl. No. | Operator Symbol | Meaning          |
|---------|-----------------|------------------|
| 1       | &               | Bitwise AND      |
| 2       |                 | Bitwise OR       |
| 3       | ^               | Bitwise XOR      |
| 4       | ~               | One's complement |
| 5       | <<              | Left-shift       |
| 6       | >>              | Right-shift      |

**6.8.1 Bitwise AND, OR, XOR**

Like the logical AND operator, bitwise AND follows the same truth table and requires two operands. But the working procedure of this operator is different. The former works on any value and returns either 0 or 1, but the latter converts the value to its corresponding binary number and performs the AND operation bit by bit.

Let us take an example (Figure 6.17) to show the differences between their working.

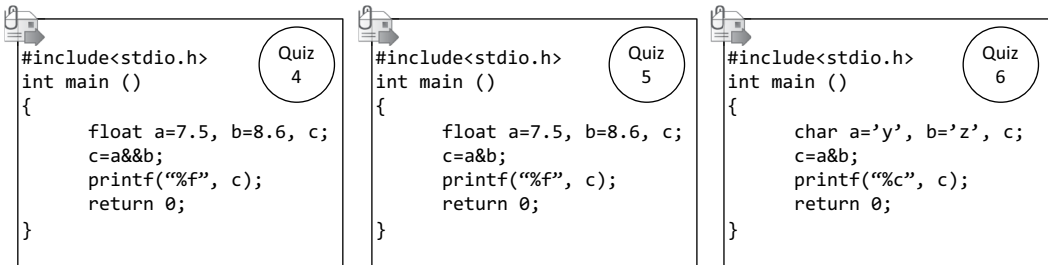


**FIGURE 6.17**  
Example showing the difference between the logical AND and Bitwise AND operators.

TABLE 6.7

XOR Truth Table

| A | B | A^B |
|---|---|-----|
| 0 | 0 | 0   |
| 0 | 1 | 1   |
| 1 | 0 | 1   |
| 1 | 1 | 0   |



**FIGURE 6.18**  
Quiz questions.

Similarly, for the OR and XOR operators, the working procedure is the same. Students are encouraged to modify the above program and check for the OR and XOR operators. The XOR operator works as per the truth table shown in Table 6.7. When both the inputs are the same, the output will be 0, and 1 otherwise.

**Quiz:** What is the output or error you will get when you execute the program code shown in Figure 6.18?

### 6.8.2 One's Complement (~) Operator

One's complement converts the bit from 0 to 1 and vice versa. This is a unary operator and works upon one operand. It first translates the operand to its corresponding binary and flips the bits. Then we can display the result as per our requirement. Let us take some examples and understand its working style (Figure 6.19).

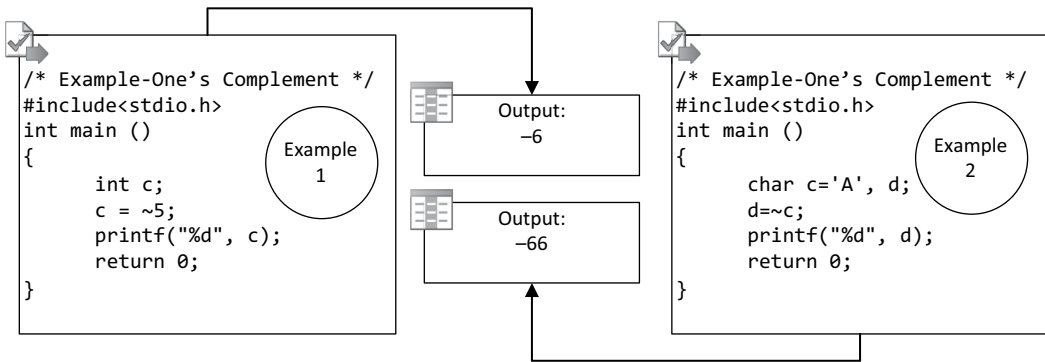
To understand the working of this operator, we need to know how the computer stores a negative number. There is a concept called two's complement representation, which the computer uses to store a negative number, as shown in the following section.

### 6.8.3 Two's Complement Representation

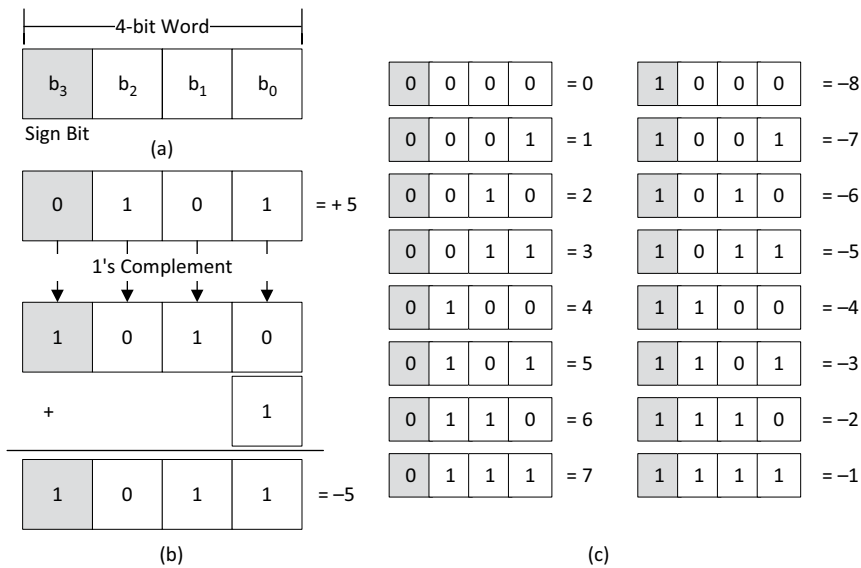
With this technique the most significant bit (leftmost bit) acts as a sign bit. When the leftmost bit (sign bit) is "0", the number is positive, and for "1", the number is negative. The remaining bit of the word is used to represent the actual number. Figure 6.20 shows the representation style.

As shown in Figure 6.20a, a four-bit word is used to represent a number. The leftmost bit ( $b_3$ ) is reserved for the sign bit and the remaining bits (i.e., from  $b_0$  to  $b_2$ ) are used to represent the number.





**FIGURE 6.19**  
One's complement operator example.



**FIGURE 6.20**  
Two's complement representation.

- To represent a positive number, we simply convert it to its equivalent binary number;
- To represent a negative number, we first convert the number to its binary form and then apply two's complement;
- To find two's complement, we add 1 to the one's complement of that number.

Figure 6.20b shows the representation of -5. First, we find the binary equivalent of 5 which is 0101 in four bits. Then we flip the bits (1 to 0 and 0 to 1) to get the one's complement (i.e., 1010). Finally, we add 1 to the one's complement to get the resultant two's complement which is 1011. Hence, -5 is represented as 1011 in two's complement representation. Figure 6.20c shows the complete range of numbers that can be represented with four bits using two's complement representation. The complete range is -8 to +7 with a single

representation of 0's. In general, if an  $n$ -bit word is given, then the range will vary from  $-(2^{n-1})$  to  $+(2^{n-1} - 1)$ .

Now that we understand the concept of two's complement representation, let us see the working of the examples shown in Figure 6.19. Assume that our system uses 16 bits to represent a number. In Example 1, the value of  $d$  is assigned with  $\sim 5$ . So, the compiler first converts the number to its equivalent binary 16-bit number:

- 5 in 16-bit binary representation:

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

- Then the compiler flips the bit to find its one's complement which is:

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

By observing the sign bit, we can easily say that it is a negative number. If you analyze it carefully, you will find that it is the two's complement representation of  $-6$ . Hence the output is  $-6$ .

Similarly, for Example 2, the character  $c$  is assigned with "A", and A's ASCII value is 65. So, the compiler first converts 65 to its equivalent 16-bit binary number.

- 65 in 16-bit binary representation:

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

- Then the compiler flips the bit to find its one's complement which is:

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

- By observing the sign bit, we can easily say that it is a negative number. If you analyze it carefully, you will find that it is the two's complement representation of  $-66$ . Hence the output is  $-66$ .

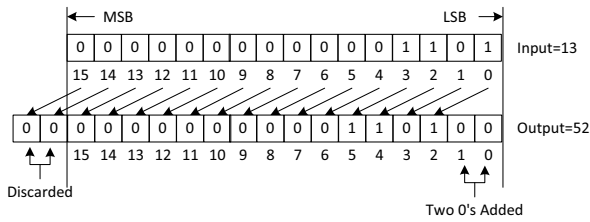
#### NOTE

To easily find the output, add 1 to the operand's value and change the sign. In Figure 6.19 Example 1, the value of the operand is 5, so the output will be  $-(5 + 1) = -6$ .

#### 6.8.4 Left Shift Operator ( $\ll$ ) and Right Shift Operator ( $\gg$ )

This is a binary operator and acts upon two operands. The right operand specifies the number of bits to be shifted left; the same number of 0's will be added from the right. The operation is quite simple. Let us take an example to understand it.

Suppose,  $x = 13$  and we want to perform  $x \ll 2$ . Then every bit will be left shifted by two bits, and two 0's will be added from the right. Figure 6.21 demonstrates this shifting operation.



**FIGURE 6.21**  
Two-bit left-shift operation.

The two most significant bits (MSB) that come out will be discarded, and two 0's will be added from the right to the least significant bit (LSB). The result will be calculated by converting the decimal equivalent of the 0th to 15th bits. Hence, the output in this case will be 52.

**NOTE**

After the left shift, the MSB will be dropped, and after the right shift, the LSB will be dropped.

Users can also find out the result of left shift by just multiplying  $2^n$  and the number, where  $n$  is the number of bits to be shifted. In the above example,  $x$  is shifted by two positions, equivalent to multiplying  $x$  by  $2^2$  ( $13 \times 2^2=52$ ), and the result will be 52.

To test the above concept, let us write a programming example and check whether the result is correct or not. Students are encouraged to execute the following program (Figure 6.22) and check the result.

Let us take another example for negative numbers (Figure 6.23). We know that negative numbers are represented in memory by the two's complement method.

To understand the output, we need to represent  $-13$  using 16-bit representation:

- +13 in 16-bit representation:

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

- One's complement of the above representation:

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

```

/* Example-Left shift operator */
#include<stdio.h>
int main()
{
 int a=13;
 printf("Output=%d",a<<2);
 return 0;
}

```

Output:  
Output=52

**FIGURE 6.22**  
Example program to test the left shift operator.

```

/* Example-Left shift operator */
#include<stdio.h>
int main ()
{
 int a=-13;
 printf("Output=%d",a<<2);
 return 0;
}

```

Output:  
Output=-52

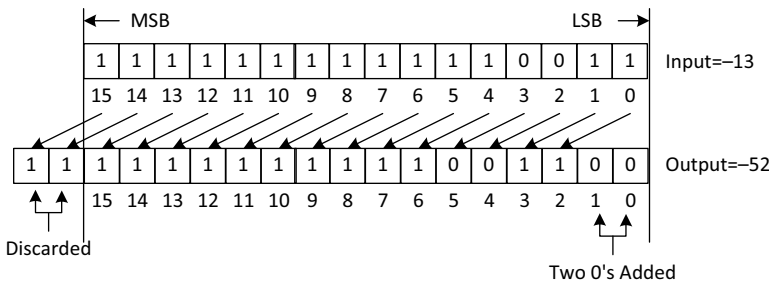
**FIGURE 6.23**  
Example program applying left shift to a negative number.

- Add 1 to the above result to get its two’s complement:

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Finally, the two’s complement of -13 is 111111111110011 which is actually represented in memory. We will apply a two-bit left shift to the above value to find the output (shown in Figure 6.24).

Similarly, for the right shift operator the bits are shifted to the right. Two programming examples and their output are shown in Figure 6.25, which are self-explanatory.



**FIGURE 6.24**  
Two-bit left-shift operation on -13.

```

/*Example-Right shift operator*/
#include<stdio.h>
int main()
{
 int a=13;
 printf("Output=%d",a>>2);
 return 0;
}

```

Output:  
Output=-4

```

/*Example-Right shift operator*/
#include<stdio.h>
int main()
{
 int a=-13;
 printf("Output=%d",a>>2);
 return 0;
}

```

Output:  
Output=3

**FIGURE 6.25**  
Example programs showing the output of the right shift operator.

## 6.9 Special Operators

Beside the operators described above, C supports some special operators that include the following. These operators have several uses, and we will discuss them with some examples that describe the characteristics of these operators.

1. Comma operator;
2. `sizeof()` operator;
3. `&` and `*`;
4. `->` (arrow) and (Dot).

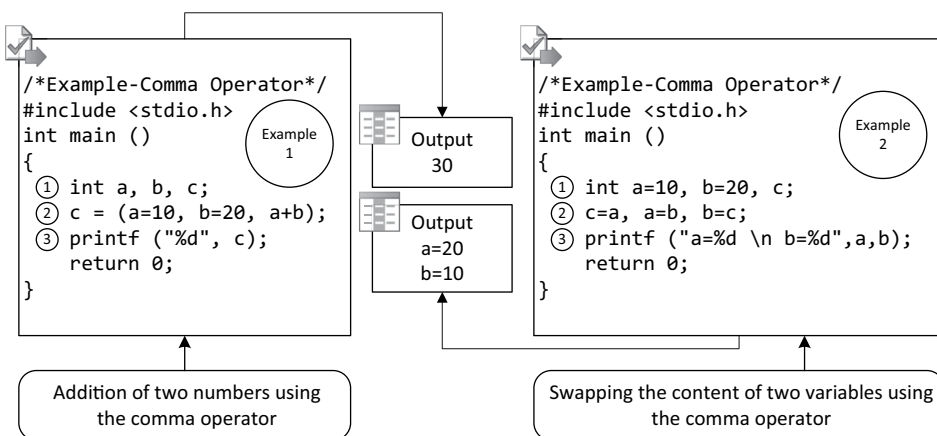
In this section we will discuss the comma operator and the `sizeof` operator. Other operators will be discussed whenever there is a need for them.

### 6.9.1 The Comma Operator

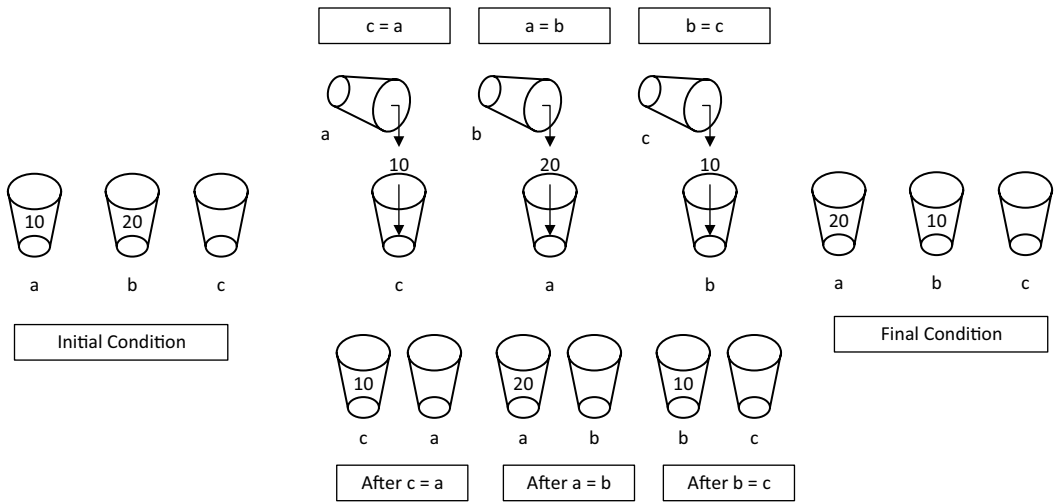
The comma operator permits two different expressions to appear in situations where only one operation is ordinarily used. Comma separated operands can be chained together and evaluated in a left-to-right sequence with the right-most value yielding the result of the expression. Let us see this with an example (Figure 6.26).

As discussed above, the execution of a comma-separated operation will start from the left; the rightmost value yields the result. In Example 1, when line number 2 is executed it assigns 10 to `a` and 20 to `b`. Finally, the statement `a+b` is executed and the result, which is 30, is assigned to `c`. Similarly, in Example 2 when line number 2 is executed, the swapping occurs by assigning `a` to `c`, `b` to `a`, and `c` to `b`. A glass analogy is shown in Figure 6.27 to illustrate how swapping occurs.

A comma operator is used to separate the variable names during the declaration, and also used in a for loop. We will discuss the for loop in Chapter 8.



**FIGURE 6.26**  
Example showing the use of a comma operator.



**FIGURE 6.27** Swapping two numbers using a glass analogy.

```
Example: sizeof operator
#include<stdio.h>
int main()
{
 int s;
 printf("\n%d", sizeof(float));
 printf("\n%d", sizeof(s));
 printf("\n%d", sizeof('A'));
 return 0;
}
```

**FIGURE 6.28** Using a sizeof operator.

### 6.9.2 The sizeof Operator

The sizeof operator returns the number of bytes the operand occupies in memory. The operands may be variables, constants, or data types. Figure 6.28 shows the way a program uses a sizeof operator. Students are instructed to execute this program and analyze the results on their own.

---

## 6.10 Expressions

An expression is a combination of variables, constants, and operators written according to the syntax of the C language. In C every expression evaluates to a value (i.e., every expression results in some value of a certain type that can be assigned to a variable).

**TABLE 6.8**

Examples of C Expressions

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

Some examples of C expression are shown in Table 6.8.

### 6.10.1 Evaluation of Expressions

Expressions are evaluated using an assignment statement of the form:

$$\text{variable} = \text{expression}$$

- The variable is any valid C variable name;
- When the statement is encountered, the expression is evaluated first and then replaces the variable's previous value on the left-hand side;
- All variables used in the expression must be assigned values before evaluation is attempted;
- We can write only one variable on the left-hand side of = ; that is, the expression  $x = k * i$  is legal, whereas  $k * i = x$  is illegal.

### 6.10.2 Rules for Evaluation of Expressions

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

## 6.11 Type Conversion

Sometimes we need to convert the value of an expression from one data type to another, known as type conversion. Let us take an example (Figure 6.29) that shows the concept of type conversion.

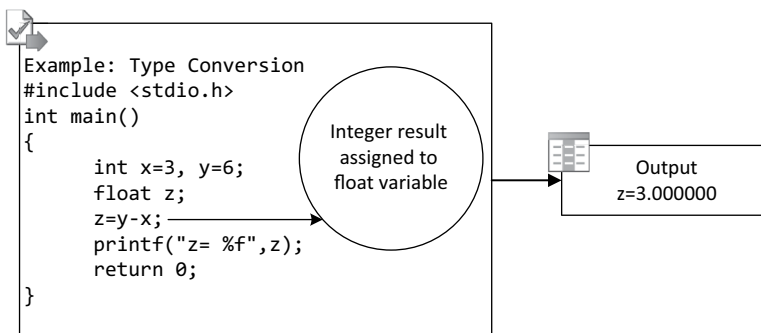
In this program an automatic type conversion is made by the compiler (i.e., from integer to float conversion) and this is called *type conversion* or *type casting*.

Type casting in C is of two types: implicit and explicit.

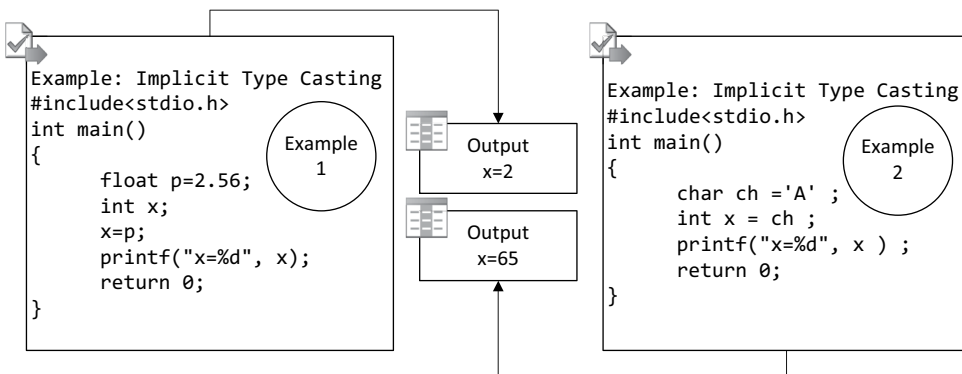
### 6.11.1 Implicit Type Casting

C permits the mixing of constants and variables of different types in an expression. C automatically converts any intermediate values to the proper type so that the expression can be evaluated without losing any significance. This automatic type conversion is known as *implicit type conversion*.

Figure 6.30 shows an example. In the program (Example 1), p is a float variable, and we assign the value 2.56 to it. Here x is an integer. We assign the value of p to x, so the value of p (i.e., 2.56) will automatically be converted into integer value 2. So, the output will be 2.

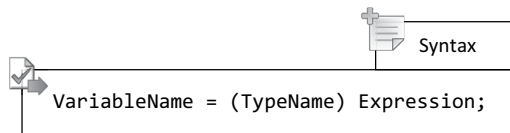


**FIGURE 6.29**  
Type conversion.

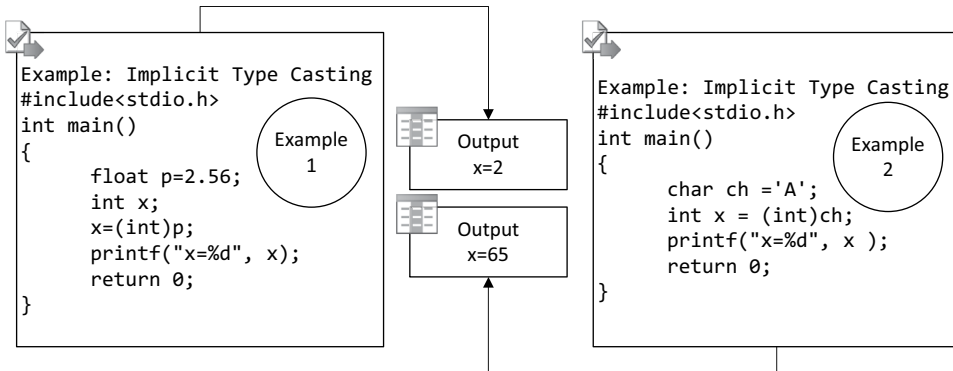


**FIGURE 6.30**  
Implicit type conversion example.





**FIGURE 6.31**  
Syntax of explicit type conversion.



**FIGURE 6.32**  
Example of explicit type conversion.

Similarly, in Example 2, the character variable `ch` will be assigned with `A`. When we try to assign the value of `ch` to an integer variable `x`, then implicitly the ASCII value of `A`, an integer, is assigned to `x`. So, the output will be 65.

### 6.11.2 Explicit Type Conversion

Often there may arise a situation where we want to force a type conversion in a way that is different from automatic conversion. The syntax for doing this is shown in Figure 6.31.

Figure 6.32 shows the code for how to use the syntax to convert the type explicitly. Students are advised to execute the program and see the result.

## 6.12 Operator Precedence and Associativity

Precedence helps us to decide which operation to perform first if our expression contains several operations. For instance, in the following expression, several multiplications, divisions, and other operations are present:

$$R = 3 + 5 * 8 / 2 - 6 + 7 * 9 / 3$$

Hence, we require a precedence rule to follow. As in mathematics, we follow the BODMAS rule that specifies the precedence of operators. There are several operators in the C programming language; and we follow the precedence shown in Table 6.9.

**TABLE 6.9**

Precedence and Associativity Table

| Operators | Type                                 | Associativity |
|-----------|--------------------------------------|---------------|
| ()        | Parentheses (function call operator) | Left to right |
| []        | Array subscript                      |               |
| .         | Member selection via object          |               |
| ->        | Member selection via pointer         |               |
| ++        | Post-increment                       |               |
| --        | Post-decrement                       |               |
| ++        | Pre-increment                        | Right to left |
| --        | Pre-decrement                        |               |
| +         | Unary plus                           |               |
| -         | Unary minus                          |               |
| !         | Unary logical negation               |               |
| ~         | Unary bitwise complement             |               |
| (type)    | Cast                                 |               |
| *         | Dereference                          |               |
| &         | Address                              |               |
| sizeof    | Determine size in bytes              |               |
| *         | Multiplication                       | Left to right |
| /         | Division                             |               |
| %         | Modulus                              |               |
| +         | Addition                             | Left to right |
| -         | Subtraction                          |               |
| <<        | Bitwise left shift                   | Left to right |
| >>        | Bitwise right shift                  |               |
| <         | Less than                            | Left to right |
| <=        | Less than or equal to                |               |
| >         | Greater than                         |               |
| >=        | Greater than or equal to             |               |
| ==        | Equal to                             | Left to right |
| !=        | Not equal to                         |               |
| &         | Bitwise AND                          | Left to right |
| ^         | Bitwise exclusive OR                 | Left to right |
|           | Bitwise OR                           | Left to right |
| &&        | Logical AND                          | Left to right |
|           | Logical OR                           | Left to right |
| ?:        | Conditional operator                 | Right to left |
| =         | Assignment                           | Right to left |
| +=        | Addition assignment                  |               |
| -=        | Subtraction assignment               |               |
| *=        | Multiplication assignment            |               |
| /=        | Division assignment                  |               |
| %=        | Modulus assignment                   |               |
| &=        | Bitwise AND assignment               |               |
| ^=        | Bitwise XOR assignment               |               |
| =         | Bitwise OR assignment                |               |
| <<=       | Bitwise left shift assignment        |               |
| >>=       | Bitwise right shift assignment       |               |
| ,         | Comma                                | Left to right |

|                                                                                     |                                                                                    |                                                                                    |                                                                                    |                                                                                     |
|-------------------------------------------------------------------------------------|------------------------------------------------------------------------------------|------------------------------------------------------------------------------------|------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
| $\begin{aligned} 18 / 3 * 10 / 5 \\ = 18 / 30 / 5 \\ = 18 / 6 \\ = 3 \end{aligned}$ | $\begin{aligned} 18 / 3 * 10 / 5 \\ = 18 / 3 * 2 \\ = 6 * 2 \\ = 12 \end{aligned}$ | $\begin{aligned} 18 / 3 * 10 / 5 \\ = 18 / 3 * 2 \\ = 18 / 6 \\ = 3 \end{aligned}$ | $\begin{aligned} 18 / 3 * 10 / 5 \\ = 6 * 10 / 5 \\ = 6 * 2 \\ = 12 \end{aligned}$ | $\begin{aligned} 18 / 3 * 10 / 5 \\ = 6 * 10 / 5 \\ = 60 / 5 \\ = 12 \end{aligned}$ |
| (a)                                                                                 | (b)                                                                                | (c)                                                                                | (d)                                                                                | (e)                                                                                 |

**FIGURE 6.33**  
Execution of an expression in different orders.

Following the precedence is not enough to execute an expression. Consider the following example:

$$R = 18 / 3 * 10 / 5$$

There are several ways you can execute this expression if you follow the concept of precedence because \* and / have the same precedence. Hence you get a different result for different executions, as shown in Figure 6.33.

Figure 6.33 shows several possible ways a person can execute the above expression. However, Figure 6.33a–d are not correct even though you get correct answers for some executions; the correct one is shown in Figure 6.33(e), which follows an execution pattern, from left to right. There are three operations to choose from:  $18/3$ ,  $3*10$ , and  $10/5$ . If you go from left to right, the first operation encountered is  $18/3$ , and you execute it. Now you have two operations left to choose from,  $6*10$  or  $10/5$ . You choose the first one to perform because this is the first operation you encounter when you go from left to right. Finally, you have only one operation to complete (i.e.,  $60/5$ ), and you perform it to get your result. The rule that you follow here (left to right execution rule) is known as the “rule of associativity.” This also plays an important role in executing expressions in C where the precedence of operators is the same.

To elaborate further, let us take another example. Assume that we have three variables, a, b, and c, with values 5, 6, and 7, respectively. Let us perform the following operation:

$$a = b = c$$

We cannot execute this expression from left to right, so we follow a right to left execution flow and end with a, b, and c having the values 7, 7, and 7, respectively.

From the above two examples, we find that not only the precedence of operators is important; rather, associativity plays an essential role in executing expressions. Table 6.9 shows this precedence as well as its associativity.

## 6.13 Review Questions

### 6.13.1 Objective Type Questions

1. In the C programming language, the division operation returns \_\_\_\_\_, and the modulus operator returns \_\_\_\_\_.
2. The result of relational operators in C is either \_\_\_\_\_ or \_\_\_\_\_.
3. \_\_\_\_\_ is the short form of the expression  $x=x*(y+z)$ .

4. In logical AND operation, when both inputs are 1, the output is \_\_\_\_\_.
5. In logical OR operation, when both inputs are 0, the output is \_\_\_\_\_.
6. Operators ++ and -- are unary operators. True/false?
7. The conditional operator requires two operators: \_\_\_\_\_ and \_\_\_\_\_.
8. How many bitwise operators are there in C?
9. Two's complement representation of -23 is \_\_\_\_\_.
10. \_\_\_\_\_ operator is used to find the size of any variable.
11. C language supports two kinds of type casting: \_\_\_\_\_ and \_\_\_\_\_.
12. \_\_\_\_\_ rule helps us to decide which operation to perform first if our expression contains several operations.
13. \_\_\_\_\_ rule helps us to decide which operation to perform first if our expression contains operators with the same precedence.

### 6.13.2 Programming Questions

1. Write a program to convert from centigrade to fahrenheit and vice versa.
2. Write a program to calculate the area of a triangle if three sides are given.  
**Hint:** Use the `sqrt()` function and `#include <math.h>`
3. Write a program to calculate the area of a triangle where the base and height are given as arguments.
4. Write a program to calculate the gross salary of a person if the user enters the basic salary. Dearness Allowance (DA) should be 50 percent of basic salary, and House Rent Allowance (HRA) is 10 percent of basic salary.
5. Write a program to check whether a number is even or odd using conditional operators.
6. Write a program to find the biggest number among three numbers using conditional operators.
7. Write a program to enter a four-digit number from the keyboard. Add the 1st and 4th digit of the number entered, and print the result.
8. Write a program to convert radians to degrees and vice versa.
9. Write a program to check whether a year is a leap year or not. Use conditional operators.  
**Hint:** `(year % 400 == 0 || (year % 100 != 0 && year % 4 == 0))` then the year is a leap year.
10. Analyze the following programs to find the errors or outputs.

```
#include<stdio.h>
void main()
{
 int a=10,b=20,c;
 c=(b%a);
 printf("%d",c);
}
```

```
#include<stdio.h>
void main()
{
 int a=5,b=6,c;
 c=(a>b) && (b=99);
 printf("%d%d%d",a,b,c);
}
```

```
#include<stdio.h>
void main()
{
 int a=5,b=6,c;
 c=(b>a)&&(b=99);
 printf("%d%d%d",a,b,c);
}
```

```
#include<stdio.h>
void main()
{
 int a=5,b=6,c;
 c=(a>b)|| (b=99);
 printf("%d%d%d",a,b,c);
}
```

```
#include<stdio.h>
void main()
{
 int a=5,b;
 b=a++ + ++a + a++ + ++a;
 printf("%d %d",a,b);
}
```

```
#include<stdio.h>
void main()
{
 int a=5,b;
 b=++a + --a + ++a + a--;
 printf("%d %d",a,b);
}
```

```
#include<stdio.h>
void main()
{
 int a=-25,b;
 b=a>>3;
 printf("%d %d",a,b);
}
```

```
#include<stdio.h>
void main()
{
 int a=5;
 printf("%d %d %d",a++, a++, a++);
}
```

```
#include<stdio.h>
void main()
{
 int a=5,b=6,c;
 c=(b>a)|| (b=99);
 printf("%d%d%d",a,b,c);
}
```

```
#include<stdio.h>
void main()
{
 int a=5,b;
 b=a++ + a++ + a++;
 printf("%d %d",a,b);
}
```

```
#include<stdio.h>
void main()
{
 int a=5,b;
 b=++a + ++a + ++a + ++a;
 printf("%d %d",a,b);
}
```

```
#include<stdio.h>
void main()
{
 int a=-25,b;
 b=a<<3;
 printf("%d %d",a,b);
}
```

```
#include<stdio.h>
void main()
{
 int a=6,b=7,c=8,d;
 d=a>b?a>c?a:b:c>a?a:c;
 printf("%d",d);
}
```

```
#include<stdio.h>
void main()
{
 int a=5;
 printf("%d %d %d",a++, ++a, a++);
}
```

### 6.13.3 Subjective Type Questions

1. What are operators and operands? Explain unary, binary, and ternary operators with appropriate examples.
2. Explain the list of arithmetic operators supported by the C programming language. Explain with an example to show their differences.

3. What are relational operators? List all relational operators found in the C language.
4. What is the difference between logical operators and bitwise operators? How do they function? Explain with proper programming examples.
5. Write short notes on increment and decrement operators in C.
6. What is a conditional operator? Explain its syntax with an appropriate example.
7. Can a conditional operator be nested? If yes, explain with an appropriate example.
8. Explain two's complement representation of negative numbers with an example.
9. List the special operators supported by C. Write at least two example uses of the comma operator.
10. What is a sizeof operator? Write some programs to show how the sizeof operator works.
11. Explain the rules for evaluating an expression in C.
12. What is type casting in C? Explain implicit and explicit type casting with examples.
13. What are precedence and associativity? Explain both these terms with appropriate examples.
14. What are the benefits of using shorthand notations while writing expressions in C?



**Taylor & Francis**

Taylor & Francis Group

<http://taylorandfrancis.com>

# 7

## Basic Input/Output

### 7.1 Introduction

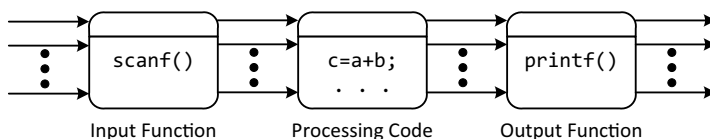
Every program receives some input, processes it, and produces output. The C language provides us with several predefined functions that help us supply input to the program and generate output. Up until now, we have come across two such functions: `scanf()` and `printf()`. To supply input to a program, we need a `scanf()` function; a program produces output through a `printf()` function. The overall structure of a program may take the form shown in Figure 7.1.

Besides the above functions, C provides other I/O (input/output) functions. These are called predefined functions or library functions. The compiler knows how they work and offers us a syntax to write them, such as the `printf()` and `scanf()` functions. This chapter provides a detailed understanding of these functions, their syntax, and how to use them in our programs.

There are numerous library functions available for I/O. They are classified into three groups:

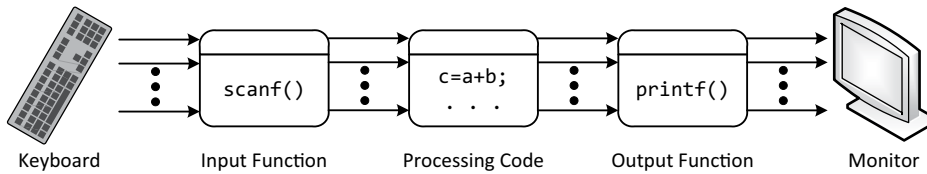
1. *Console I/O functions:* These functions take input from a keyboard and produce output through a display device like a monitor. Figure 7.2 shows an overall visualization of this process.
2. *Disk or file I/O functions:* We generally store files on our hard disk drive. We can write information on a file and read data from a file too. To read or write data from a file, we need some functions that are called file I/O functions. C language provides many such functions, which we will discuss in Chapter 14.
3. *Port I/O functions:* Several other functions are also available in C to perform I/O operations on ports, though those functions are out of the scope of this book.

This chapter concerns console I/O functions, divided into different categories, as shown in Figure 7.3.

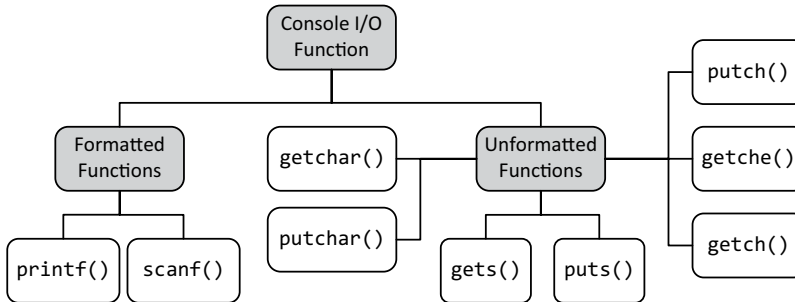


**FIGURE 7.1**  
Three-step program structure.





**FIGURE 7.2**  
Overall visualization of console I/O functions.



**FIGURE 7.3**  
Classification of console I/O functions.

Most of a function's definition is present in the `<stdio.h>` header file. But there are some functions whose definition is not present there. Hence, I would like to introduce another header file called "console input output" which is represented as `<conio.h>`. This header file contains the definitions of all console I/O functions. Hence, when you use any of these functions in your program, it is recommended to include this header file using the following line above the `main()` function:

```
#include<conio.h>
```

## 7.2 Unformatted Functions

Many functions come under this category, and we use them either to read a single character through the keyboard or a string of characters. Similarly, we use some functions to write a single character on the screen or a string of characters. Let us discuss the detail of these functions.

### 7.2.1 `getchar()` and `putchar()`

These two functions help in reading a character from the standard input unit (usually a keyboard) and writing it to the standard output unit (usually a screen).

`getchar()`: Reads a character from the standard input device.

`putchar()`: Writes a character to the standard output device.

1. The `getchar()` function takes the form:

```
varName=getchar();
```

`VarName` is a character variable. When a user types a character in the keyboard, then the `getchar()` function scans that character and assigns it to the `VarName`.

2. The `putchar()` function takes the form:

```
putchar(varName);
```

Here `VarName` is a character variable that contains a character. The `putchar()` function will send the value of the variable to the output unit for printing on the screen.

Let us take an example to explain the working of these two functions. Program 7.1 shows an example that declares a character variable `ch`, assigns a character using the `getchar()` function, and displays the character using the `putchar()` function.

### 7.2.2 `gets()` and `puts()`

These two functions help in reading a string from the standard input unit (usually a keyboard) and writing it to the standard output unit (usually a screen).

1. `gets()` reads a string from the standard input device. It accepts the name of a string as a parameter and fills the string with characters that are input from the keyboard.

#### PROGRAM 7.1

```
1. #include <stdio.h>
2. #include <conio.h>
3. int main()
4. {
5. char ch;
6. printf("Type one Character: ");
7. ch=getchar();
8. printf("The typed character is: ");
9. putchar(ch);
10. return 0;
11. }
```

#### *Output:*

```
Type one Character: p
The typed character is: p
```

#### *Explanation:*

Line 5: Declare `ch` as a character.

Line 7: Read the typed character from the keyboard and assign it to `ch`.

Line 9: Display the character present in `ch` on the screen.

2. `puts()` writes a string to the standard output device. It accepts the name of a string and displays the accepted string on the screen.

Program 7.2 shows an example of how to use the `gets()` and `puts()` functions for reading and writing a string.

### PROGRAM 7.2

```
1. #include <stdio.h>
2. #include <conio.h>
3. int main()
4. {
5. char str[50];
6. printf("Type a string[less than 50 characters]: ");
7. gets(str);
8. printf("\nThe string typed is: ");
9. puts(str);
10. return 0;
11. }
```

#### *Output:*

Type a string [less than 50 characters]: C programming Learn to Code  
The string typed is: C programming Learn to Code

#### *Explanation:*

Line 5: Creates a string `str` which will store 50 characters and allocate memory for them. `str[50]` represents a character array. We will discuss the concept of character array in Chapter 10. For now, just assume that it is a long chain of 50 memory blocks and each block can store one character.

Line 6: Prompt the user to enter a string of 50 characters, and this because we have allocated space for 50 characters only.

Line 7: Read the string from the keyboard when the user types it in and assign it to the previously allocated block.

Line 8: Will display "The string typed is:" for the user to see.

Line 9: Will display the content of `str` which was previously assigned by the `gets()` function. It reads the entire string from the memory and displays it on the screen.

### 7.2.3 `getch()` and `getche()`

These two functions will return the character that has been most recently typed. The "e" in the `getche()` function means it echoes (displays) the character that you typed to the screen. As against this `getch()` just returns the character that you typed without echoing

it. `getch()` simply halts program execution to wait for the user to press a key. In effect it is a “pause” and waits for the user. Program 7.3 shows a simple program to illustrate the difference between these functions.

### PROGRAM 7.3

```

1. #include<stdio.h>
2. #include<conio.h>
3. int main()
4. {
5. char ch;
6. printf("\nI am using the function getche() of C");
7. printf("\nPlease enter a character: ");
8. ch=getche();
9. printf("\nEntered character is %c",ch);
10. printf("\nI am using the function getch() of C");
11. printf("\nPlease enter a character: ");
12. ch=getch();
13. printf("\nEntered character is %c",ch);
14. getch();
15. return 0;
16. }
```

*Output:*

I am using the funtion getche() of C  
Please enter a character: r  
Entered character is r  
I am using the funtion getch() of C  
Please enter a character: u  
Entered character is u

In this case, we typed a character 'r', it is echoed on the screen and displayed.

In this case, we typed a character 'u' but it is not echoed on the screen and displayed.

*Quiz:* What is the difference between `getchar()`, `getch()`, and `getche()`? Write a program and analyze the differences.

#### 7.2.4 `putch()`

`putch()` will print a character on the screen. It is similar to the `putchar()` function which also prints a character on the screen. Look at Program 7.4 and see how both have the same function.

**PROGRAM 7.4**

```
1. #include<stdio.h>
2. #include<conio.h>
3. void main()
4. {
5. char ch = 'A';
6. putchar(ch);
7. putch(ch);
8. }
```

*Output:*  
AA

---

**7.3 Formatted Functions**

Formatted functions allow us to supply the input in a fixed format and obtain the output in a specified form. There are two formatted functions available in C: `printf()` and `scanf()`. In this section, we will discuss different formatting styles supported by these two functions. We will use several examples to explain them in detail.

**7.3.1 `printf()` Function**

As already discussed, the `printf()` function is one of the most used functions in C. The syntax of this function is:

```
printf("Format strings", list of variables);
```

Program 7.5 will help you to learn more about the `printf()` function. This program adds two numbers and shows the result.

**PROGRAM 7.5**

```
1. #include <stdio.h>
2. int main()
3. {
4. int a, b, c;
5. a = 5;
6. b = 7;
7. c = a + b;
8. printf("%d + %d = %d\n", a, b, c);
9. return 0;
10. }
```

*Output:*

5+7=12

*Explanation:*

- Line 4, `int a, b, c,` declares three integer variables named `a`, `b`, and `c`. The next line 5 initializes the variable named `a` to the value 5.
- Line 6 sets `b` to 7.
- Line 7 adds `a` and `b` and “assigns” the result to `c`.
- The `printf` statement then prints the line “5 + 7 = 12”. The `%d` placeholders in the `printf` statement act as placeholders for values. There are three `%d` placeholders, and at the end of the `printf` line there are three variable names: `a`, `b`, and `c`. `Printf` matches up the first `%d` with `a` and substitutes 5. It matches the second `%d` with `b` and substitutes 7. It matches the third `%d` with `c` and substitutes 12. Then it prints the completed line to the screen: 5 + 7 = 12. The `+`, the `=`, and the spacing are a part of the format line and are embedded automatically between the `%d` operators as specified by the programmer.

### 7.3.2 Formatting with `printf()`

The `printf()` function is known as a formatted function. So in this section we are going to discuss some of the formatting principles of the `printf()` function. Formatting can be applied to any data values and helps us to produce more readable output.

For *integer formatting*, we need to specify the syntax as follows:

`%wd`

Here,

- `w` indicates the width of the field for output. However, if a number is greater than the specified field width, it will be printed in full.
- `d` specifies that the value to be printed is an integer.

For example, if you want to display an integer using a minimum of 8 spaces, you’d write `%8d` in your `printf` statement.

Program 7.6 demonstrates this:

#### PROGRAM 7.6

```

1. #include<stdio.h>
2. int main()
3. {
4. int x = 123;
5. printf("Output- 1 123 displays %0d\n", x);
6. printf("Output- 2 123 displays %1d\n", x);
7. printf("Output- 3 123 displays %2d\n", x);
8. printf("Output- 4 123 displays %3d\n", x);

```

```

9. printf("Output- 5 123 displays %4d\n", x);
10. printf("Output- 6 123 displays %5d\n", x);
11. printf("Output- 7 123 displays %6d\n", x);
12. printf("Output- 8 123 displays %7d\n", x);
13. printf("Output- 9 123 displays %8d\n", x);
14. printf("Output- 10 123 displays %9d\n", x);
15. return 0;
16. }

```

*Output:*

```

Output- 1 123 displays 123
Output- 2 123 displays 123
Output- 3 123 displays 123
Output- 4 123 displays 123
Output- 5 123 displays 123
Output- 6 123 displays 123
Output- 7 123 displays 123
Output- 8 123 displays 123
Output- 9 123 displays 123
Output-10 123 displays 123

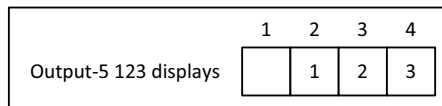
```

Notice that in the first four cases, 123 is displayed in the same way as when you normally use `%d`. Why? Simple: the number of spaces on the screen that 123 can be displayed in is greater than or equal to 3. In output 5 we are using `%4d`, so the compiler will create 4 spaces and display the number by making a right alignment (see Figure 7.4).

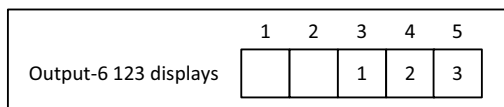
In output 6 we are using `%5d`, so the compiler will create 5 spaces and display the number by making a right alignment (see Figure 7.5).

In the same way as for all the other output the compiler will create the space accordingly and internally, as shown in Figure 7.6.

*Note:* If you write `%09d`, the program will display zeros before the number itself. In the above example, printing 123 using `%09d` displays 000000123 (see Figure 7.7).



**FIGURE 7.4**  
Analysis of output 5.



**FIGURE 7.5**  
Analysis of output 6.

|            |              | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|------------|--------------|---|---|---|---|---|---|---|---|---|
| Output -1  | 123 displays | 1 | 2 | 3 |   |   |   |   |   |   |
| Output -2  | 123 displays | 1 | 2 | 3 |   |   |   |   |   |   |
| Output -3  | 123 displays | 1 | 2 | 3 |   |   |   |   |   |   |
| Output -4  | 123 displays | 1 | 2 | 3 |   |   |   |   |   |   |
| Output -5  | 123 displays |   | 1 | 2 | 3 |   |   |   |   |   |
| Output -6  | 123 displays |   |   | 1 | 2 | 3 |   |   |   |   |
| Output -7  | 123 displays |   |   |   | 1 | 2 | 3 |   |   |   |
| Output -8  | 123 displays |   |   |   |   | 1 | 2 | 3 |   |   |
| Output -9  | 123 displays |   |   |   |   |   | 1 | 2 | 3 |   |
| Output -10 | 123 displays |   |   |   |   |   |   | 1 | 2 | 3 |

**FIGURE 7.6**

Analysis of the output produced by Program 7.6.

|              | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|--------------|---|---|---|---|---|---|---|---|---|
| 123 displays | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3 |

**FIGURE 7.7**

Analysis of %09d.

As discussed above all the alignments are right by default. To make it left align we have to use a *-(minus)* sign. Program 7.7 demonstrates this.

### PROGRAM 7.7

```

1. #include<stdio.h>
2. int main()
3. {
4. int x = 12;
5. int z = 12345;
6. printf("Output- 1 12 \t\t displays %9d\n", x);
7. printf("Output- 2 12 \t\t displays %09d\n", x);
8. printf("Output- 3 12 \t\t displays % -9d\n", x);
9. printf("Output- 4 12 \t\t displays %-09d\n", x);
10. printf("Output- 5 12345 \t displays %9d\n", z);
11. printf("Output- 6 12345 \t displays %09d\n", z);
12. printf("Output- 7 12345 \t displays % -9d\n", z);
13. printf("Output- 8 12345 \t displays %-09d\n", z);
14. return 0;
15. }
```



Output:

```
Output- 1 12 displays 12
Output- 2 12 displays 00000012
Output- 3 12 displays 12
Output- 4 12 displays 12
Output- 5 12345 displays 12345
Output- 6 12345 displays 000012345
Output- 7 12345 displays 12345
Output- 8 12345 _ displays 12345
```

For the *real number formatting*, we need to specify the syntax as follows:

```
%w.vf
```

Here,

w indicates the minimum number of positions that are to be used for the display of the values.

v indicates the number of digits to be displayed after the decimal point.

f specifies that the value to be printed is a real number.

Now, let us take an example to show how will we use it (see Program 7.8).

### PROGRAM 7.8

```
1. #include<stdio.h>
2. int main()
3. {
4. float x = 3.141592;
5. printf("Printing-1 3.141592 \t displays %f\n", x);
6. printf("Printing-2 3.141592 \t displays %1.1f\n", x);
7. printf("Printing-3 3.141592 \t displays %1.2f\n", x);
8. printf("Printing-4 3.141592 \t displays %3.3f\n", x);
9. printf("Printing-5 3.141592 \t displays %4.4f\n", x);
10. printf("Printing-6 3.141592 \t displays %4.5f\n", x);
11. printf("Printing-7 3.141592 \t displays %09.3f\n", x);
12. printf("Printing-8 3.141592 \t displays %-09.3f\n", x);
13. printf("Printing-9 3.141592 \t displays %9.3f\n", x);
14. printf("Printing-10 3.141592 \t displays %-9.3f\n", x);
15. return 0;
 }
```

Output:

```
Printing-1 3.141592 displays 3.141592
Printing-2 3.141592 displays 3.1
Printing-3 3.141592 displays 3.14
Printing-4 3.141592 displays 3.142
Printing-5 3.141592 displays 3.1416
Printing-6 3.141592 displays 3.14159
Printing-7 3.141592 displays 00003.142
Printing-8 3.141592 displays 3.142
Printing-9 3.141592 displays 3.142
Printing-10 3.141592 displays 3.142
```

Let's apply this type of formatting *on the string*. Programs 7.9 and 7.10 shows two examples.

#### PROGRAM 7.9

```
1. #include<stdio.h>
2. int main()
3. {
4. printf("%4.5s\n", "C Programming Learn to Code");
5. printf("%10.7s\n", "C Programming Learn to Code");
6. printf("%11.7s\n", "C Programming Learn to Code");
7. printf("%12.7s\n", "C Programming Learn to Code");
8. printf("%13.8s\n", "C Programming Learn to Code");
9.
10. return 0;
11. }
```

Output:

```
C Pro
 C Progr
 C Progr
 C Progr
 C Progra
```

**PROGRAM 7.10**

```

1. #include<stdio.h>
2. int main()
3. {
4. printf("%31s\n","C Programming Learn to Code");
5. printf("%30s\n","C Programming Learn to Code");
6. printf("%29s\n","C Programming Learn to Code");
7. printf("%28s\n","C Programming Learn to Code");
8. printf("%27s\n","C Programming Learn to Code");
9. return 0;
10. }

```

*Output:*

```

 C Programming Learn to Code
 C Programming Learn to Code
 C Programming Learn to Code
 C Programming Learn to Code
 C Programming Learn to Code

```

The output of the above programs is self-explanatory. Students are encouraged to spend more time on coding this type of program and analyze what happened to the output.

**7.3.3 scanf () Function**

The `scanf()` function allows you to accept input from a standard input device, which for us is generally the keyboard. The simplest form of `scanf()` looks like the following:

```
scanf("format string", &VariableName);
```

- `scanf` takes at least two arguments.
- The first one is a string that consists of format specifiers.
- The rest of the arguments should be variable names preceded with the address of the operator (&).

**7.3.4 Formatting with scanf**

When we read an integer, the specification will be:

```
%wd
```

Let us see what happens when we use this in `scanf()`. Program 7.11 shows this.

In Program 7.12 we will extend this concept and see what happened to the value that was discarded in Run-2.

**PROGRAM 7.11**

```
1. #include<stdio.h>
2. int main()
3. {
4. int x,y;
5. printf("Enter x and y:");
6. scanf("%4d%5d",&x,&y);
7. printf("x=%d,y=%d",x,y);
8. return 0;
9. }
```

*Output:*

Run-1:

Enter x and y: 5434 11232  
x = 5434, y = 11232

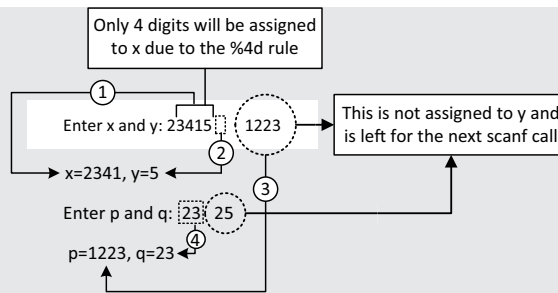
Run-2:

Enter x and y: 23415 1223  
x = 2341, y = 5

- In the program, we take %4d and %5d and in Run-1 we provide a 4-digit number and a 5-digit number. So, the value is assigned to x and y, and displayed.
- But in Run-2 we provide a 5-digit number and a 4-digit number so from the number 23415 (5-digit number) only 2341 (because of %4d) is assigned to x and the last number (i.e., 5) is assigned to y. The value 1223 (4 digit number) is not assigned and left for the next scanf () call.

**PROGRAM 7.12**

```
1. #include<stdio.h>
2. int main()
3. {
4. int x,y,p,q;
5. printf("Enter x and y:");
6. scanf("%4d%5d",&x,&y);
7. printf("x=%d,y=%d",x,y);
8. printf("\nEnter p and q:");
9. scanf("%d%d",&p,&q);
10. printf("p=%d,q=%d",p,q);
11. return 0;
12. }
```



**FIGURE 7.8**  
Execution steps of Program 7.12.

*Output:*

```
Enter x and y: 23415 1223
x = 2341, y = 5
Enter p and q: 23 25
p = 1223, q = 23
```

*Explanation:*

Refer to Figure 7.8 to understand this explanation.

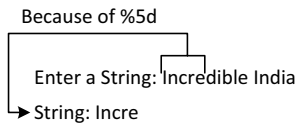
1. In Program 7.12, when the user enters two number 23415 and 1223, `scanf` only reads 4 digits from the first number and assign them to `x`. This is because, `%4d` means you can only read 4 or less than 4-digit numbers.
2. The remaining number, that is 5, will be assigned to `y`.
3. The next number 1223 is not assigned to `y` and will be available if the program has any further `scanf()` functions (it will wait for the next `scanf()` call). As our program contains another `scanf()` call the value 1223 will be assigned to `p`.
4. As the first variable `p` is assigned a value from the previous `scanf()` call, 23 is assigned to `q`; 25 is also not assigned to any variable. This will wait for the next `scanf()` call.

**Quiz:** What will be the output of Program 7.12 if we write `%1.4f` in place of `%4d` and `%2.5f` in place of `%5d`?

We can use the `scanf()` function to read a string too. The format for this can take the following form. Program 7.13 shows an example which uses `%ws`; `%wc` is left for the student. The output of the program is shown below and its explanation is shown in Figure 7.9.

|                  |                  |
|------------------|------------------|
| <code>%ws</code> | <code>%wc</code> |
|------------------|------------------|

Students are encouraged to write C code and observe the output by changing the format specifier and a different value for `w`.



**FIGURE 7.9**  
Analysis of execution of Program 7.13.

### PROGRAM 7.13

```

1. #include<stdio.h>
2. int main()
3. {
4. char str[20];
5. printf("Enter a string:");
6. scanf("%5s",str);
7. printf("String: %s",str);
8. return 0;
9. }
```

#### Output:

```

Enter a string: Incredible India
String: Incre
```

#### Explanation:

A user enters a string "Incredible India", out of which the first five characters "Incre" are assigned to the string.

## 7.4 Review Questions

### 7.4.1 Short Answer Questions

- \_\_\_\_\_ functions take input from the keyboard and produce output through a display device such as a monitor.
- \_\_\_\_\_ function helps us to read and write on a file.
- List the functions that you use to read a character from the input device.
- List the functions that you use to write a character on the output device.
- The `getchar` functions \_\_\_\_\_ a character from the standard input device and the `putchar` function \_\_\_\_\_ a character to the standard output device.
- Write down the syntax of the `getchar` and `putchar` functions.

7. List the functions that you use to read a string from the standard input unit.
8. List the functions that you use to write a string to the standard output unit.

### 7.4.2 Programming Questions

1. Write a program to show the difference between the `getchar`, `getch`, and `getche` functions.
2. Write a program to read a string from the user using the `gets` function and print it on the screen. Rewrite the same program to read and display a string using the `scanf` function. Is there any difference between the output of these two programs?
3. Analyze the program shown below and write down the output it produces.

```
#include<stdio.h>
int main()
{
 printf("%-10.1s\n", "Program");
 printf("%-10.2s\n", "Program");
 printf("%-10.3s\n", "Program");
 printf("%-10.5s\n", "Program");
 printf("%-10.6s\n", "Program");
 printf("%-10.7s\n", "Program");
 return 0;
}
```

4. Rewrite the program in Question 3 by removing the minus sign and write down the output it produces. Analyze both the outputs and explain the differences.
5. Given three integers, 15, 150, and 1500, write a program that prints the integers on the screen in hexadecimal format.
6. Write a program that uses `getchar()` and `putchar()` to read a character entered by the user and write it to the screen.
7. What is the output of the following programs?

```
a. #include<stdio.h>
void main()
{
 int ch;
 ch='A';
 putchar(ch);
}
```

```
b. #include<stdio.h>
void main()
{
 int ch;
 ch='A';
 ch=ch+1;
 putchar(ch);
}
```

```
c. #include<stdio.h>
void main()
{
 int ch;
 ch='A';
 ch=ch++;
 putchar(ch);
}
```

```
d. #include<stdio.h>
void main()
{
 int ch;
 ch='A';
 putchar(ch++);
}
```

```
e. #include<stdio.h>
void main()
{
 int ch;
 ch='A';
 putchar(++ch);
}
```

```
f. #include<stdio.h>
void main()
{
 int ch;
 ch='A';
 putchar(++ch);
 putchar(ch++);
 putchar(ch);
 putchar(ch--);
 putchar(ch);
}
```

```
g.#include<stdio.h>
void main()
{
 int ch;
 ch='A';
 putchar(ch=97);
}
```

```
h.#include<stdio.h>
void main()
{
 int ch;
 ch='A';
 putchar(ch=97+5);
}
```

```
i.#include<stdio.h>
void main()
{
 int ch;
 ch='A';
 putchar(97+5);
}
```

### 7.4.3 Subjective Questions

1. What is the difference between `getch()` and `getche()`? Explain with an example.
2. What do the `getchar()` and `putchar()` functions do?
3. What is the difference between `putc()` and `putchar()`? Explain with an example.
4. What does the `getchar()` function return?
5. Within `%10.3f`, which part is the minimum field width specifier, and which one is the precision specifier?
6. What is the difference between the `scanf()` and `gets()` functions?
7. Explain the syntax of the `printf()` and `puts()` functions. Which one is more convenient to use, and what is the difference between them?
8. List the functions whose definition is present in the `<conio.h>` header file. Explain their syntax and their uses.
9. Write the syntax of all the console I/O functions with appropriate examples.
10. Every function returns something, such as the `printf()` function which returns the number of characters it prints on the screen. The task is to list out all the functions you have read so far and write what they return.





**Taylor & Francis**

Taylor & Francis Group

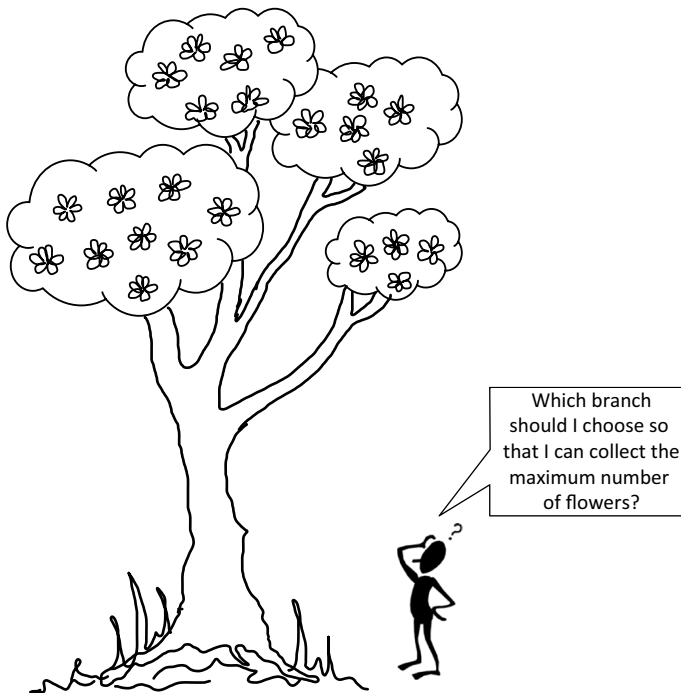
<http://taylorandfrancis.com>

# 8

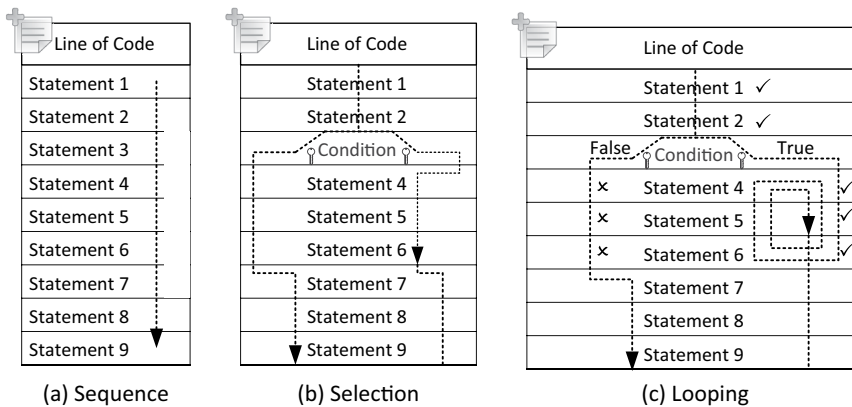
## *Control Structures*

### 8.1 Introduction

Whatever we have discussed till now, all problems have been relatively simple and did not require any decision-making process. But in real life, we come across many problems where we need to decide what to do and what not to do. Assume that we want to collect some flowers from a tree, and the tree has many branches. So, we need to choose a branch in such a manner that we can get the maximum number of flowers (Figure 8.1). Similarly, we may have problems that need some work to be done repeatedly. So, to solve different kinds of problems, we need the concept of control structure. Control structure determines the flow of control in a program, that is, it indicates the order in which the various instructions in a program are executed inside the computer.



**FIGURE 8.1**  
Decision-making problem example.



**FIGURE 8.2**  
Categories of control statements.

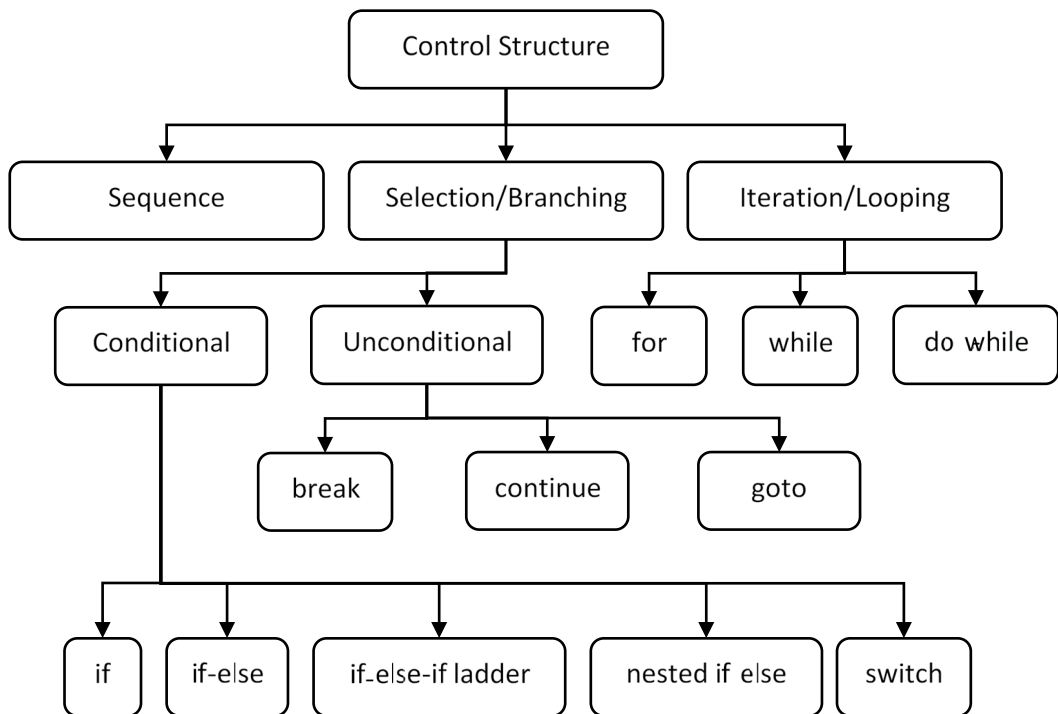
Control statements in C are divided into three categories (Figure 8.2):

- **Sequence Control Structure:** This ensures that the instructions in the program are executed in the same order in which they appear. As you can see in Figure 8.2a, there are nine program statements, and the compiler starts execution from Statement 1 and continues through to Statement 9 without skipping any of the intervening statements.
- **Selection Control Structure:** This is also known as decision control structure, and it ensures the computer makes decisions about which statement is to be executed next. Figure 8.2b shows the flow of execution. After executing Statement 2, the compiler arrives at Statement 3 which is a condition, and if the condition is satisfied (is true) then the compiler executes Statements 4, 5, and 6, otherwise it executes 7, 8, and 9.
- **Loop Control Structure:** This helps the computer to execute group/single statements repeatedly until a condition is satisfied. Figure 8.2c shows the flow of execution. The compiler executes Statements 4 through 6 repeatedly until the condition becomes true, and the moment it is false, it starts its execution from Statement 7 onwards.

All programs discussed so far come under the sequence control structure. In this chapter, we will introduce several programs that will reflect the properties of the selection and loop control structure. After completing this chapter, the student will know the following:

- What different control structures are available in the C language.
- The syntax of different control structures.
- Be able to differentiate between sequence and selection control structure.
- Write code for more realistic problems.
- What a compound statement is.

The C language provides several keywords for control structure declaration. We classify all those keywords according to our control structure category. Figure 8.3 shows a detailed



**FIGURE 8.3**  
Classification of control structure.

classification of all control statements and introduces several new keywords. We divide the selection control statements into two groups, because some selection statements do not require conditions; we can break the flow of execution arbitrarily.

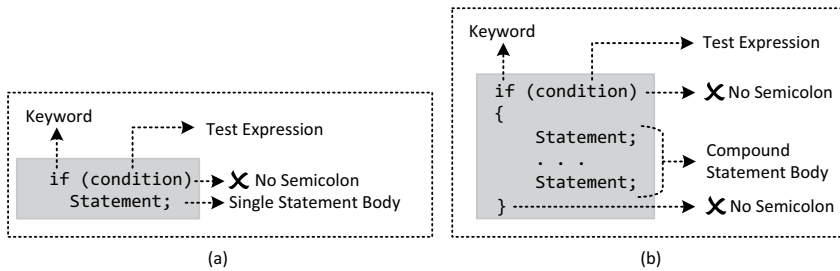
In the next section we will discuss all control statements, their syntaxes, flowcharts, and some example programs that show their properties and execution procedure.

---

## 8.2 Selection with if Statements

Suppose we are executing a set of instructions, and we want some instruction to execute only when a specific condition is satisfied. With the help of an *if* selection control statement, we can achieve this. The syntax of *if*, with and without a compound statement, is shown in Figure 8.4.

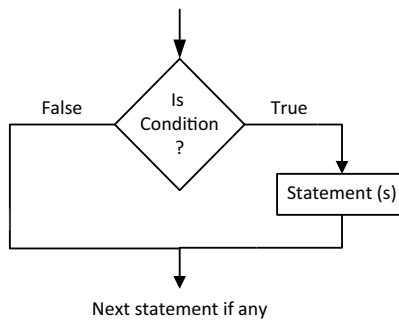
Figure 8.4 shows the syntax where a single statement depends on the *if* condition. Hence, we don't require curly braces to enclose the statement. But when multiple statements depend on a condition to execute them, we need to enclose them within curly braces. Even if you enclose a single statement inside the curly braces, the compiler does not show an error, but, for multiple statements, it is mandatory. In Figure 8.4 multiple statements

**FIGURE 8.4**

Syntax of the if selection control structure (a) without a compound statement (b) with a compound statement.

depend on the condition which is enclosed within the curly braces. When multiple statements are enclosed inside curly braces, we call it a *compound statement*. Figure 8.5 shows the flowchart of an if statement that describes the flow of execution.

The following example shows a program that uses an if statement to find out the biggest among three numbers.

**FIGURE 8.5**

if statement flowchart.

### PROGRAM 8.1

```

1. #include <stdio.h>
2. int main()
3. {
4. int a, b, c, big;
5. printf("\nEnter three numbers:");
6. scanf("%d%d%d", &a, &b, &c);
7. big=a;
8. if(b > big)
9. big=b;

```

```

10. if(c > big)
11. big=c;
12. printf("\nThe greater number = %d", big);
13. return 0;
14. }

```

*Output:*

```

Enter three numbers: 25 67 38
The greater number = 67

```

*Explanation:*

- Line 4 declares four variables and allocates memory space for them.
- Lines 5–6 prompt the user to input three numbers and store them in memory locations designated a, b, c.
- Line 7 assigns the value of a to big, assuming that initially a is big.
- Lines 8–9 check whether b is bigger than big or not. If it is, then it assigns the value of b to big, otherwise big will have its previous value intact.
- Similarly, lines 9–10 do the same comparison with c.
- Finally, line 11 displays the biggest number among a, b, and c.

### 8.2.1 Some Points to Remember

The reader is encouraged to remember the following points. These are common mistakes made by a beginner. These points will help you understand the concept in greater detail too.

1. Putting a semicolon at the end of an *if (condition)* statement does not show an error. But the statement following the *if* is executed directly even if the condition become false:

#### PROGRAM 8.2

```

1. #include <stdio.h>
2. int main()
3. {
4. int x=5, y=6;
5. if(x > y);
6. {
7. x=x+1;
8. y=y+1;
9. }
10. printf("\n x=%d", x);
11. printf("\n y=%d", y);
12. return 0;
13. }
14.

```

Due to this semicolon, line 7 and 8 gets executed like normal sequential execution.

*Output:*

```
x=6
y=7
```

*Quiz:* What is the output of Program 8.2 if we remove the semicolon after the *if* statement?

2. If no curly braces are used in a compound statement then the statement immediately following the *if* statement depends on the condition of the *if* statement. Any other statement after the first statement does not depend on the *if* condition.

### PROGRAM 8.3

```
1. #include <stdio.h>
2. int main()
3. {
4. int x=5, y=6;
5. if(x > y)
6. x=x+1;
7. y=y+1;
8. printf("\n x=%d", x);
9. printf("\n y=%d", y);
10. return 0;
11. }
```

Only line 6 depends on the condition given in line 5. Line 7 executes sequentially as if it is not a part of the *if* condition.

*Output:*

```
x = 5
y = 7
```

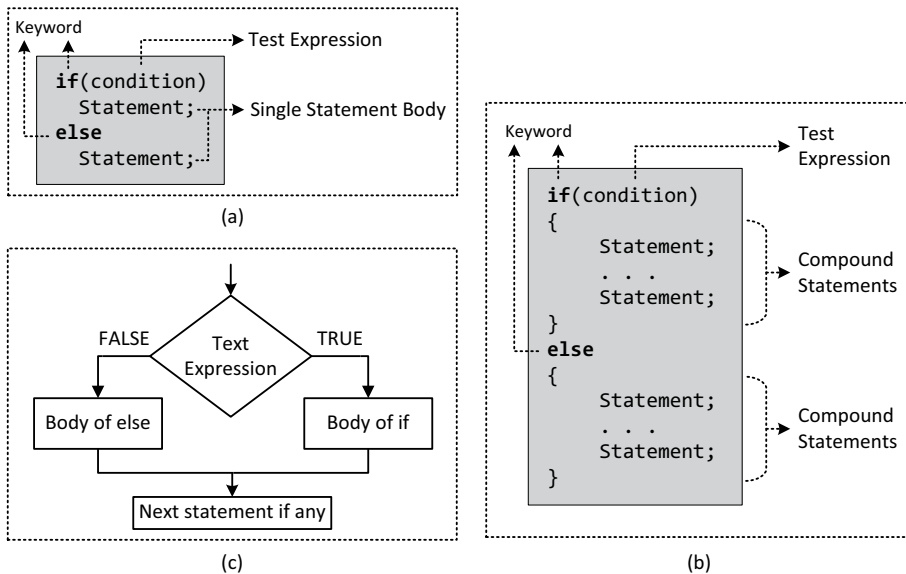
## 8.3 if-else Statement

If you analyze the concept of *if*-statements discussed in Section 8.2, you can quickly notice that the *if* condition only shows what happens if the condition is true, but it does not show what happens if the condition is false. The *if-else* construct will show what happens both ways.

We use an if-else statement to break the sequential flow of the execution of program statements. We require a condition to do that. Some instructions depend on this condition and others not. During the flow of execution, when the compiler encounters this condition, it will decide which set of instructions will be executed next. Hence, there will be two paths: one follows the truthfulness of the condition, and the other follows its falsity. The remaining statement(s) will execute sequentially. The syntax of *if-else*, with and without compound statements and flow of execution is shown in Figure 8.6.

In Figure 8.6, if the condition is satisfied, then the statement after the `if` is executed, else the statement following the `else` is executed. Similarly, if the condition is satisfied, the statements present inside the `if` block are executed, else the statement inside the `else` block is executed. The figure shows the flow of execution. The flowchart shows that if the test expression is evaluated to true, then all the statements present inside the body of the `if` statement are executed, and if it (the test expression) is false, then the statement contained in the body of the `else` part is executed.

Let us take some examples to show the flexibility of using an if-else control structure, and how they help us in solving real-world problems. We will discuss two fundamental programs. The first program is to check whether a number entered by the user is zero or nonzero. The second program is to calculate the travel fare of a person. The person needs to enter how many kilometers he or she has traveled.



**FIGURE 8.6** (a) Syntax of if-else without compound statements; (b) Syntax of if-else with compound statements; (c) Flow chart of if-else that shows the execution flow.



### 8.3.1 Write a Program to Check Whether a Number Entered by the User is Zero or Nonzero

#### PROGRAM 8.4

```
1. #include <stdio.h>
2. int main()
3. {
4. int input;
5. printf("Enter an integer:");
6. scanf("%d", &input);
7. if(input)
8. printf("\nIt is Non-Zero");
9. else
10. printf("\nIt is Zero");
11. return 0;
12. }
```

*Output:*

*Run-1*

Enter an integer:0  
It is Zero

*Run-2*

Enter an integer:65  
It is Non-Zero

*Explanation:*

As we know the compiler will treat all nonzero values as true, and zero values as false, so here the condition written in line 7 is as follows:

Line 7: `if (input)`

indicates that if the input is nonzero the compiler takes it as true and prints:

*It is Non-Zero*

But if the input is zero then the compiler takes it as false and hence the *else* statement is executed which prints the output:

*It is Zero*

### 8.3.2 Write a Program to Calculate the Travel Fare of a Person

Write a program to calculate the travel fare of a person. If the person travels more than 20 km then he or she needs to pay Rs 7/km plus a minimum fare of Rs 100. If the person travels less than or equal to 20 km, then he or she needs to pay Rs 10/km plus a minimum fare of Rs 50. Find the total fare of the person if the traveled kilometers are entered into your program.

To write a program for the above problem, we need to check one condition: whether the person traveled more than 20 km or not. If true, then the total fare will be calculated as follows:

$$\text{Minimum Fare} = 100$$

$$\text{Fare} = \text{Traveled km} \times 7$$

$$\text{Total Fare} = \text{Fare} + \text{Minimum Fare}$$

Otherwise, the total fare is calculated as follows:

$$\text{Minimum Fare} = 50$$

$$\text{Fare} = \text{Traveled km} \times 10$$

$$\text{Total Fare} = \text{Fare} + \text{Minimum Fare}$$

#### PROGRAM 8.5

```
1. #include <stdio.h>
2. int main()
3. {
4. int Tkm;
5. float Fare, mFare, tFare;
6. printf("\n How many kilometers have you traveled: ");
7. scanf("%d", &Tkm);
8. if (Tkm>20)
9. {
10. mFare=100;
11. Fare=Tkm*7;
12. tFare=Fare+mFare;
13. }
14. else
15. {
16. mFare=50;
17. Fare=Tkm*10;
```

```

18. tFare=Fare+mFare;
19. }
20. printf("Your total fare will be: %f", tFare);
21. return 0;
22. }

```

*Output:*

*Run-1*

How many kilometers have you traveled: 17  
Your total fare will be: 220.000000

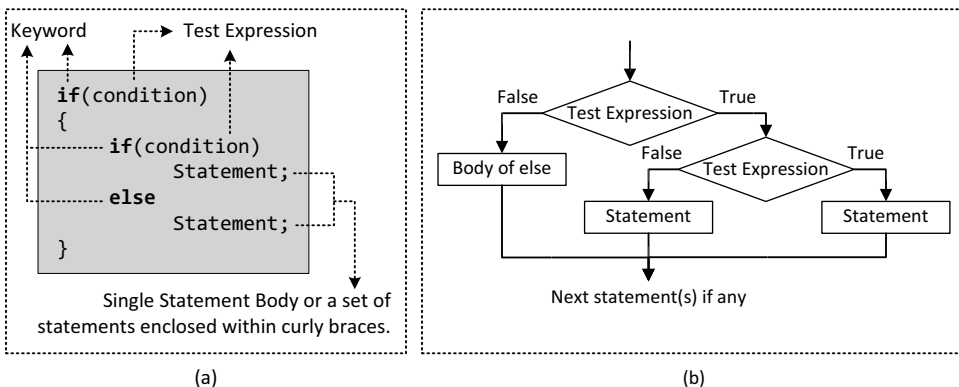
*Run-2*

How many kilometers have you traveled: 62  
Your total fare will be: 534.000000

## 8.4 Nested if-else Statements

It is possible to write an entire if-else construct within another if statement or an else statement. This is called nesting of if-else, that is, the entire if-else construct is contained within another if statement or else statement. The syntax of the nested if-else statement and its flowchart are shown in Figure 8.7.

Let us write a program to show how to use nested if-else statements in solving problems. The first program we discuss here is to find the biggest among three numbers. There are several ways to solve this problem, but we will try to write the program in such a manner that it will reflect the syntax of the nested if-else.



**FIGURE 8.7**

(a) Syntax of the nested if-else statement (b) Nested if-else flowchart.

### 8.4.1 Write a Program to Find the Biggest Among Three Numbers

#### PROGRAM 8.6

```

1. #include <stdio.h>
2. int main()
3. {
4. int A, B, C;
5. printf("\nEnter three numbers: ");
6. scanf("%d%d%d", &A, &B, &C);
7. if(A > B)
8. {
9. if(A > C)
10. printf("A= %d is greater", A);
11. else
12. printf("C= %d is greater", C);
13. }
14. else
15. {
16. if(B > C)
17. printf("B= %d is greater", B);
18. else
19. printf("C= %d is greater", C);
20. }
21. return 0;
22. }
```

*Output:*

*Run-1*

Enter three numbers: 23 56 78  
C= 78 is greater

*Run-2*

Enter three numbers: 25 56 45  
B= 56 is greater

## 8.5 if-else-if Ladders

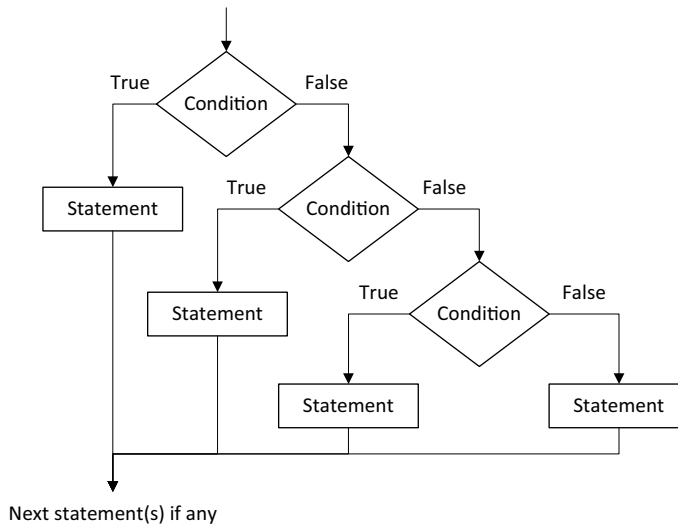
This is another form of if-else, where each else statement is associated with an immediate if statement.

### Syntax of an if-else-if ladder

```

if (condition 1)
statement 1;
else if (condition 2)
statement 2;
else if (condition 3)
statement 3;
else if (condition n)
statement n;
else
default statement;

statement x;
```



**FIGURE 8.8**  
Flowchart of the if-else-if ladder.

The conditions are evaluated from the top (of the ladder), downwards. As soon as a true condition is found, the statement associated with it is executed and control is transferred to statement  $x$  (skipping the rest of the ladder). The flowchart of the if-else-if ladder is given in Figure 8.8.

There are many problems that need concepts like the if-else-if ladder. One such problem is discussed in Section 8.5.1.

### 8.5.1 Write a Program to Perform as a Four-Function Calculator

The objective of this program is to perform four arithmetic operations: addition, subtraction, multiplication, and division. The user needs to enter two numbers and an operation symbol. For example, suppose the user enters  $25 + 30$ , then our program must add the two values and display the result as 55. Our program must read the operator and analyze what to do with the operand.

#### PROGRAM 8.7

```

1. #include <stdio.h>
2. int main()
3. {
4. char op;
5. float number1, number2, result;
6. printf("Enter two numbers and an operator in the following
format\n");

```

```
7. printf(" number1 operator number2\n");
8. scanf("%f %c %f", &number1, &op, &number2);
9. if(op == '*')
10. result = number1 * number2;
11. else if(op== '/')
12. result = number1 / number2;
13. else if(op=='+')
14. result = number1 + number2;
15. else if(op=='-')
16. result = number1 - number2;
17. else
18. printf("Please enter a correct
 operator");
19. printf("%f %c %f = %f\n", number1, op, number2, result);
20. return 0;
21. }
```

*Output:*

Run-1

Enter two numbers and an operator in the following format

number1 operator number2

3.6 + 7.8

3.600000 + 7.800000 = 11.400000

Run-2

Enter two numbers and an operator in the following format

number1 operator number2

25 \* 35

25.000000 \* 35.000000 = 875.000000

Run-3

Enter two numbers and an operator in the following format

number1 operator number2

45 - 5.7

45.000000 - 5.700000 = 39.299999

*Explanation:*

We declare three float variables, number1, number2, and result (see line 5). To read the operator, we take a character variable op (line 4). Lines 6–8 read the operand and operator and store them in appropriate variables. Lines 9–18 represent the if-else-if ladder that performs the actual operation. For any input one condition will be satisfied. If the user enters any wrong input, then line 18 will execute and prompt the user to correct his or her input. Finally, line 20 displays the result in a readable format.

---

## 8.6 Compound Statements

A compound statement is a set of statements enclosed within a pair of curly braces, “{” and “}”. It (also called a “block”) typically appears as the body of another statement, such as an if statement. Compound statements are not a simple sequence of executable statements, but can also contain variable declarations at the beginning.

Program 8.8 shows the output that describes how the compiler treats a compound statement.

### PROGRAM 8.8

```
1. #include <stdio.h>
2. int main()
3. {
4. int i=10;
5. {
6. int i=20;
7. printf("\nInside compound statement i = %d", i);
8. }
9. printf("\nOutside compound statement i = %d", i);
10. return 0;
11. }
```

*Output:*

```
Inside compound statement i = 20
Outside compound statement i = 10
```

*Explanation:*

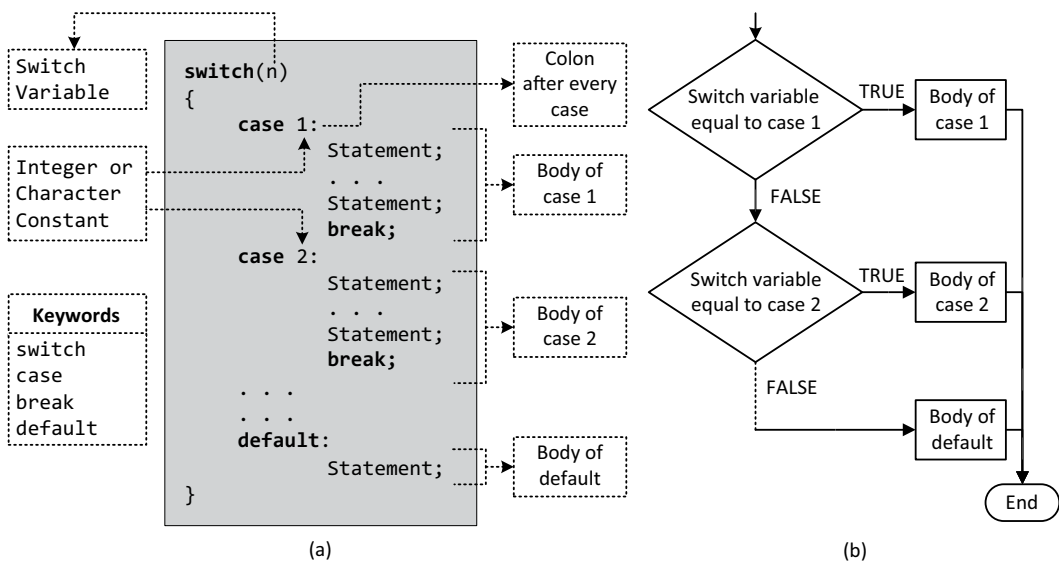
- The value of `i` inside and outside of the compound statement does not coincide.
- The value of `i` inside the compound statement is local to that compound statement block, so the output is 20. But the `i` that declares outside the compound statement is local to the function `main()`, so the output is 10.
- Generally, compound statements are used in control statements where a group of statements is executed according to a particular condition. The group of statements is enclosed within two curly braces “{” and “}”.

## 8.7 Multiway Selection with Switch Statements

This is a selection control structure that helps in selecting some lines to execute from a number of choices given. It causes a particular statement (or a group of statements) to be chosen from several available groups.

- Figure 8.9 shows the syntax of the **switch** case and its flowchart. First, the expression (n) following the keyword **switch** is evaluated.
- The value given by the expression (n) is then matched one by one against the constant value that follows the **case** statement.
- When a match is found, the program executes the statement following that **case** (body of the case).
- When a **break** statement is encountered in that **case**, flow exits the **switch** statement.
- If no match is found with any of the case statements, the statement following the default (body of the default) is executed.

In the following, we show an example program to explain the concept of a switch case statement. This program will read the numbers from 1 to 7 and display the corresponding day name, where 1 corresponds to Sunday, 2 corresponds to Monday, and so on.



**FIGURE 8.9**  
(a) Switch statement syntax; (b) Flowchart of switch statement.



**PROGRAM 8.9**

```
1. #include <stdio.h>
2. int main()
3. {
4. int x;
5. printf("Enter an integer between 1 and 7:");
6. scanf("%d",&x);
7. switch(x)
8. {
9. case 1:
10. printf("\n Sunday");
11. break;
12. case 2:
13. printf("\nMonday");
14. break;
15. case 3:
16. printf("\nTuesday");
17. break;
18. case 4:
19. printf("\nWednesday");
20. break;
21. case 5:
22. printf("\nThursday");
23. break;
24. case 6:
25. printf("\nFriday");
26. break;
27. case 7:
28. printf("\nSaturday");
29. break;
30. default:
31. printf("\nMatch not found");
32. }
33. return 0;
34. }
```

*Output:*

```
Run-1
Enter an integer between 1 and 7:3
Tuesday

Run-2
Enter an integer between 1 and 7:9
Match not found
```

### 8.7.1 Some Points to Remember

1. A maximum 257 cases are possible within a **switch** statement.
2. The **cases** inside a **switch** statement can be placed in any sequence.

#### PROGRAM 8.10

```
1. #include<stdio.h>
2. void main()
3. {
4. int x=2;
5. switch(x)
6. {
7. case 3:
8. printf("\nI am in case 3");
9. break;
10. case 1:
11. printf("\nI am in case 1");
12. break;
13. case 2:
14. printf("\nI am in case 2");
15. break;
16. default:
17. printf("\nI am in default");
18. }
19. }
```

*Output:*

```
I am in case 2
```

3. The **default** case is not compulsory. It can be placed anywhere inside the **switch** statement.

#### PROGRAM 8.11

```
1. #include<stdio.h>
2. void main()
3. {
4. int x=2;
5. switch(x)
6. {
7. case 3:
8. printf("\nI am in case 3");
9. break;
10. default:
11. printf("\nI am in default");
12. case 1:
13. printf("\nI am in case 1");
14. break;
15. case 2:
16. printf("\nI am in case 2");
17. break;
18. }
19. }
```

*Output:*

I am in case 2

4. If no **break** statement is there then all the statements inside the **switch** will be executed from the point where the case satisfies.

#### PROGRAM 8.12

```
1. #include<stdio.h>
2. void main()
3. {
4. int x=2;
5. switch(x)
6. {
7. case 1:
8. printf("\nI am in case 1");
9. case 2:
10. printf("\nI am in case 2");
11. default:
12. printf("\nI am in default");
13. }
14. }
```

Output:

```
I am in case 2
I am in default
```

---

## 8.8 goto Statement

The goto statement is used to alter the normal sequence of program execution by unconditionally transferring control to some other part of the program.



- The label is an identifier used to label the target statement to which control will be transferred.
- The target statement must be labeled, and it must be followed by a colon (:).

### 8.8.1 Notes on goto

- Avoid **goto** statements! They make a C programmer's life miserable.
- Their use is one of the reasons why programs become unreliable, unreadable, and hard to debug.
- **goto** statements take control wherever you want; however almost always there is a more elegant way of writing the same program using **if**, **for**, **while**, and **switch**.

Consider the following program that prints the numbers from 1 to 10 using goto.

#### PROGRAM 8.13

```
1. #include<stdio.h>
2. void main()
3. {
4. int i=0;
5. loop:
6. i=i+1;
7. if(i<=10)
8. {
9. printf(" %d",i);
10. goto loop;
11. }
12. }
```

*Output:*

1 2 3 4 5 6 7 8 9 10

*Explanation:*

- In the above program, *i* is initialized at 0. The increment portion of the *i* value is stored inside the label;
- Inside the *if* statement the value of *i* is printed and the *goto* statement takes control to the label where *i* is incremented until the condition inside the *if* is false.

Let us take another example. The following program finds the factorial of a number.

#### PROGRAM 8.14

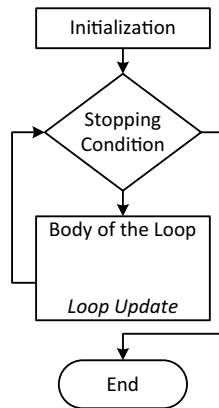
```
1. #include<stdio.h>
2. void main()
3. {
4. int x=0, N, fact=1;
5. printf("Enter a number: ");
6. scanf("%d", &N);
7. inr:
8. x=x+1;
9. if(x<=N)
10. {
11. fact=fact*x;
12. goto inr;
13. }
14. printf("Factorial= %d", fact);
15. }
```

*Output:*

Enter a number: 5  
Factorial= 120

## 8.9 Introduction to Loops

A loop is nothing but the execution of a statement or a series of statements repeatedly until the work is done. To make sure that it ends, we must have a condition that controls the loop. The loop control structure is generally divided into three sections and takes the form as shown in Figure 8.10.



**FIGURE 8.10**  
Sections of a loop.

1. **Loop initialization:** Initialization must be done before the first execution of the loop body. The initialization statement of a loop is always executed once.
2. **Loop update:** The stopping condition of a loop can only be achieved by loop updating. A loop update is a statement inside the loop body that updates in each iteration and changes the stopping condition from true to false.
3. **Stopping condition:** This is a condition that controls the loop for execution. The whole body of the loop will be executed until the stopping condition becomes false.

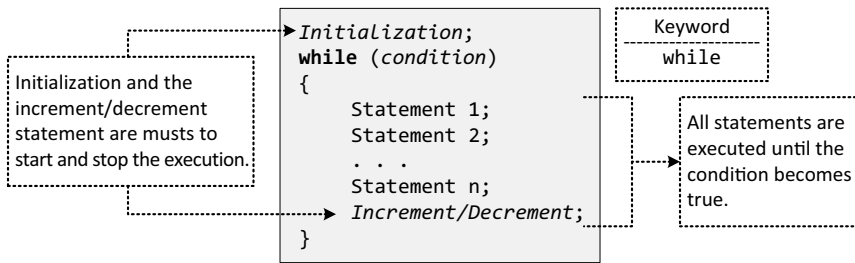
## 8.10 while Loops

A *while loop* is a control flow statement that allows code to be executed repeatedly based on a given condition.

- The while construct consists of a block of code and a condition.
- The condition is first evaluated – if the condition is true the code within the block is then executed.
- This repeats until the condition becomes false (while loops check the condition before the block is executed).
- The while loop is often called a *pre-test loop* or *entry-controlled loop*.

The syntax of the while loop is shown in Figure 8.11.

Suppose you want to print your name ten times, then one way to solve this problem is to use 10 `printf()` statements which will print your name. But this type of problem can be solved easily by a while loop.



**FIGURE 8.11**  
while loop syntax.

### PROGRAM 8.15

```

1. #include<stdio.h>
2. void main()
3. {
4. int i=0;
5. while (i < 10)
6. {
7. printf("\nC Programming Learn to Code");
8. i = i + 1;
9. }
10. }

```

*Output:*

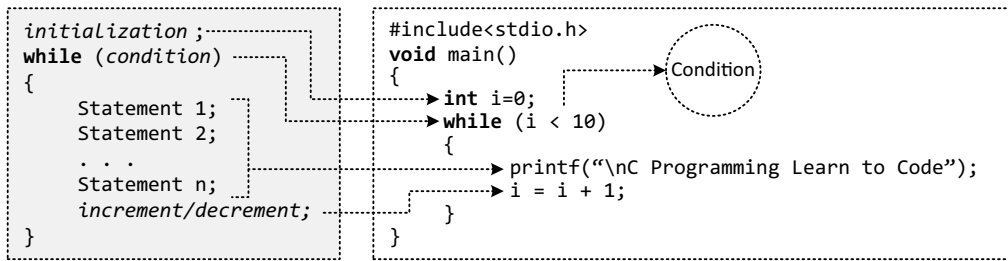
```

C Programming Learn to Code
C Programming Learn to Code
C Programming Learn to Code
C Programming Learn to Code
C Programming Learn to Code
C Programming Learn to Code
C Programming Learn to Code
C Programming Learn to Code
C Programming Learn to Code
C Programming Learn to Code
C Programming Learn to Code

```

*Explanation:*

- Line 4 (i.e., `int i=0`) will initialize the value of `i` to 0.
- Line 5 is our while loop which includes the condition `i<10` that indicates the statement in between the curly braces (lines 6 and 9) will execute until `i<10` becomes false.
- To make the condition false the `i` value should be incremented and this is done in line 8 (`i=i+1`). If this line is not included, then the loop will execute endlessly, which is an infinite loop.
- Line 7 is our statement which is to be executed. As this statement is executed ten times so "C Programming Learn to Code" will be printed ten times.



**FIGURE 8.12**  
Integration between syntax and program.

Figure 8.12 shows the integration between the syntax and the program.

Let's take another example that explains more about while loops. This program will calculate the simple interest for three sets of  $p$ ,  $n$ , and  $r$ . Here  $p$  stands for the principal amount,  $n$  for the number of years, and  $r$  for the rate of interest.

#### PROGRAM 8.16

```

1. #include<stdio.h>
2. void main()
3. {
4. int p, n, count;
5. float r, si;
6. count=1;
7. while (count<=3)
8. {
9. printf("\nEnter value of p, n and r: ");
10. scanf("%d %d %f", &p, &n, &r);
11. si = p*n*r/100;
12. printf("\nSimple Interest=Rs. %f ", si);
13. count = count + 1;
14. }
15. }
```

*Output:*

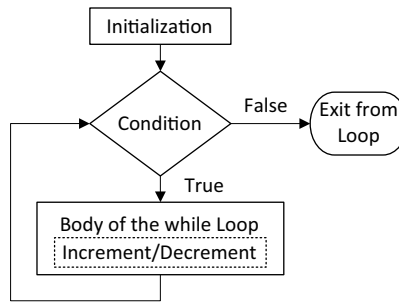
```

Enter value of p, n and r: 5000 3 10
Simple Interest=Rs. 1500.000000
Enter value of p, n and r: 7500 2 15
Simple Interest=Rs. 2250.000000
Enter value of p, n and r:15000 6 4
Simple Interest=Rs. 3600.000000
```

*Explanation:*

The body of the while loop will be executed three times for count 1, 2 and 3, because our condition is  $\text{count} \leq 3$ . Every time it enters the loop, it asks for the value of  $p$ ,  $n$ ,  $r$ , and then calculates the simple interest and produces the result.





**FIGURE 8.13**  
Flowchart of a while loop.

Figure 8.13 shows the flowchart of the while loop. The increment/decrement statement is a part of the loop body. The body executes only when the condition is satisfied, otherwise control goes out of the loop.

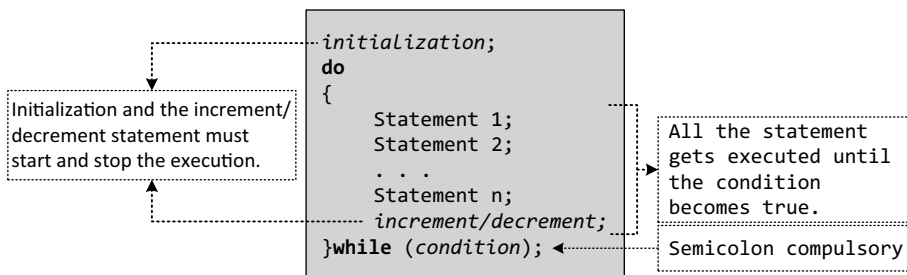
### 8.11 do-while Loops

As with the while loop a *do-while loop* is also a control flow statement that allows code to be executed repeatedly based on a given condition.

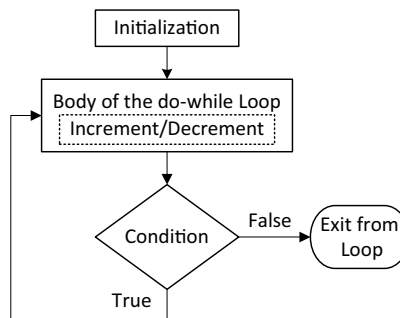
- The *do-while* construct consists of a block of code and a condition. First, the code within the block is executed, and then the condition is evaluated. If the condition is true, the code within the block is executed again. This repeats until the condition becomes false.

The syntax of a while loop and do-while loop are similar to each other, except the condition is written after the body of the loop. The syntax of the do-while loop is given in Figure 8.14 and the flowchart is given in Figure 8.15.

Program 8.17 shows the program code that checks whether a number is a palindrome or not using a do-while loop. A palindrome number is a number that remains the same after it is reversed. For example, 151 is a palindrome number because if reversed it is still 151. But, if you consider 256, then the reverse is 652. Hence 256 is not a palindrome number.



**FIGURE 8.14**  
do-while loop syntax.



**FIGURE 8.15**  
Flowchart of a do-while Loop.

### PROGRAM 8.17

```

1. #include<stdio.h>
2. void main()
3. {
4. int N,rev=0,digit,temp;
5. printf("Input the number:");
6. scanf("%d",&N);
7. temp=N;
8. do
9. {
10. digit=N%10;
11. rev=rev*10+digit;
12. N=N/10;
13. }while(N!=0);
14
15. if(temp==rev)
16. printf("PALINDROME NUMBER\n");
17. else
18. printf("NOT PALINDROME\n");
19.
20. }
```

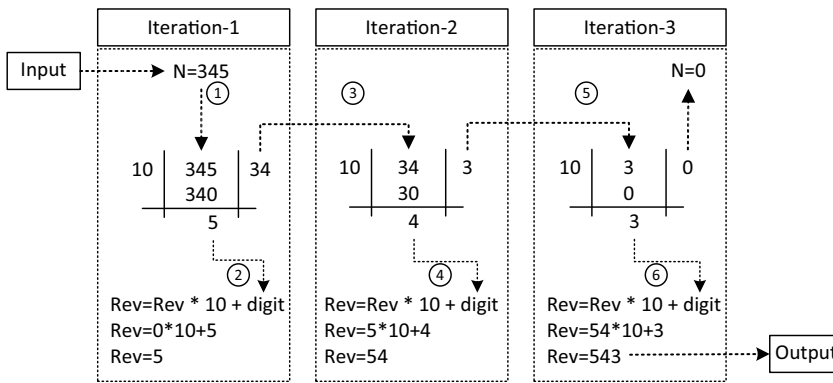
*Output:*

Run-1  
Input the number:12345  
NOT PALINDROME

Run-2  
Input the number:12521  
PALINDROME NUMBER

*Explanation:*

To check whether a number is a palindrome, we must reverse it. Line numbers 8–13 do this work. The next step is easy; we need to compare the reversed number with the original one for similarity, and if it is similar then the number is a palindrome. Lines 15–18 do this work. The tricky part of this program is to find the reverse. To understand this, see Figure 8.16.



**FIGURE 8.16**  
Execution steps (Lines 8–13).

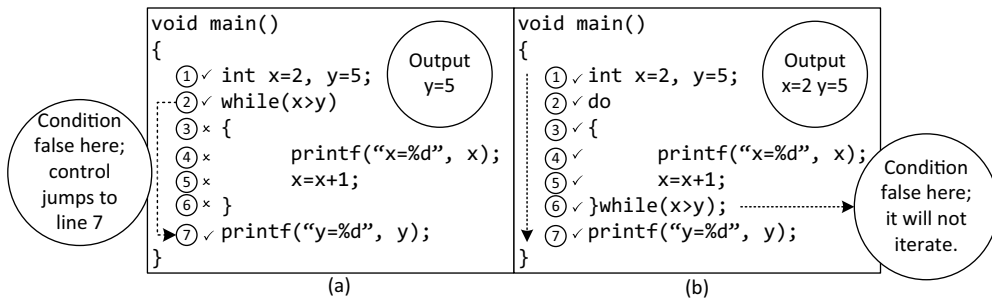
In Figure 8.16,  $N=345$  is the input to our do-while loop, and  $543$  is the output, which is the reverse of  $345$ . At each iteration, the number is divided by  $10$ . After iteration 3,  $N$  becomes  $0$  and the iteration stops. The reverse number is also calculated in each iteration; in iteration 1,  $Rev=5$ , in iteration 2,  $Rev=54$ , and finally, at iteration 3,  $Rev=543$ . You can read the figure from ① through ⑥ to understand its execution procedure. After we get the reverse number in  $Rev$ , we compare it with  $temp$  ( $temp$  contains the value of  $N$ ; see line number 7) and print out whether the number is a palindrome or not. Why do we keep the value of  $N$  in  $temp$ ? Because, after iteration 3,  $N$  becomes  $0$ . But we need the value of  $N$  to compare with  $Rev$ , that's why we keep the value of  $N$  in  $temp$ .

### 8.11.1 Difference Between while and do-while Loops

There is a minor *difference* between the working of *while* and *do-while* loops.

- The *while* loop tests the condition before executing any of the statements within a while loop, but the *do-while* loop tests the condition after executing the statements at least once.
- The *do-while* executes the statements at least once even if the condition is false, but the *while* loop does not execute any statement if the condition is false.
- The *do-while* loop is often called a *post-test loop* or *exit-controlled loop*. Similarly, the *while* loop is called an *entry-controlled loop* or *pre-test loop*.

Figure 8.17 illustrates the concept.



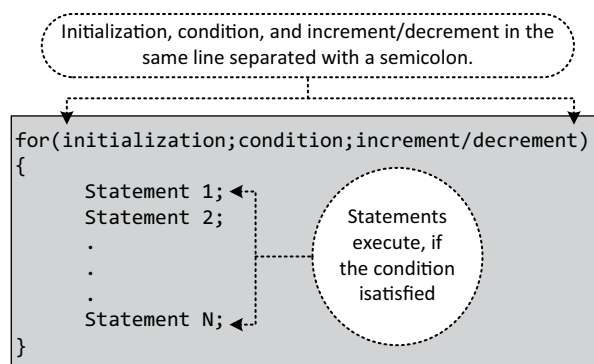
**FIGURE 8.17**  
Difference between while loops and do-while loops.

In Figure 8.17, a sample program is given that shows the execution of a while loop. You can see that, as the condition becomes false at line 2, control directly jumps to line 7. Lines 3 to 6 do not get executed and are marked with a cross symbol (×). Hence, the output in this case will be  $y = 5$ . The same program is also written with a do-while loop in Figure 8.17. We know that the condition of the do-while loop executes after the execution of the body of the loop. In this case also, the condition is false. So, it will not iterate and will instead exit from the loop after executing line 6 and jump to line 7. But before exiting, it will have already executed lines 1 to 5. You can see that all the lines are executed at least once. Hence the output is  $x = 2$   $y = 5$ .

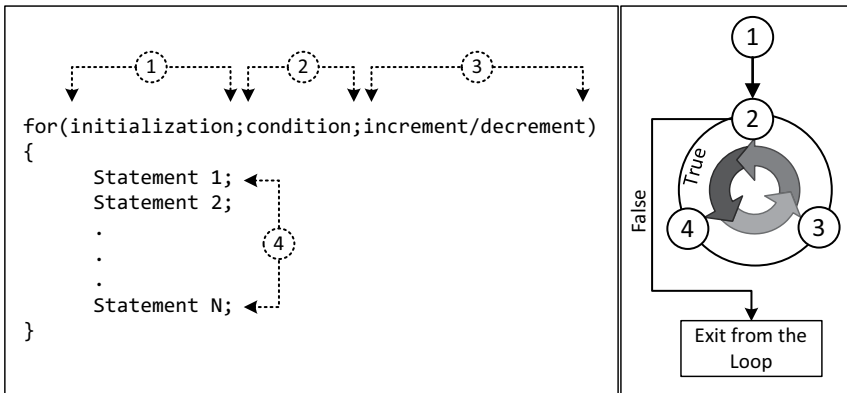
## 8.12 for Loops

A *for loop* is a control statement which allows the code to execute repeatedly. It is the most popular looping construct in C. The for loop combines all the sections of a looping control structure into a single line, though the execution is the same as other looping techniques.

There is a slight difference between the syntax of a while loop and a for loop. Here the initialization, condition, and update section are written in a single line. Figure 8.18 shows the syntax of the for loop.



**FIGURE 8.18**  
for loop syntax.



**FIGURE 8.19**  
Execution of the for loop.

- The statement inside the *for* loop is executed until the condition becomes false;
- Control first initializes the variable, then checks the condition, and enters the loop for executing the statements.
- After executing the statements, it will increment the loop counter and again will check the condition.

To understand the execution procedure, we need to observe the syntax in a different fashion, as shown in Figure 8.19. Let us assign numbers from 1 to 4 to the different components of a for loop. As you can see, the initialization statement is assigned with ①, the condition with ②, the increment/decrement with ③, and finally, statements inside the for-loop body are assigned with ④. ① executes only once, and ② to ④ will iterate until the condition in ② is satisfied. The line designated as “true” indicates that when the condition in ② is satisfied, control starts executing all the statements present in ④. Then it executes the increment/decrement statement in ③ and again checks the condition in ②. This way it will form an execution circle. When the condition in ② becomes false, control exits from the loop.

The *for* loop can take different forms. The following programs show all the forms of the *for* loop for printing the natural numbers from 1 to 10.

```
I. #include<stdio.h>
void main()
{
 int i=1;
 for(; i<=10 ; i++)
 printf("%d", i);
}
```

```
III.#include<stdio.h>
void main()
{
 int i;
 for(i=1; i<=10 ;)
 {
 printf("%d", i);
 i=i+1;
 }
}
```

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

```
IV. #include<stdio.h>
void main()
{
 int i=1;
 for(; i<=10 ;)
 {
 printf("%d", i);
 i=i+1;
 }
}
```

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

All these programs will produce the output:

1 2 3 4 5 6 7 8 9 10

Students are encouraged to write the above code using any C compiler and verify the output and try to analyze the execution process of these code segments.

- It is also possible to use two initialization statements, two increment/decrements, and two conditions inside a **for loop**. But they should be separated by commas. In the following Program 8.18, we display the numbers from 1 to 10 and 10 to 1 simultaneously using a single for loop. The code follows a simple structure and is self-explanatory.

#### PROGRAM 8.18

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

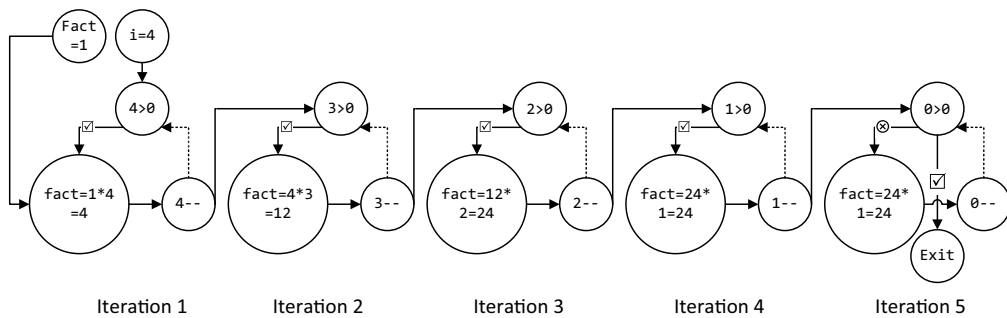
*Output:*

```
1 10
2 9
3 8
4 7
5 6
6 5
7 4
8 3
9 2
10 1
```

In the following section, we will write code for some sample programs that reflects the property of the for loop and its strengths as compared to the other loop control structures discussed above.

Now let's write a program to find the factorial of a number ( $n$ ), which is calculated by the formula:  $factorial(n) = n \times (n - 1) \times (n - 2) \dots \times 1$

For example:  $factorial(4) = 4 \times 3 \times 2 \times 1 = 24$ . The code is shown in Program 8.19 and the execution of the program is shown in Figure 8.20. The figure shows the iterations for  $num=4$ .



**FIGURE 8.20**  
Execution of Program 8.18.

### PROGRAM 8.19

```

1. #include<stdio.h>
2. void main()
3. {
4. int num,i,fact=1;
5. printf("Enter the number:");
6. scanf("%d",&num);
7. for(i=num;i>0;i--)
8. {
9. fact=fact*i;
10. }
11. printf("\nNumber= %d Factorial=%d",num,fact);
12. }
```

*Output:*

```

Enter the number:5
Number= 5 Factorial=120
```

The next program we consider is to check whether a number is a palindrome or not. We already have written the code in Program 8.17. Let us rewrite it using a for loop.

**PROGRAM 8.20**

```

1. #include<stdio.h>
2. void main()
3. {
4. int N,rev=0,digit,temp;
5. printf("Input the number:");
6. scanf("%d",&N);
7. temp=N;
8. for(; N!=0 ; N=N/10)
9. {
10. digit=N%10;
11. rev=rev*10+digit;
12. }
13. if(temp==rev)
14. printf("\n PALINDROME NUMBER ");
15. else
16. printf("\n NOT PALINDROME ");
17. }

```

*Output:*

```

Run-1
Input the number:12521
PALINDROME NUMBER

Run-2
Input the number:4532
NOT PALINDROME

```

**8.12.1 Some Solved Problems (Printing Patterns)**

```

#include<stdio.h>
void main()
{
 int n,i,k,c=1;
 printf("Enter the number of rows: ");
 scanf("%d",&n);
 for(i=1;i<=n;i++)
 {
 for(k=1;k<=i;k++)
 {
 printf(" %d ", c);
 c++;
 }
 printf("\n");
 }
}

```

```

Output
Enter the number of rows: 4
1
2 3
4 5 6
7 8 9 10

```



```
#include<stdio.h>
void main()
{
 int n,i,k;
 printf("Enter the number of rows: ");
 scanf("%d",&n);
 for(i=1;i<=n;i++)
 {
 for(k=1;k<=i;k++)
 {
 printf(" %d ", k);
 }
 printf("\n");
 }
}
```

```
Output
Enter the number of rows: 4
1
1 2
1 2 3
1 2 3 4
```

```
#include<stdio.h>
void main()
{
 int n,i,k;
 printf("Enter the number of rows: ");
 scanf("%d",&n);
 for(i=1;i<=n;i++)
 {
 for(k=1;k<=i;k++)
 {
 printf(" %d ", i);
 }
 printf("\n");
 }
}
```

```
Output
Enter the number of rows: 4
1
2 2
3 3 3
4 4 4 4
```

```
#include<stdio.h>
void main()
{
 int n,i,j;
 printf("Enter the number of rows: ");
 scanf("%d",&n);
 for(i=1;i<=n;i++)
 {
 for(j=n;j>=i;j--)
 {
 printf(" %d ", j);
 }
 printf("\n");
 }
}
```

```
Output
Enter the number of rows: 4
4 3 2 1
4 3 2
4 3
4
```

```

#include<stdio.h>
void main()
{
 int n,i,j,k;
 printf("Enter the number of rows: ");
 scanf("%d",&n);
 for(i=1;i<=n;i++)
 {
 for(j=1;j<=n-i;j++)
 {
 printf(" ");
 }
 for(k=1;k<=i;k++)
 {
 printf("%d", k);
 }
 printf("\n");
 }
}

```

```

Output
Enter the number of rows: 5
1
12
123
1234
12345

```

```

#include<stdio.h>
void main()
{
 int n,i,j,k;
 printf("Enter the number of rows: ");
 scanf("%d",&n);
 for(i=1;i<=n;i++)
 {
 for(j=n;j>=i;j--)
 {
 printf(" ");
 }
 for(k=1;k<=i;k++)
 {
 printf(" ");
 printf("%d", k);
 }
 printf("\n");
 }
}

```

```

Output
Enter the number of rows: 5
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5

```

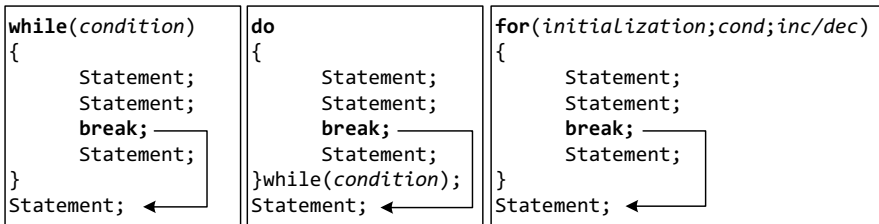
---

## 8.13 Unconditional Branching: break and continue

C programmers use two different keywords that come under unconditional branching statements known as break and continue. These two keywords are used either to come out of the loop or to restart the loop. The details of both is discussed in the following section.

### 8.13.1 break Statements

Sometimes we want to jump out of the loop immediately, without waiting to check the conditional. This can be done using the break statement.



**FIGURE 8.21**  
Execution of break in different kinds of loop.

- When break is encountered inside any C loop, control automatically passes to the first statement after the loop. Figure 8.21 shows the control move whenever a break is encountered in different loop control structures.

Let us take a complete example program to show you how the break statement works.

#### PROGRAM 8.21

```
1. #include<stdio.h>
2. void main()
3. {
4. int i=1;
5. while(i<=100)
6. {
7. printf("%d ",i);
8. if(i==10)
9. break;
10. i=i+1;
11. }
12. printf("\nBreak Encounters");
13. }
```

*Output:*

```
1 2 3 4 5 6 7 8 9 10
Break Encounters
```

*Explanation:*

Even if the condition given is  $i \leq 100$  (line 5), it will print 1 to 10 and terminate because the moment  $i$  becomes 10, the break statement will be encountered which takes control out of the while loop; the last printf statement (line 12) is printed.

### 8.13.2 continue Statements

Sometimes we want to take control to the beginning of the loop, skipping the statements inside the loop which have not yet been executed. This can be achieved by the **continue** statement.

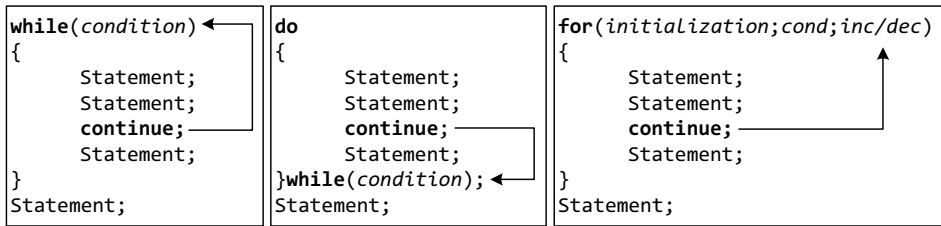


FIGURE 8.22

Execution of continue in different types of loop.

- The **continue** statement is somewhat the opposite of the **break** statement. It forces the next iteration of the loop to take place, skipping any code in between. Figure 8.22 shows how control moves from one place to another when continue statements are executed.

Consider Program 8.22 which shows how the continue keyword is incorporated inside a loop.

#### PROGRAM 8.22

```
1. #include<stdio.h>
2. void main()
3. {
4. int i=0;
5. while(++i<=20)
6. {
7. if(i%3==0)
8. continue;
9. printf("%d ",i);
10. }
11. }
```

*Output:*

1 2 4 5 7 8 10 11 13 14 16 17 19 20

*Explanation:*

The program will display the numbers 1 to 20 except those which are divisible by 3, because whenever  $i$  is divisible by 3 the if statement  $\text{if}(i\%3==0)$  is evaluated to true, which causes the execution of the **continue** statement, which in turn passes control to the while loop without executing the printf statement.

## 8.14 Review Questions

### 8.14.1 Short Questions

1. What is the purpose of a continue statement?
2. What is the use of a break statement?
3. \_\_\_\_\_ is an arithmetic operator that gives the remainder of a division problem.
4. The switch statement and if statements are examples of \_\_\_\_\_ statements.
5. In a switch statement the \_\_\_\_\_ branch is followed if none of the case expressions match the given switch expression.
6. A block of code that repeats forever is called \_\_\_\_\_.
7. In the conditional statement  $\text{if}(++\text{number} < 9)$ , the comparison  $\text{number} < 9$  is made \_\_\_\_\_ and the number is incremented \_\_\_\_\_. (Choose "first" or "second" for each blank.)
8. A loop within a loop is called a \_\_\_\_\_.
9. In a nested loop the \_\_\_\_\_ loop goes through all of its iterations for each iteration of the \_\_\_\_\_ loop. (Choose "inner" or "outer" for each blank.)
10. The \_\_\_\_\_ statement is used to skip the rest of the statements in a loop and start a new iteration without terminating the loop.

### 8.14.2 Long Questions

1. Write a program to check whether a number entered by the user is even or odd.
2. Write a program to find the greatest among three numbers by using an if statement.
3. Write a program to find the greatest among three numbers by using a nested if-else statement.
4. Write a program to find the greatest among three numbers by using an if statement and logical operators.
5. Write a program to find the greatest among four numbers.
6. While purchasing certain items, a discount of 10% is offered if the quantity purchased is more than 1000. If the quantity and price per item are input through the keyboard, write a program to calculate the total expenses.
7. Write a program to check whether the number inputted by the user is zero or non-zero.
8. An electric distribution company charges its domestic consumers as follows:

| Consumption in Units   | Rate of Charges                   |
|------------------------|-----------------------------------|
| 0–200                  | Rs 0.50 per unit                  |
| 201 to 400 Rs 100 plus | Rs 0.65 per unit in excess of 200 |
| 401 to 600 Rs 230 plus | Rs 0.80 per unit in excess of 400 |
| Above 600 Rs 425 plus  | Rs 1.25 per unit in excess of 600 |

Write a program to find out the total amount paid by a customer if the number of units consumed by the customer is entered by him or her.

9. A company has introduced a policy of recruiting employees based on their sex and age. The policy is as follows:
- For a female category the eligibility criterion is that the age of a person should be more than 24.
  - For a male category the age of a person should be more than 28.

Write a program to find out whether a person can be employed if the age and sex of the person is entered through the keyboard.

10. Calculate the commission for a sales representative as per the sales amount:
- if sales  $\leq$  Rs 500, commission is 5%.
  - if sales  $>$  500 and  $\leq$  5000, commission is Rs 35 plus 10% above Rs 500.
  - if sales  $>$  2000 and  $\leq$  5000, commission is Rs 185 plus 12% above Rs 2000.
  - if sales  $>$  5000, commission is 12.5%.
11. A company insures its drivers in the following cases:
- if the driver is married;
  - If the driver is unmarried, male, and above 30 years of age;
  - If the driver is unmarried, female, and above 25 years of age.

In all other cases the driver is not insured. If the marital status, sex, and age of the driver are the inputs, write a program to determine whether the driver is to be insured or not.

12. The marks obtained by a student in five different subjects are input through the keyboard. The student is awarded a division as per the following rules. Write a program that takes five subject marks as input and produces the student's division as output.

|                                 |                 |
|---------------------------------|-----------------|
| Percentage above or equal to 60 | First Division  |
| Percentage between 50 and 59    | Second Division |
| Percentage between 40 and 49    | Third Division  |
| Percentage less than 40         | Fail            |

13. In a company the employee's gross salary is calculated according to the following conditions:
- If basic salary  $<$  1500 then House Rent Allowance (HRA)=10% of Basic and Dearness Allowance (DA)= 25% of Basic;
  - If basic salary  $\geq$  1500 then HRA=Rs 500 and DA= 50% of Basic;
  - The employee salary is input through the keyboard.
14. Draw the flowchart to find the largest of any three numbers.
15. What are the two branch statements for making a decision in C? Give their syntax.
16. Using the two branch statements you have mentioned, write two C programs to solve the following:
- Read a mark X between  $0 \leq X \leq 100$ ;
  - Print "A" if  $80 \leq X$ ;
  - Print "B" if  $60 \leq X < 80$ ;
  - Print "C" if  $40 \leq X < 60$ ;

- Print "D" if  $30 \leq X < 40$ ;
  - Print "F" otherwise.
17. What is meant by "control statements" in C?
  18. Write a short note on if-else statements and compare with conditional operators.
  19. Write the algorithm and C program to find the sum of the digits of a number.
  20. Write a flow chart and C program to reverse a given number N.
  21. Write a program to print ten natural numbers.
  22. Write a program to count the number of even numbers between 1 and 20.
  23. Write a program to count the number of even numbers in a range provided by the user.
  24. Write a program to check whether a number entered by the user is a prime number or not.
  25. Write a program to print all the prime numbers in a range provided by the user.
  26. Write a program to check whether a number is an Armstrong number or not. (Armstrong number: the addition of the cube of the digit of a number = the number itself. For example 153,  $1^3 + 5^3 + 3^3 = 153$ .)
  27. Write a program to count the digits of a number.
  28. Write a program to reverse a number.
  29. Write a program to check whether a number is a palindrome or not.
  30. Write a program to generate the Fibonacci series.
  31. Write a program to find the factorial of a number.
  32. Write a program to print the following pattern using a for loop:

---

|           |           |       |           |
|-----------|-----------|-------|-----------|
| 1         | 1 2 3 4 5 | *     | 1         |
| 1 2       | 1 2 3 4   | **    | 0 1       |
| 1 2 3     | 1 2 3     | ***   | 0 1 0     |
| 1 2 3 4   | 1 2       | ****  | 1 0 1 0   |
| 1 2 3 4 5 | 1         | ***** | 1 0 1 0 1 |

---

33. Write a program to find the average of the numbers entered by the user. Note that a user can enter as many numbers as they want. When the user enters 0, the loop stops and prints the average of the previously inputted numbers.
34. Write a program to print the ASCII value and the corresponding character in tabular format.
35. Write a program to generate the multiplication table up to 10 using a for loop.

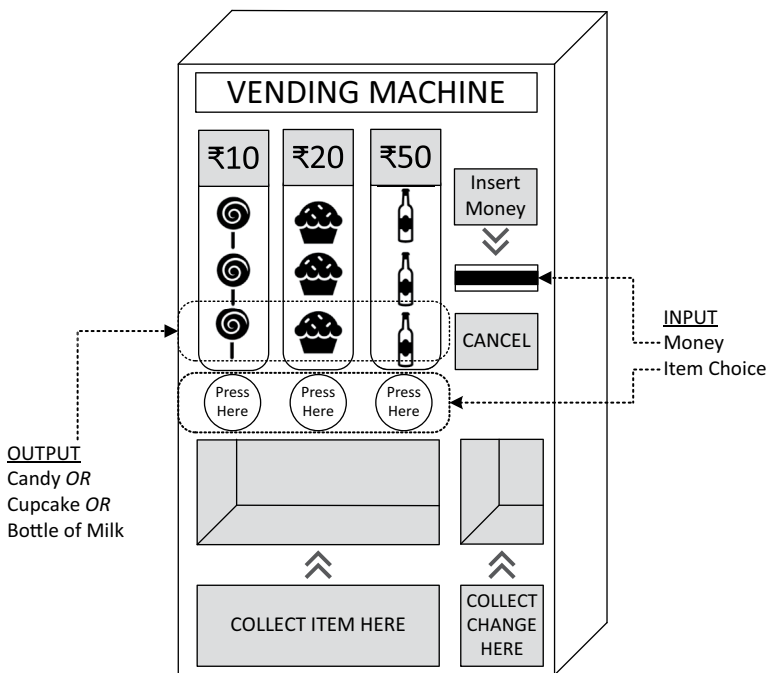
# 9

## Functions

### 9.1 Introduction

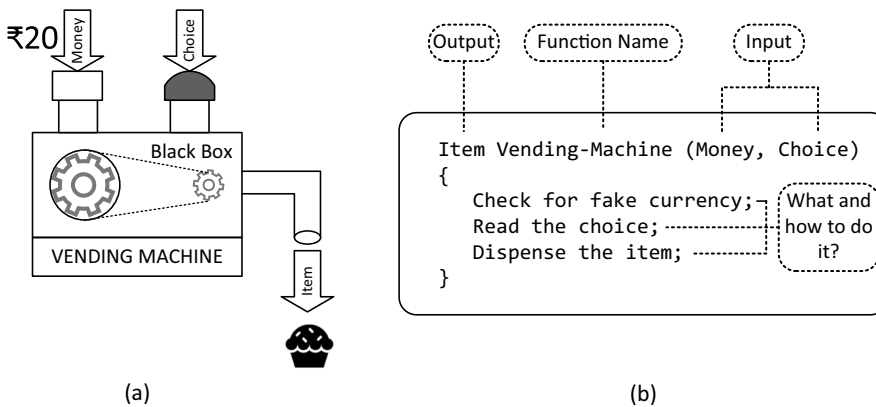
This chapter introduces the concept of a function which is the most fundamental element of any programming language. A function is something that, when called, performs some tasks. We can think of a function as a small machine that takes some input and provides some output. . For example, a vending machine (Figure 9.1)

We all know how a vending machine works. In Figure 9.1, the machine offers three items: candy at ₹10, a cupcake at ₹20, and a milk bottle at ₹50. The user who wishes to obtain an item needs to insert the appropriate amount to the machine and press the desired item button “Press Here”. Then the machine dispenses the item through the “Collect Item Here” box, and the change (if any) through the “Collect Change Here” box. There is a cancel button that helps the user to cancel the transaction at any point in time.



**FIGURE 9.1**  
A vending machine example for understanding a function.





**FIGURE 9.2**  
Vending machine analogy and overview of function declaration.

Now, let us understand some points with respect to the vending machine that helps us understand the function concept later in this chapter. We can think of a vending machine as a black box because we do not know how it functions. There might be a person sat inside the machine who collects your money, and dispenses the item and change; or the machine may be programmed in such a way that it can read the user input and act accordingly. Whatever it may be, the machine takes some input; in our case, it takes two inputs: money, choice of item. It produces some output; in this case, it dispenses an item: candy or a cupcake or a milk bottle.

In a more general way, we can think of a vending machine as a function that takes two inputs: money and choice, and produces one output: item. For easy understanding, ignore the change dispensed by the vending machine; assume that the user can only insert the exact amount of ₹10, ₹20, or ₹50. The analogy is shown in Figure 9.2.

Think like a programmer; Figure 9.2b is the programmatic representation of Figure 9.2a, where the vending machine represents the name of the function, money and choice are treated as the input to the function, and the item is the output produced by the function. You can also observe that some statements are enclosed within two curly braces; these are the steps followed by the vending machine to dispense the item. These steps are called the body of the function, which specifies how to process the input to produce the desired output. In our example, we might first check whether the inputted currency is a valid currency or not, what choice has been selected by the user, and finally dispense the selected item.

The above example is just an analogy to understand what a function is and how we are going to write it in C code. But, the actual syntax may slightly differ as we will discuss in a later section. After completing this chapter, readers will be able to:

- Write functions for any given problem and describe the need for using a function.
- Define and differentiate the different categories of functions available in C.
- Explain the components of a function and how they work.
- Know what a recursive function is and how to write one.
- Know all the different kinds of storage classes available in C.

## 9.2 The Need for Functions

There are several uses of a function. We will take an example to show you why a function is needed. Suppose you are a carpenter and you build different types of wooden furniture. One day you receive an order to cut 10,000 pieces of wood in a zigzag manner, as shown in Figure 9.3. Assume that no cutting machine has yet been developed to cut a piece of wood in a zigzag pattern. As the order is significant, and you must finish it on time, you need a cutter for this work. So, you decide to build a cutting machine. Your problem is solved, and you have a machine that can be used in the future for the same type of work.

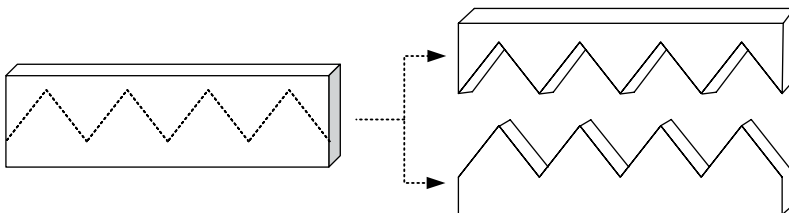
What do we learn from the above example? A machine was developed that performs a specific task. It can be reused for the same type of work. It takes a piece of wood as input and generates two pieces of wood in a zigzag pattern. Here, the machine is a function designed to solve a specific problem, and it can be reused any number of times. Similarly, during our program development, we can create some functions for solving a specific task, and then use that function several times throughout our code.

We have already used several *predefined functions* like `printf()` and `scanf()` for console I/O purposes. We have used these functions in every program. In fact, whatever program we have written till now contains at least one function, i.e., the `main()` function. That means some functions were there already as predefined functions; some functions we wrote for our purposes are called user-defined functions.

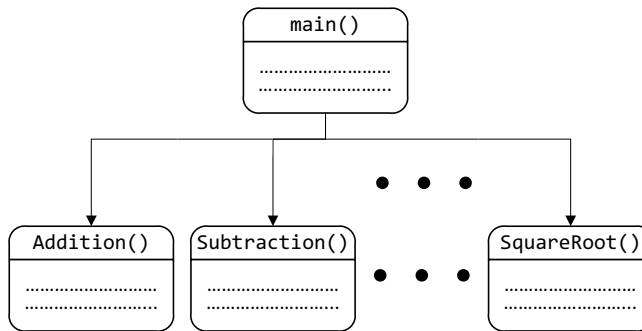
Let us think in another way. Suppose you have a large task to do. The usual procedure is to divide the task into smaller tasks, solve each one separately, and combine them to solve the bigger problem. This process is called modular programming, and modular programming employs the concept of functions. After we divide the whole task into subproblems, each subproblem can be implemented with a function. Combining the result of all functions, we can solve a bigger problem. Consider the problem of designing a calculator. A calculator performs many tasks like addition, subtraction, multiplication, and square roots. We can write code for each task using a separate function and call them with the `main()` function to solve a bigger task. Figure 9.4 shows the modular division of tasks for a calculator design.

The advantage of using functions is:

- Modular programming;
- Reduction in the amount of work and development time;
- Program and function debugging are easier;



**FIGURE 9.3**  
Woodcutting in a zigzag pattern.



**FIGURE 9.4**  
Modular design of the task and each task employed with a function.

- Reduction in size of the program;
- Reuse of code.

### 9.3 Types of Function

The C programming language supports two types of functions:

1. Library functions or predefined functions;
2. User-defined functions.

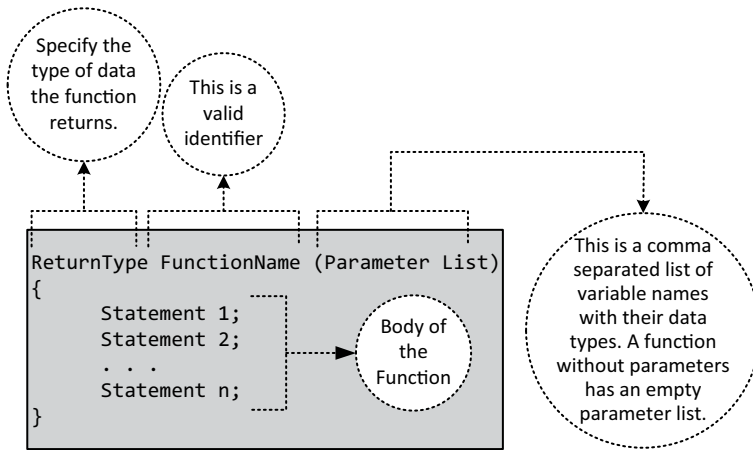
We have come across many library functions like `printf()` and `scanf()` which were used in our previous programs, and the definition of these functions are predefined. In the following section, we are going to discuss user-defined functions.

The basic difference between these two types of function is that we do not write library functions; we only use them, whereas we write user-defined functions for specific problems.

### 9.4 User-defined Functions

A function can be defined as a group of statements enclosed within a block with a valid identifier and can perform a specific task. Generally, a function will process information that is passed to it from the calling portion of the program and return a single value. Information is passed to the function via special identifiers called arguments (also called parameters) and returned via the return statement. A user-defined function only executes when it is called.

A function can be defined as a group of statements enclosed within a block with a valid identifier and can perform a specific task.



**FIGURE 9.5**  
Syntax of user-defined function.

The general syntax of a user-defined function is shown in Figure 9.5.

So, to begin our discussion, let us write a simple function using the above syntax. In this function, we include a single statement body. The work of this function is to print a line of a statement.

A simple one line function

```
void printLine()
{
 printf("\n C Programming Learn to Code");
}
```

- In the above function, it is clearly mentioned that the function does not return anything, so the return type is void;
- This function also does not take any parameters, so the parameter list is empty;
- The name of the function is printLine, and when this function is called, it executes a single line present in the body of the function.

As discussed earlier, a user-defined function cannot be executed of its own. It should be called by the main() function whenever it is needed.

**PROGRAM 9.1**

```
1. #include<stdio.h>
2. void printLine()
3. {
4. printf("\n C Programming Learn to Code");
5. }
```

```
6. void main()
7. {
8. printLine();
9. printLine();
10. printLine();
11. }
```

*Output:*

```
C Programming Learn to Code
C Programming Learn to Code
C Programming Learn to Code
```

*Explanation:*

The `printLine()` function is called three times, so it prints “C Programming Learn to Code” three times. We can solve the above problem by writing three `printf()` statements inside the function `main()` as shown in Program 9.2, and we get the same output as from Program 9.1.

Now the question is *why is the function necessary?* The above `printLine()` function is written for the sake of understanding the concept of a function. To explain the benefit of the function, let us modify the `printLine()` function as Program 9.3:

### PROGRAM 9.2

```
1. #include<stdio.h>
2. void main()
3. {
4. printf("\n C Programming Learn to Code");
5. printf("\n C Programming Learn to Code");
6. printf("\n C Programming Learn to Code");
7. }
```

*Output:*

```
C Programming Learn to Code
C Programming Learn to Code
C Programming Learn to Code
```

### PROGRAM 9.3

```
1. #include<stdio.h>
2. void printLine()
3. {
```

```

4. printf("\n *****");
5. printf("\n C Programming Learn to Code");
6. printf("\n *****");
7. }
8. void main()
9. {
10. printLine();
11. printLine();
12. printLine();
13. }

```

The function contains three statements to execute. So, the output of the program is as:

```

C Programming Learn to Code

C Programming Learn to Code

C Programming Learn to Code

```

Now, if I want to write the above program without using a function, then I require nine `printf()` statements to be written inside the function `main()`. So, this increases the program size. If the program size grows, then debugging also takes more time. But by using a function, we obtain benefits such as:

- The concept of modular programming: that is, we can divide a bigger problem into smaller problems, and all smaller problems can be developed individually and used whenever necessary.
- Reduction in the program size.
- Debugging is easy.
- Code reusability.

Let us take another example. Suppose I want to find out the cube of a number using a function. Now the function has to take a single parameter.

#### PROGRAM 9.4

```

1. #include<stdio.h>
2. void cube(int x)
3. {
4. int c;
5. c=x*x*x;

```

```
6. printf("\n CUBE OF THE NUMBER=%d", c);
7. }
8. void main()
9. {
10. int n;
11. printf("\n ENTER A NUMBER: ");
12. scanf("%d", &n);
13. cube(n);
14. }
```

*Output:*

```
ENTER A NUMBER: 4
CUBE OF THE NUMBER = 64
```

*Explanation:*

- The execution of the above program starts from the `main()` function and line 13 calls the `cube` function by passing `n` as an argument.
- The value of `n` will be copied to `x` in the function `cube()`. `x` is a variable local to the function `cube()` and `n` is a variable local to the function `main()`. But both the variables contain the same value, because the `n` value is copied to `x`.
- The `cube()` function will multiply the `x` value (indirectly it is `n`) three times and store it in another variable `c`.
- The `printf()` function inside the `cube()` function prints the cube of `x` (indirectly it is the cube of `n`). So, we obtain our result.

---

## 9.5 Components and Working of a Function

The `cube()` function in Program 9.4 is written before the `main()` function, and the return type of the function is `void` (which means it does not return anything). But the function can be written after the `main()` function, and a function can also return a value. Generally, a function always returns a single value.

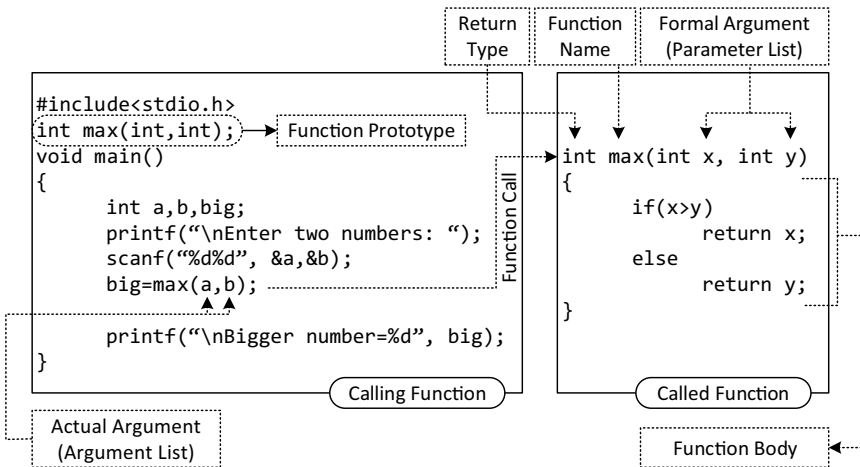
There are various terms associated with functions. A function can take various forms. So before discussing the detail, let us first understand how a function can be written and how it works. Figure 9.6 illustrates the different components of a function, and it computes the maximum of two integer numbers.

### 9.5.1 Calling Function

The function which calls other functions is known as the calling function. Here, the `main()` function is the calling function because it calls the function `max()`.

### 9.5.2 Called Function

The function which is called by other functions is known as a called function. Here `max()` function is the called function.



**FIGURE 9.6**  
Components of a function.

### 9.5.3 Function Prototype

A function prototype is called the declaration of a function. We know that any variable or identifier must be declared before it is used inside the program. As the function is also an identifier, it should be declared before it is used. Hence, the declaration of the function is known as a function prototype.

A function prototype provides the following information to the compiler:

- The name of the function;
- The type of the value returned by the function;
- The number and the type of arguments that are passed in a call to the function.

The prototype declaration should be done before the function `main()`, and it should end with a semicolon. The prototype declaration can be eliminated by defining the function before calling it. In the above example (Figure 9.6), if we write the `max()` function above the `main()` function then prototype declaration is not required.

#### NOTE

A function can be defined before or after the `main()` function. If the function definition is present before the `main()` function then a prototype is not required. In Program 9.4 we have not used the prototype because the function `cube()` is defined before the `main()` function.

### 9.5.4 Function Definition

The function itself is referred to as a function definition. The first line of the function definition is known as the **function declarator** and is followed by the **function body**.



### 9.5.5 Function Call

A function is a dormant entity, which comes to life only when a call to the function is made. A function call is specified by the function name followed by the arguments enclosed in parentheses and terminated by semicolons.

In Figure 9.6, the line `c = max ( a , b ) ;` invokes the function `max ( )`.

### 9.5.6 Actual Arguments

The arguments which are specified in the function call are known as actual arguments. In this example (Figure 9.6), `a` and `b` are the actual arguments.

### 9.5.7 Formal Arguments

The parameters specified in the function definition are known as formal parameters. In this example (Figure 9.6), `x` and `y` are the formal parameters.

### 9.5.8 Return Type

The function may return integer, char, float, etc. When the function does not return anything then we use the void keyword. In this example (Figure 9.6), the function returns an integer. The calling function must be able to receive the value returned by the called function. In the program the variable `c` is used to receive the value returned by the function `max ( )`.

To understand more about functions, let us write a program with and without using a function and analyze the differences. We will write a program to check whether a number is prime, as well as whether it is a perfect number or not.

*Prime number:* This is a number greater than 1 and cannot be made by multiplying other whole numbers. We can also define a prime number as a number that can only be divisible by 1 and itself. For example, 5 is a prime number because it cannot be divided by 2, 3, and 4.

*Perfect number:* This is a positive integer that is equal to the sum of its divisor. For example, 6 is a perfect number because the perfect divisor of 6 is 1, 2, and 3;  $1 + 2 + 3 = 6$ .

The above problem needs to solve two tasks:

1. We should write the code to check whether the number is a prime number or not;
2. And whether the number is a perfect number or not.

#### PROGRAM 9.5

```
1. #include<stdio.h>
2. void main()
3. {
4. int N, i, sum=0, count=0;
5. printf("\nEnter a number: ");
6. scanf ("%d", &N);
```

```
7.
8. /*Code to check whether the number is prime or not*/
9. for(i=1;i<=N;i++)
10. {
11. if(N%i==0)
12. count=count+1;
13. }
14. if(count==2)
15. printf("\nTHE NUMBER IS A PRIME NUMBER");
16. else
17. printf("\nTHE NUMBER IS NOT A PRIME NUMBER");

18. /*Code to check whether the number is a perfect number or
19. not*/
20. for(i=1;i<=N/2;i++)
21. {
22. if(N%i==0)
23. sum=sum+i;
24. }
25. if(sum==N)
26. printf("\nTHE NUMBER IS A PERFECT NUMBER");
27. else
28. printf("\nTHE NUMBER IS NOT A PERFECT NUMBER");
29. }
```

*Output:*

Run-1

Enter a number: 28

THE NUMBER IS NOT A PRIME NUMBER

THE NUMBER IS A PERFECT NUMBER

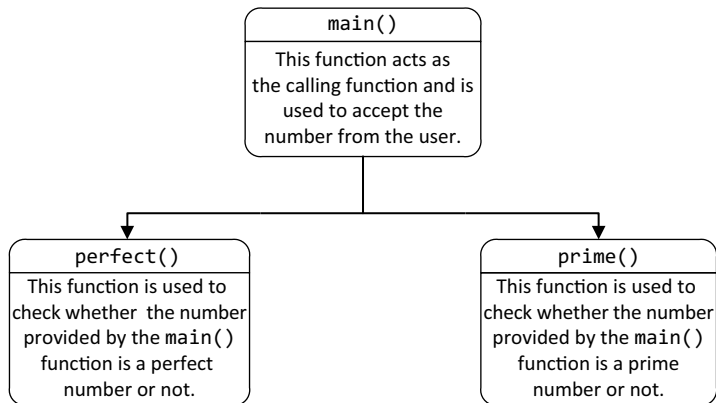
Run-2

Enter a number: 43

THE NUMBER IS A PRIME NUMBER

THE NUMBER IS NOT A PERFECT NUMBER

By analyzing the above Program 9.5, we conclude that if we solve multiple works inside a single function then the program size increases and is prone to errors. Debugging of the above program is also difficult. To avoid this difficulty, we can use the concept of a function. We can separate the above functionality into three functions as shown in Figure 9.7.



**FIGURE 9.7**  
Separating functionality using different functions.

### PROGRAM 9.6

```

1. #include<stdio.h>
2. void Prime(int);
3. void Perfect(int);
4. void main()
5. {
6. int No;
7. printf("\nEnter a number: ");
8. scanf("%d", &No);
9. Prime(No);
10. Perfect(No);
11. }
12. void Prime(int N)
13. {
14. int i,count=0;
15. for(i=1;i<=N;i++)
16. {
17. if(N%i==0)
18. count=count+1;
19. }
20. if(count==2)
21. printf("\nTHE NUMBER IS A PRIME NUMBER");
22. else
23. printf("\nTHE NUMBER IS NOT A PRIME NUMBER");
24. }
25. void Perfect(int N)
26. {
27. int i,sum=0;

```

```
28. for(i=1;i<=N/2;i++)
29. {
30. if(N%i==0)
31. sum=sum+i;
32. }
33. if(sum==N)
34. printf("\nTHE NUMBER IS A PERFECT NUMBER");
35. else
36. printf("\nTHE NUMBER IS NOT A PERFECT NUMBER");
37. }
```

*Output:*

Run-1

Enter a number: 28

THE NUMBER IS NOT A PRIME NUMBER

THE NUMBER IS A PERFECT NUMBER

Run-2

Enter a number: 43

THE NUMBER IS A PRIME NUMBER

THE NUMBER IS NOT A PERFECT NUMBER

Analyzing the above program we can say that it is more readable, and error finding in the program is easy because the tasks are individually developed by different functions.

---

## 9.6 Categories of a Function

A function can take different forms. In this section we are going to discuss the different ways a function can be written.

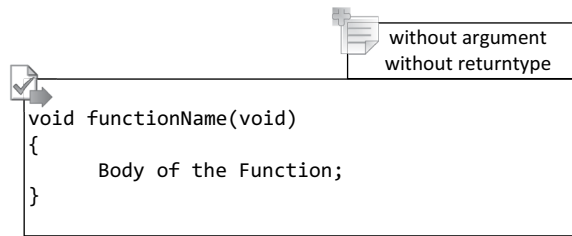
### 9.6.1 A Function Without Arguments and Without Return Types

This format of the function has no return type, that is, the return type will be void; also these functions do not take any arguments. The `println()` function discussed in Program 9.1 is an example of this type of function. The syntax of these categories of functions is given in Figure 9.8.

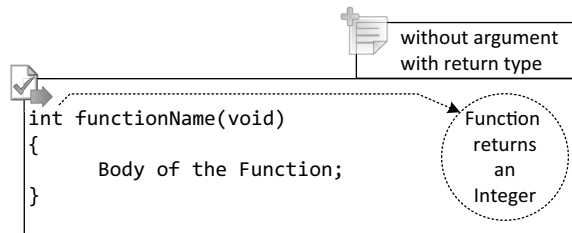
### 9.6.2 A Function Without Arguments and With Return Types

Sometimes the function does not take any arguments, but it may return a value. The general format for this type of function is shown in Figure 9.9.

A sample program of this kind is shown in Program 9.7.

**FIGURE 9.8**

General structure of a function without arguments and without return types.

**FIGURE 9.9**

General structure of a function without arguments and with return types.

### PROGRAM 9.7

```

1. #include<stdio.h>
2. int test()
3. {
4. int x=7;
5. return x;
6. }
7. void main()
8. {
9. int y;
10. printf("Enter a number: ");
11. scanf("%d", &y);
12. y=y+test();
13. printf("Total= %d", y);
14. }

```

*Output:*

```

Enter a number: 15
Total= 22

```

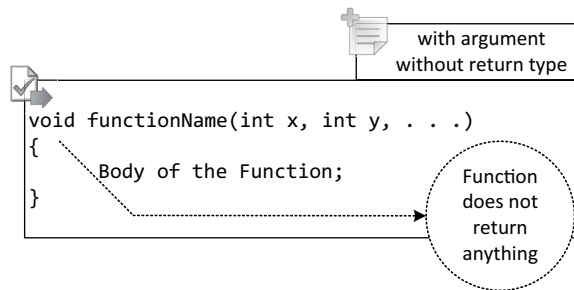
### 9.6.3 A Function With Arguments and Without Return Types

A function may take some value from the calling function as an argument but may not return any value. These type of functions come under this category. `cube()` functions and `prime()` functions, discussed above, come under this category. The general structure is shown in Figure 9.10.

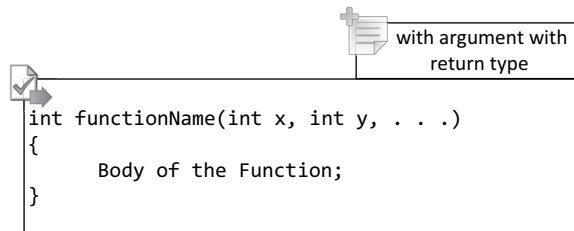
### 9.6.4 A Function With Arguments and With Return Types

The most used format in the categories of function is "with arguments and with return types." In this type, a function takes some argument from the calling function and returns a value to the calling function. We have already discussed the `max()` function in this chapter, which comes under this category. The general structure is shown in Figure 9.11.

In the following section, we have written four programs (Programs 9.8a–d) using functions. All the functions do the same work (i.e., add two numbers). We have written the programs in such a manner that they satisfy all the categories of functions. The objective is to show you how a function can take different forms. Notice the differences among them. In Program 9.8a, the function call line (line 13) requires a variable `c` to catch the value returned from the function. In Program 9.8b, the function call line (line 13) does not require any variable because the function returns nothing. We give a similar explanation for the other two cases (Programs 9.8c and d). The function call line (line 13) in Programs 9.8a and 9.8b passes two arguments `a` and `b`; hence the corresponding functions require two variables `x` and `y` to catch that value. The other two cases do not need any argument because



**FIGURE 9.10**  
General structure of a function with arguments and without return types.



**FIGURE 9.11**  
General structure of a function with arguments and with return types.

the actual function does not require the value from the `main()` function. Readers are encouraged to test every program and observe the output produced.

**PROGRAM 9.8 (a)**

```

1. #include<stdio.h>
2. int add(int x, int y)
3. {
4. int z;
5. z=x+y;
6. return z;
7. }
8. void main()
9. {
10. int a,b,c;
11. printf("\nEnter two
 numbers: ");
12. scanf("%d %d", &a,&b);
13. c=add(a,b);
14. printf("\nResult = %d", c);
15. }
```

With argument with return type

**PROGRAM 9.8 (b)**

```

1. #include<stdio.h>
2. void add(int x, int y)
3. {
4. int z;
5. z=x+y;
6. printf("\nResult= %d ", z);
7. }
8. void main()
9. {
10. int a,b;
11. printf("\nEnter two
 numbers: ");
12. scanf("%d %d", &a,&b);
13. add(a,b);
14. }
15. }
```

With argument without return type

**PROGRAM 9.8 (c)**

```

1. #include<stdio.h>
2. int add()
3. {
4. int x,y,z;
5. printf("\nEnter two
 numbers: ");
6. scanf("%d %d", &x,&y);
7. z=x+y;
8. return z;
9.
10. }
11. void main()
12. {
13. int c;
14. c=add();
15. printf("\nResult = %d", c);
16. }
```

Without argument with return type

**PROGRAM 9.8 (d)**

```

1. #include<stdio.h>
2. void add()
3. {
4. int x,y,z;
5. printf("\nEnter two
 numbers: ");
6. scanf("%d %d", &x,&y);
7. z=x+y;
8. printf("\nResult = %d", z);
9.
10. }
12. void main()
13. {
14. add();
 }
```

Without argument without return type

## 9.7 Recursion

In general, we can solve a problem using two approaches. One approach uses loops, and the other uses recursion. Recursion is a repetitive process in which a function calls itself. Some older languages (like COBOL) do not support recursion. To explain both approaches, let us first take a problem then solve it by each approach. The problem is *the factorial of a number*.

*Iterative definition:* The factorial of a given number is a product of the integral values from 1 to the number. The definition is as follows:

$$\text{factorial}(n) = \begin{cases} 1, & \text{if } n = 0 \\ n \times (n-1) \times (n-2) \dots \times 2 \times 1, & \text{Otherwise} \end{cases}$$

According to this definition we can calculate the value of factorial (3) as follows:

$$\text{Factorial}(3) = 3 \times 2 \times 1 = 6$$

The solution of the problem generally involves a loop. We have already written a program to find the factorial of a number in Program 8.19 Here we will solve it using a function as well as a recursive function. Program 9.9 shows the code to write using a function.

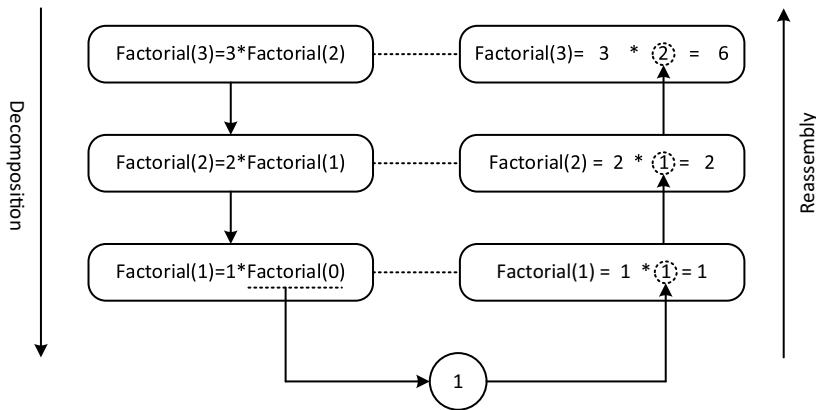
### PROGRAM 9.9: FACTORIAL OF A NUMBER USING FUNCTION

```
1. #include<stdio.h>
2. int factorial(int n)
3. {
4. int f = 1, i;
5. for(i=1;i<=n;i++)
6. f = f * i;
7. return (f);
8. }
9. void main()
10. {
11. int a, fact;
12. printf ("\nEnter a number: ");
13. scanf ("%d",&a);
14. fact = factorial(a);
15. printf ("Factorial value = %d", fact);
16. }
```

*Output:*

```
Enter a number: 5
Factorial value = 120
```





**FIGURE 9.12**  
Decomposition and reassembly of a recursive solution.

*Recursive definition:* A repetitive function defined recursively whenever the function appears within the definition itself. The factorial function can be defined recursively as shown in the following formula:

$$f(x) = \begin{cases} 1, & \text{if } n = 0 \\ n \times \text{factorial}(n-1), & \text{Otherwise} \end{cases}$$

The decomposition of factorial (3), using the above formula, is shown in Figure 9.12. Study the figure carefully and note that the recursive solution for a problem involves a two-way journey: first we decompose the problem from top to bottom, and then we solve it from bottom to top.

*Designing recursive functions:* All recursive functions have two elements: each call either solves one part of the problem or it reduces the size of the problem. The recursive solution to factorial is shown below. This program does not need a loop: the concept itself involves repetition.

#### PROGRAM 9.10: FACTORIAL OF A NUMBER USING A RECURSIVE FUNCTION

```

1. #include<stdio.h>
2. int factorial(int n)
3. {
4. if (n==0)
5. return 1;
6. else
7. return n*factorial(n-1);
8. }
9. void main()

```

```
10. {
11. int n, fact;
12. printf("Enter any number: ");
13. scanf("%d", &n);
14. fact=factorial(n);
15. printf("Factorial= %d", fact);
16. }
```

*Output:*

```
Enter any number 5
Factorial value = 120
```

*Explanation:*

In the above program if  $n==0$  then it returns 1, which means it solves a small piece of the problem  $\text{factorial}(0)$  as 1. On the other hand, the line  $\text{return } n*\text{factorial}(n-1)$  reduces the size of the problem by recursively calling the factorial with  $n-1$ .

The statement that solves the problem is known as a *base case* that stops the decomposition of the problem. Every recursive function must have a base case. The rest of the function is known as the *general case*.

- In our factorial example, the base case is  $\text{factorial}(0)$ .
- The general case is  $n*\text{factorial}(n-1)$ . The general case contains the logic needed to reduce the size of the problem.

In this problem, once the base case or stopping condition has been reached, the solution begins. The program has found one part of the answer and can return that part to the next general statement. Thus in the above problem, after the program has calculated that  $\text{factorial}(0)$  is 1, it returns the value 1.

This leads to solving the next general case:

$$\text{Factorial}(1) = 1 * \text{factorial}(0) = 1 * 1 = 1$$

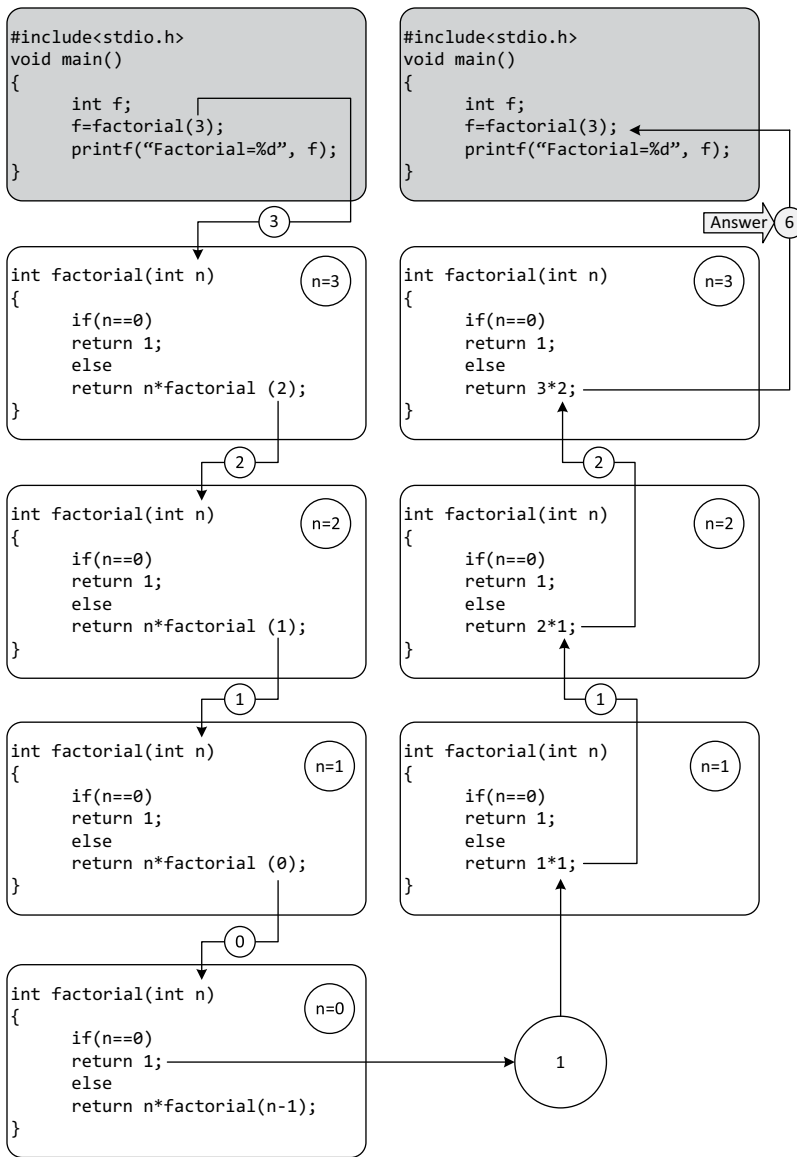
The program now returns the value of  $\text{factorial}(1)$  to the more general case,  $\text{factorial}(2)$ :

$$\text{Factorial}(2) = 2 * \text{factorial}(1) = 2 * 1 = 2$$

As the program solves each general case in turn, the program can solve the next higher general case, until it finally solves the most general case, the original problem.

The designing of recursive functions has the following rules:

- First, determine the base case;
- Then, determine the general case;
- Finally, combine the base case and the general case into a function.



**FIGURE 9.13**  
Representing each recursive call with the return value.

Figure 9.13 shows each recursive call separately by passing the reduced value of *n*; it also shows the return value from each recursive call.

### 9.7.1 Example: Find the Value of $x^y$

The solution to this problem can also be done in two ways: by an iterative method or recursive method. The iterative method employs a loop where *x* is multiplied *y* times. The iterative solution of this problem is shown in Program 9.11.

**PROGRAM 9.11: FIND THE VALUE OF  $X^Y$  USING A FUNCTION**

```

1. #include<stdio.h>
2. int power(int x, int y)
3. {
4. int R=1, i=0;
5. for(i=0;i<y;i++)
6. {
7. R= R*x;
8. }
9. return R;
10. }
11. void main()
12. {
13. int x,y,p;
14. printf("Enter x and y value: ");
15. scanf ("%d%d",&x,&y);
16. p=power(x,y);
17. printf("Result=%d ",p);
18. }

```

*Output:*

```

Enter x and y value: 3 4
Result= 81

```

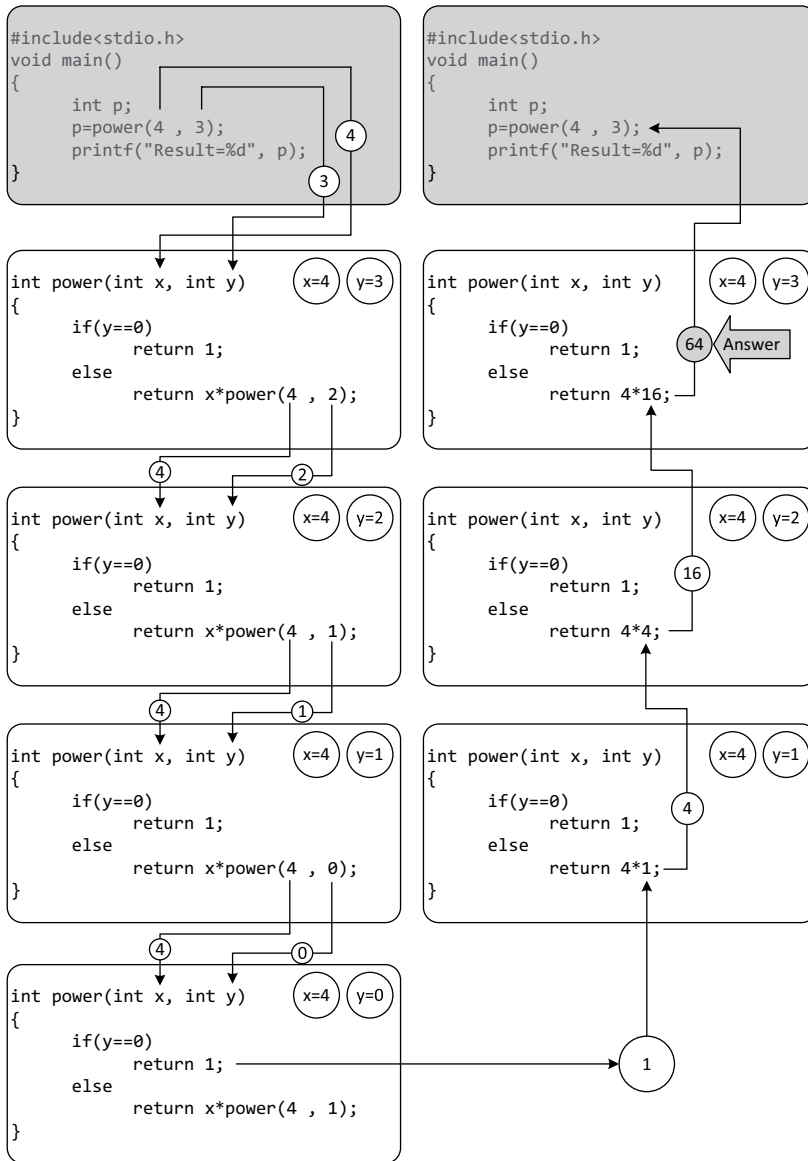
To solve this problem using the recursive method we have to first determine the stopping condition or the base case:

- So here the base case is: if ( $y==0$ ) then  $x^y$  becomes 1.
- The general case will be  $x*\text{power}(x, y-1)$  which reduces the problem into more general cases.

The recursive definition of the above problem can be written as:

$$\text{power}(x,y) = \begin{cases} 1, & \text{if } y = 0 \\ x \times \text{power}(x, y - 1), & \text{Otherwise} \end{cases}$$

Program 9.12 shows the complete C code for the above problem. The execution of the program is described in Figure 9.14 for  $x=4$  and  $y=3$ .



**FIGURE 9.14**  
Execution of Program 9.12.

**PROGRAM 9.12: FIND THE VALUE OF  $X^Y$  USING A RECURSIVE FUNCTION**

```

1. #include<stdio.h>
2. int power(int x, int y)
3. {
4. if(y==0)
5. return 1;
6. else
7. return x*power(x,y-1);
8. }
9. void main()
10. {
11. int x,y,p;
12. printf("Enter x and y value: ");
13. scanf ("%d%d" ,&x,&y) ;
14. p=power(x,y) ;
15. printf("Result=%d ",p) ;
16. }

```

*Output:*

```

Enter x and y value: 4 3
Result = 64

```

**9.7.2 Programming Examples**

In this section, we will show you some programming examples that use the concept of recursion. The objective is not to explain each line of the code; rather, students can take it as a practice program and execute it to see how the recursion works.

**PROGRAM 9.13**

**Write a recursive function to add all the digit of a number.**

```

#include<stdio.h>
int add_digit(int n)
{
 static int r,s;
 if(n==0)
 return 0;
 else
 {
 r=n%10;
 s=s+r;

```

**PROGRAM 9.14**

**Write a recursive function to check a number is palindrome or not.**

```

#include<stdio.h>
int add_digit(int n)
{
 static int r,s;
 if(n==0)
 return 0;
 else
 {
 r=n%10;
 s=s*10+r;

```

```

 add_digit(n/10);
 }
 return s;
}
void main()
{
 int n,res;
 printf("Enter a number ");
 scanf("%d",&n);
 res=add_digit(n);
 printf("Result=%d ",res);
}

```

**PROGRAM 9.15**

**Write a recursive function to reverse a number.**

```

#include<stdio.h>
int rev_digit(int n)
{
 static int r,s;
 if(n==0)
 return 0;
 else
 {
 r=n%10;
 s=s*10+r;
 rev_digit(n/10);
 }
 return s;
}
void main()
{
 int n,res;
 printf("Enter a number ");
 scanf("%d",&n);
 res=add_digit(n);
 printf("Reverse
number=%d",res);
}

```

```

 add_digit(n/10);
 }
 return s;
}
void main()
{
 int n,res;
 printf("Enter a number ");
 scanf("%d",&n);
 res=add_digit(n);
 if(res==n)
 printf("pallindrome");
 else
 printf("Not Pallindrome");
}

```

**PROGRAM 9.16**

**Write a recursive function to add the number from 1 to n.**

```

#include<stdio.h>
int add(int n)
{
 if(n==0)
 return 0;
 else
 return n+add(n-1);
}
void main()
{
 int n,res;
 printf("Enter a number ");
 scanf("%d",&n);
 res=add(n);
 printf("Result=%d",res);
}

```

**PROGRAM 9.17**

**Write a recursive function to print the number from 1 to n.**

```
#include<stdio.h>
void print(int n)
{
 static int i=1;
 if(i>n)
 return;
 else
 {
 printf("%d ",i++);
 print(n);
 }
}
void main()
{
 int n;
 printf("Enter a number ");
 scanf ("%d",&n);
 print (n);
}
```

**PROGRAM 9.19**

**Write a recursive function to check a number is prime number or not.**

```
#include<stdio.h>
int prime(int n)
{
 static int i=1,c=0;
 if(i>n)
 return 0;
 else
 {
 if(n%i==0)
 c=c+1;
 i=i+1;
 prime(n);
 }
 return c;
}
```

**PROGRAM 9.18**

**Write a recursive function to print the number from n to 1.**

```
#include<stdio.h>
void rev_print(int n)
{
 if(n==0)
 return;
 else
 {
 printf("%d ",n);
 print(n-1);
 }
}
void main()
{
 int n;
 printf("Enter a number ");
 scanf ("%d",&n);
 print (n);
}
```

**PROGRAM 9.20**

**Write a recursive function to generate a Fibonacci series.**

```
#include<stdio.h>
void fibonacci(int a,int b,int n)
{
 static int c;
 if(c>n)
 return;
 else
 {
 c=a+b;
 printf("%d ",c);
 a=b;
 b=c;
 print(a,b,n);
 }
}
```



```

void main()
{
 int n,count;
 printf("Enter a number ");
 scanf("%d",&n);
 count=prime(n);
 if(count==2)
 printf("Prime Number");
 else
 printf("Not Prime Number");
}

void main()
{
 int a=0,b=1;
 printf("%d ",a);
 printf("%d ",b);
 fibonacci(a,b,10);
}

```

---

## 9.8 Storage Classes

From the C compiler's point of view, a variable name identifies some physical location within the computer where the string of bits representing the variable's value is stored. There are basically two kinds of locations in a computer where such a value may be kept – memory and CPU registers. Generally, the values which are stored in the CPU register can be executed faster. The storage class determines in which of these two locations the value is stored.

A variable's storage class tells us:

- Where the variable will be stored;
- What will be the initial value of the variable, if the initial value is not specifically assigned (i.e., the default initial value);
- What the scope of the variable is (i.e., in which functions the value of the variable would be available);
- What the life of the variable is (i.e., how long the variable would exist).

There are four storage classes in C:

| Sl. No. | Name                    | Keywords |
|---------|-------------------------|----------|
| 1       | Automatic Storage Class | auto     |
| 2       | Register Storage Class  | register |
| 3       | Static Storage Class    | static   |
| 4       | External Storage Class  | extern   |

All the above storage classes are classified and can be differentiated with the following features:

| Features of Classification    | Automatic (auto)                                     | Register (register)                                  | Static (static)                                                  | External (extern)                       |
|-------------------------------|------------------------------------------------------|------------------------------------------------------|------------------------------------------------------------------|-----------------------------------------|
| Where is the variable stored? | Memory                                               | CPU Register                                         | Memory                                                           | Memory                                  |
| What is the initial value?    | Garbage                                              | Garbage                                              | 0 (Zero)                                                         | 0 (Zero)                                |
| What is the scope?            | Local to the block in which the variable is defined. | Local to the block in which the variable is defined. | Local to the block in which the variable is defined.             | Global                                  |
| What is the life time?        | Local to the block in which the variable is defined. | Local to the block in which the variable is defined. | Value of the variable persists between different function calls. | As long as the program comes to an end. |

### 9.8.1 Automatic Storage Class

Program 9.21 shows how an automatic storage class variable is declared, and the fact that if the variable is not initialized it contains a garbage value.

#### PROGRAM 9.21

```

1. #include<stdio.h>
2. void main()
3. {
4. auto int x;
5. printf ("\n%d", x) ;
6. }
```

The output of Program 9.21 could be:  
19152

where 19152 is a garbage value of x. When you run this program you may get different values, since garbage values are unpredictable. So always make it a point to initialize the automatic variables properly, otherwise you are likely to get unexpected results. Note that the keyword for this storage class is auto, and not automatic.

The scope and life of an automatic variable is illustrated in Program 9.22.

**PROGRAM 9.22**

```
1. #include<stdio.h>
2. void main()
3. {
4. auto int x=18;
5. {
6. auto int x=8;
7. printf("\nInner block x= %d",x);
8. }
9. printf("\nOuter block x= %d",x);
10. }
```

*Output:*

```
Inner block x = 8
Outer block x = 18
```

Note that the compiler treats the two *x*'s as totally different variables, since they are defined in different blocks. Once control comes out of the innermost block the variable *x* with value 8 is lost, and hence the *x* in the second `printf()` refers to *x* with value 18.

### 9.8.2 Register Storage Class

A value stored in a CPU register can always be accessed faster than the one that is stored in memory. Therefore, if a variable is used at many places in a program it is better to declare its storage class as a register.

Point to be remembered:

- If we declare a variable as a register variable, we cannot say for sure that the value of the variable would be stored in a CPU register, because the number of CPU registers are limited, and they may be busy doing some other task. In such an event the variable works as if its storage class is `auto`.
- Declaring a variable as a register storage class is a request to the compiler but not a command.
- Every type of variable cannot be stored in a CPU register, because if the microprocessor has 16-bit registers then they cannot hold a float value or a double value, which require 4 and 8 bytes respectively.

### 9.8.3 Static Storage Class

Compare the Program 9.23a and 9.23b. Their output shows the difference between the automatic and static storage classes.

**PROGRAM 9.23 (a)**

```

1. #include<stdio.h>
2. void increment()
3. {
4. auto int x=1;
5. printf("%d ", x);
6. x=x+1;
7. }
8. void main()
9. {
10. increment();
11. increment();
12. increment();
13. }
```

Output

1 1 1

**PROGRAM 9.23 (b)**

```

1. #include<stdio.h>
2. void increment()
3. {
4. static int x=1;
5. printf("%d ", x);
6. x=x+1;
7. }
8. void main()
9. {
10. increment();
11. increment();
12. increment();
13. }
```

Output

1 2 3

Like auto variables, static variables are also local to the block in which they are declared. The difference between them is that static variables don't disappear when the function is no longer active. Their values persist. If control comes back to the same function again the static variables have the same values they had last time around.

In the above example, when variable *x* is auto, each time `increment()` is called it is reinitialized to 1 (one). When the function terminates, *x* vanishes and its new value of 2 is lost. The result: no matter how many times we call `increment()`, *x* is initialized to 1 every time.

On the other hand, if *x* is static, it is initialized to 1 only once. It is never initialized again. During the first call to `increment()`, *x* is incremented to 2. Because *x* is static, this value persists. The next time `increment()` is called, *x* is not reinitialized to 1; on the contrary its old value 2 is still available. This current value of *x* (i.e., 2) gets printed and then `x = x + 1` adds 1 to *x* to get a value of 3. When `increment()` is called the third time, the current value of *x* (i.e., 3) gets printed and once again *x* is incremented. In short, if the storage class is static then the statement `static int x = 1` is executed only once, irrespective of how many times the same function is called.

### 9.8.4 External Storage Class

External variables differ from those we have already discussed in that their scope is global, not local. External variables are declared outside all functions. Any function can change the value of the global variable and the updated value of the variable is accessed by other functions. Analyze Program 9.24a and 9.24b to illustrate this fact.

**PROGRAM 9.24 (a)**

```

1. #include<stdio.h>
2. int x=5;
3. void change()
4. {
5. x=x+7;
6. printf("%d ",x);
7. }
8. void main()
9. {
10. printf("%d ",x);
11. change();
12. }
```

**Output**

5 12

**PROGRAM 9.24 (b)**

```

1. #include<stdio.h>
2. int x=5;
3. void change()
4. {
5. x=x+7;
6. printf("%d ",x);
7. }
8. void main()
9. {
10. change();
11. printf("%d ",x);
12. }
```

**Output**

12 12

In the first program, the `main()` function contains two statements. At first the `printf()` function gets executed and searches for the value of `x` to print. As the `main()` function does not have a local `x`, so it accesses the global `x` value for printing and prints 5. After executing the first statement control executes the `change()` function. The `change()` function also has no local `x`, so it accesses the global `x` value and increments it by 7. After incrementing the global `x`, its value is 12. So, the output is 12.

The second program also contains two statement inside the `main()` function, but the output differs. Because the `change()` function calls first, so the value of global `x` is changed to 12. So, the output will be 12. After executing the `change()` function, control returns to the `main()` function and then executes the `printf()` statement. As the value of `x` has already been modified by the `change()` function, so the output of this statement becomes 12.

Look at Program 9.25:

**PROGRAM 9.25**

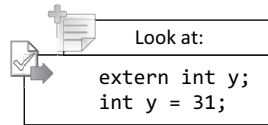
```

1. #include<stdio.h>
2. int x = 21;
3. void main()
4. {
5. extern int y ;
6. printf ("\n%d %d", x, y) ;
7. }
8. int y = 31;
```

**Output:**

21 31

Here, `x` and `y` both are global variables. Since both of them have been defined outside all the functions, both enjoy an external storage class. Note the difference between the following:



Here the first statement is a declaration, whereas the second is the definition. When we declare a variable, no space is reserved for it, whereas when we define it, space gets reserved for it in memory. We had to declare `y` since it is being used in `printf()` before its definition is encountered. There was no need to declare `x` since its definition was made before its usage.

## 9.9 Review Questions

### 9.9.1 Objective Questions

1. We can divide a bigger problem into smaller problems, and all smaller problems can be developed individually and used whenever necessary. This description refers to \_\_\_\_\_ programming concept.
2. The C programming language supports two types of function: \_\_\_\_\_ and \_\_\_\_\_.
3. \_\_\_\_\_ can be defined as a group of statements enclosed within a block with a valid identifier and can perform a specific task.
4. A function can return multiple values simultaneously. True/false?
5. If your function does not return anything, then the return type will be \_\_\_\_\_.
6. Blank parentheses after the function name specifies \_\_\_\_\_.
7. The function which calls other functions is known as \_\_\_\_\_.
8. The function which is called by other functions is known as \_\_\_\_\_.
9. \_\_\_\_\_ is called the declaration of a function.
10. The prototype declaration can be eliminated by \_\_\_\_\_.
11. The arguments which are specified in the function call are known as \_\_\_\_\_ arguments.
12. The parameters specified in the function definition are known as \_\_\_\_\_ parameters.
13. If a variable is declared as static, what is its initial value?
14. A variable declared as extern has initial value \_\_\_\_\_.
15. If you want to allocate space for a variable inside the CPU register, what storage class will you use to declare it?

16. With \_\_\_\_\_, \_\_\_\_\_, and \_\_\_\_\_ storage classes, the scope of a variable is local to the block in which it is defined.
17. With \_\_\_\_\_ storage class, the scope of a variable is global in nature.
18. With \_\_\_\_\_ storage class, the lifetime of a variable persists between different function calls.
19. With \_\_\_\_\_ storage class, the lifetime of a variable exists until the program comes to an end.

### 9.9.2 Subjective Questions

1. Define a function. What is the syntax of the user-defined function, and explain all its components?
2. Why do we need functions? State at least three advantages of using functions.
3. Write short notes on (1) predefined functions and (2) user-defined functions.
4. What is the purpose of the return statement?
5. What is the difference and relationship between formal arguments and actual arguments?
6. Can the names of the formal arguments within a function coincide with the names of other variables defined outside of the function? Explain.
7. What is the purpose of the keyword void? Where is this keyword used?
8. What are function prototypes? What is their purpose? Where within a program are function prototypes normally placed?
9. How do functions help in reducing the size of a program? Explain with an example.
10. What is recursion? What advantage is there in its use?
11. What is a storage class? List all the storage classes used in the C programming language.
12. Explain how the storage classes are differentiated from each other in terms of storage, initial value, scope, and lifetime.

### 9.9.3 Programming Questions

1. Write a function to calculate  $x^y$  using an iterative method and a recursive method.
2. Any year is entered through the keyboard. Write a function to determine whether the year is a leap year or not.
3. Write a user-defined function that receives a float and an int from the `main()` function. Your user defined function should calculate their product and return it to the `main()` function. Finally, the `main()` functions receive the result and print it.
4. Write a function that receives five integers and returns the average of these numbers. Call this function from `main()` and print the results in `main()`.
5. Write a function to find out the largest of three numbers.
6. Write a function to add the digits of a number.
7. Write a C program to obtain the greatest common divisor (GCD) of two integers using iterative and recursive methods.

8. Write a function to generate the Fibonacci series.

9. Find the output of the following programs:

```
#include<stdio.h>
void main()
{
 int x=5;
 {
 int x=7;
 printf("%d",x);
 }
 printf("%d",x);
}
```

```
#include<stdio.h>
int x=7;
void main()
{
 {
 int x=5;
 printf("%d",x);
 }
 printf("%d",x);
}
```

```
#include<stdio.h>
int x=25;
void test(int x)
{
 x=x+1;
 printf("%d",x);
}
void main()
{
 test(x);
 printf("%d",x);
}
```

```
#include<stdio.h>
void test(int x)
{
 printf("%d",x);
}
void main()
{
 int x=5;
 test(++x);
 printf("%d",x);
}
```

```
#include<stdio.h>
void main()
{
 int x=5;
 {
 printf("%d",x);
 }
 printf("%d",x);
}
```

```
#include<stdio.h>
void test(int x)
{
 x=x+1;
 printf("%d",x);
}
void main()
{
 int x=5;
 test(x);
 printf("%d",x);
}
```

```
#include<stdio.h>
void test(int x)
{
 printf("%d",x);
}
void main()
{
 int x=5;
 test(x++);
 printf("%d",x);
}
```

```
#include<stdio.h>
int test(int x)
{
 return x++;
}
void main()
{
 int x=5,y;
 y=test(x);
 printf("%d",y);
}
```



```
#include<stdio.h>
int test(int x)
{
 return ++x;
}
void main()
{
 int x=5,y;
 y=test(x);
 printf("%d",y);
}
```

```
#include<stdio.h>
int test(int x)
{
 return ++x + ++x;
}
void main()
{
 int x=5,y;
 y=test(x);
 printf("%d",y);
}
```

# 10

## *Arrays and Strings*

### 10.1 Introduction

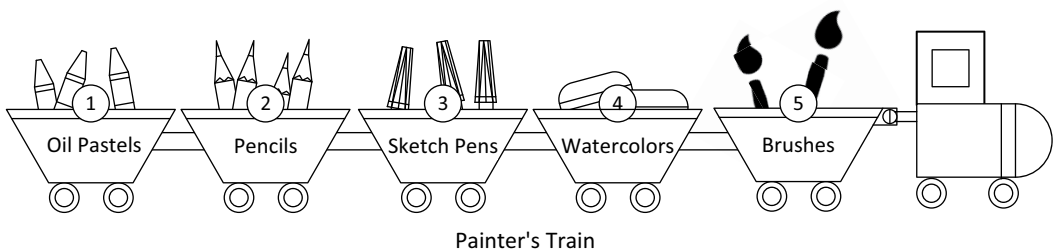
Many times, we come across a situation where we use a set of data rather than a single datum. For example, assume that you are an instructor and you teach C programming. You want to store the marks of your students and later perform different types of operations on them, such as finding the top performer or knowing how many students score less than 50 marks. In that case usage of a single variable is not enough: you need multiple variables to store the data of your students. Again, if you use many variables in your program then remembering those variable names will become difficult. So, the solution is the array – a concept provided by C that handles large numbers of items simultaneously.

In our day-to-day life we also came across situations where we need to group items and keep them in a sequential manner for easy access. For example, Figure 10.1 shows a toy train built to store painting items such as watercolors, sketch pens, brushes, pencils, and oil pastels. We name this train the Painter's Train. We number the boxes from 1 to 5 and store many items in them. The concept of arranging similar data and calling them using a common name is sometimes known as an array of items.

An array is a series of elements of the same type, placed in contiguous memory locations that can be individually referenced by adding an index number to each location.

Other definitions are:

- An array is a single programming variable with multiple "compartments." Each compartment can hold a value.
- A collection of data items, all of the same type, in which the position of each item is uniquely designated by a discrete type.



**FIGURE 10.1**  
Introducing the concept of an array.

This chapter is dedicated to a discussion of arrays. After completion of this chapter, readers will have learnt the following:

- How to define an array and its type;
- How to write code to declare different types of arrays, like 1D arrays and 2D arrays, and use them for solving problems;
- Declaration, processing, and manipulation of strings (also known as character arrays).

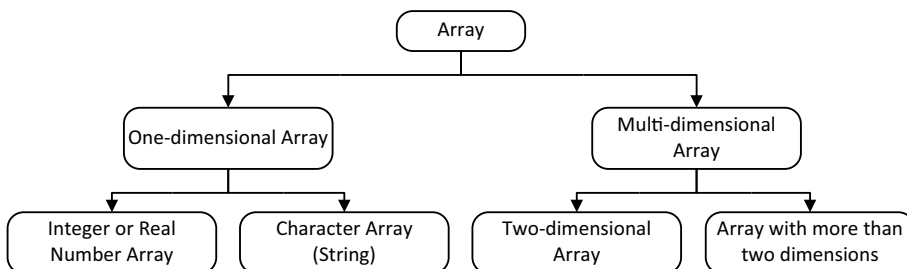
## 10.2 Need for Arrays

A variable can hold one value at a time. There are situations in which we want to store more than one value at a time. Suppose we want to store the age of 100 persons and arrange them in a sorted manner. This type of problem can be solved by taking 100 variables, where each variable contains the age of a single person. But this is an inconvenient way to solve this problem. A more elegant way is to use an *array*. So, before we solve different types of problems, we should know the basic concept of an *array*.

## 10.3 Types of Arrays

According to the definition, an array is a collection of compartments. Further, we can say the elements of an array can be stored in a row of compartments, or can be stored in the form of a table. Depending upon the representation style and the type of data it contains, the array can be classified as shown in Figure 10.2.

A character array (string) is of the type 1D array (i.e., the array which contains only character data).



**FIGURE 10.2**  
Classification of arrays.

### 10.4 1D Arrays

A 1D array is a collection of items having common data types stored in a contiguous memory location, identified by a single name with its index number. More generally, suppose we have stored five different integers in a continuous memory location with a common name. We can access each integer with the help of the name of the array and index number – then the whole scenario is called an integer array.

A one-dimensional array is a collection of items having common data types stored in a contiguous memory location, identified by a single name with its index number.

#### 10.4.1 Declaration of 1D Arrays

As we know, before using any variable, we must declare it. Similarly, we must declare the array before we use it in our program code. Let us recall how a variable is declared because there is a little difference between the declaration of a variable and the declaration of an array (see Figure 10.3).

In this example, x represents the name of the variable, and int represents the type of data that x can store and the size (in bytes) of x (see Figure 10.4).

In the same manner, we can declare an array. Figure 10.5 shows the syntax.

- The data type indicates the type of data we can store inside an array;
- Array names can be valid identifiers;

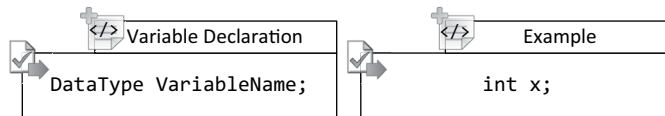


FIGURE 10.3 Variable declaration with example.

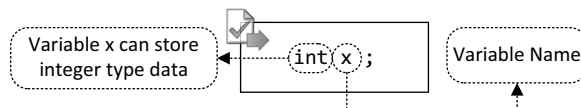


FIGURE 10.4 Describing int x.

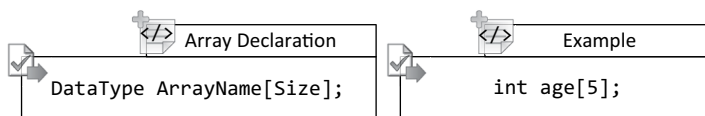
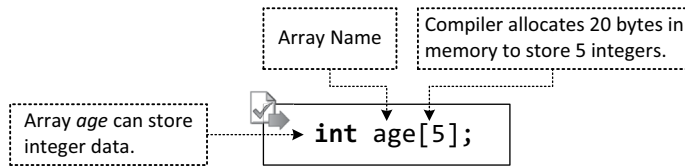


FIGURE 10.5 Array declaration with example.



**FIGURE 10.6**  
Illustration of the example shown in Figure 10.5.

- Size (also known as the dimension) indicates the number of the same type of data we can store inside the array;
- The size must be an integer and specified within two square brackets.
- See Figure 10.6 for an illustration of the declaration process. This indicates the array named age will contain five integer values;
- This declaration will immediately reserve 20 bytes of memory because each of these five integers will be four bytes long (we assume the integer takes four bytes for our illustration).

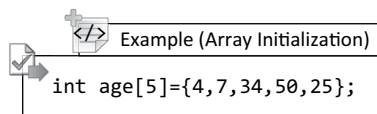
**NOTE**

The dimension of the array must be a constant. We cannot use a variable name for representing the dimension of the array. This is possible, but for the time being, we will not concentrate on this aspect, as it will be discussed later whenever there is a need.

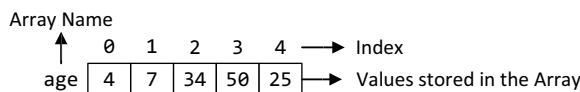
**10.4.2 Initialization of Arrays**

Once we declare an array, we can initialize it with some value. The initialization is the same as that for a general variable. As the array contains more than one element, each one is initialized in a specific order, between the curly braces and separated by commas (Figure 10.7).

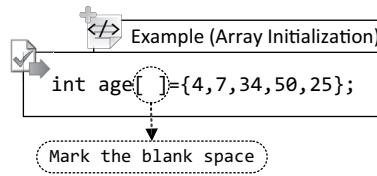
Conceptually the array can be viewed as shown in Figure 10.8.



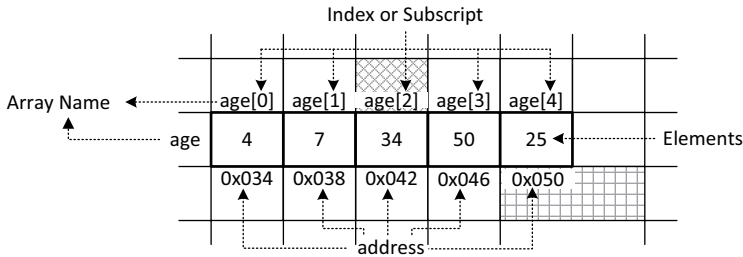
**FIGURE 10.7**  
Array initialization.



**FIGURE 10.8**  
Conceptual representation of an array.



**FIGURE 10.9**  
Another way to initialize an array.



**FIGURE 10.10**  
Representation of an array in memory.

- The initialization can also take the form shown in Figure 10.9.
- The size (represented in the empty bracket) can be assumed to be equal to the number of elements in the array (five in this case).


The array can be represented in memory as shown in Figure 10.10. The figure shows some of the memory locations where five blocks with addresses 0x034 through 0x050 are allocated for storing the elements. The shaded regions represent previously allocated blocks. The blocks with bold lines represent the newly allocated block for our declaration. All the newly allocated blocks are referred to by a single name *age*.

- Each element in the array can be referred to by using an index or subscript (the number in the bracket following the array name) and the array name. For example *age*[0] represents the first element, *age*[1] represents the second element, and so on.
- All elements in the array are numbered starting with 0. So, for example, *age*[2] represents the third element in the array.
- If the starting address of the array is 0x034 then the next element will start from 0x038 because the integer element takes four bytes.

### 10.4.3 Accessing Array Elements


To access a particular element in an array, specify the array name, followed by square braces enclosing an integer, which is called the index or subscript. An array is also called a *subscripted variable* because to access each element, we need a subscript or index number.

- The index will vary from 0 to size-1. Size indicates the total number of elements present in an array. For the above array declaration, if we write the following C code statement, it will print 50.



```
printf("%d", age[3]);
```

- We can initialize the elements at index 0 and 3 of the array `age` using the following C code statements.



```
age[0]=4;
age[3]=50;
```

#### 10.4.4 Characteristics of an Array

1. An array is a collection of similar types of elements, which can be int, float, or char.
2. The first element in the array is numbered 0, so the last element is 1 less than the size of the array.
3. An array is also known as subscripted variables.
4. Before using an array, its type and dimension must be declared.
5. However big an array, its elements are always stored in contiguous memory locations.
6. If array variables are not assigned with any value, then they will contain garbage values.
7. Usually, an array of characters is called a "string."
8. The array name represents the address of an array if it is not associated with any subscripts. Thus the array name `age` is equivalent to `&age[0]`.

#### PROGRAM 10.1

```
1. #include<stdio.h>
2. void main()
3. {
4. int age[3]={4,5,6};
5. printf("%u",&age[0]);
6. printf("\n%u", age);
7. }
```

#### Output:

The program will print two addresses as follows:

```
235383876
235383876
```

### 10.4.5 Entering Data in an Array

Let us first discuss how to enter data for a single element. For entering data to a single variable, we can write the code as shown in Program 10.2.

#### PROGRAM 10.2

```
1. #include<stdio.h>
2. void main()
3. {
4. int x;
5. printf("Enter the value:");
6. scanf("%d",&x);
7. }
```

*Output:*

Enter the value: 25

For entering data in an array, we can perform the same operation. But here we have to use `scanf()` for each element because the array does not contain a single element. That indicates that the number of `scanf()` functions used is equivalent to the number of elements present in the array. Program 10.3 shows the code for this purpose.

#### PROGRAM 10.3

```
1. #include<stdio.h>
2. void main()
3. {
4. int age[5];
5. printf("Enter the element for the array:");
6. scanf("%d",&age[0]);
7. scanf("%d",&age[1]);
8. scanf("%d",&age[2]);
9. scanf("%d",&age[3]);
10. scanf("%d",&age[4]);
11. }
```

*Output:*

Enter the element for the array:

As the number of elements increases in an array, the above code will be increase with `scanf()` statements. Hence, instead of writing a `scanf()` function for every element, we can use a for loop to write it as shown in Program 10.4. The for loop in this code helps in executing the `scanf()` function repeatedly for each input element.

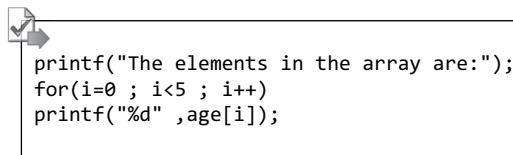


**PROGRAM 10.4**

```
1. #include<stdio.h>
2. void main()
3. {
4. int age[5] , i ;
5. printf("Enter the element for the array:");
6. for(i=0 ; i<5 ; i++)
7. scanf("%d",&age[i]);
8. }
```

**10.4.6 Displaying the Content of an Array**

For displaying all the elements of an array either we can use the `printf()` function for each element, or we can use a `for` loop that repeats a `printf()` function for each outputted element. The C code snippet that uses a `for` loop is shown below:



```
printf("The elements in the array are:");
for(i=0 ; i<5 ; i++)
printf("%d" ,age[i]);
```

As shown in the code, the `printf()` function will execute five times and print the element present in the array `age[5]`.

Now, let us write a complete program that will declare an array of five elements, ask the user to enter five elements into it, and finally display the elements present. Program 10.5 shows the complete program.

**PROGRAM 10.5**

```
1. #include<stdio.h>
2. void main()
3. {
4. int age[5] , i ;
5. printf("Enter the element for the array:");
6. for(i=0 ; i<5 ; i++)
7. scanf("%d",&age[i]);
8. printf("The elements in the array are:");
9. for(i=0 ; i<5 ; i++)
10. printf("%d " ,age[i]);
11. }
```

*Output:*

```
Enter the element for the array: 24 34 65 21 15
The elements in the array are: 24 34 65 21 15
```

The user may not know the size of the array. So, when you are using the concept of an array it is recommended that you should take an array with a maximum size and use some programming techniques to get input from the user for a specific problem. To employ this concept, the Program 10.5 needs to be modified as follows:

#### PROGRAM 10.6

```
1. #include<stdio.h>
2. void main()
3. {
4. int age[50] , i, n;
5. printf("Enter how many number you want: ");
6. scanf("%d", &n);
7.
8. printf("Enter %d elements for the array: ", n);
9. for(i=0 ; i<n ; i++)
10. scanf("%d",&age[i]);
11.
12. printf("The elements in the array are:");
13. for(i=0 ; i<n ; i++)
14. printf("%d " ,age[i]);
15. }
```

*Output:*

```
Enter how many elements you want: 5
Enter 5 elements for the array: 23 56 32 65 78
The elements in the array are: 23 56 32 65 78
```

### 10.4.7 Programming Examples

#### 10.4.7.1 Write a Program to Create an Array of N Elements and Write the Code to Find the Biggest Number and the Smallest Number Present in the Array

#### PROGRAM 10.7

```
1. #include<stdio.h>
2. void main()
3. {
4. int arr[100];
5. int large,small,i,N;
6. printf("Enter the number of elements: ");
7. scanf("%d",&N);
8. printf("Enter elements of the array\n");
```

```

9. for(i=0;i<N;i++)
10. {
11. scanf("%d",&arr[i]);
12. }
13. large=arr[0];
14. small=arr[0];
15.
16. for(i=0;i<N;i++)
17. {
18. if(arr[i]>large)
19. large=arr[i];
20. }
21. for(i=0;i<N;i++)
22. {
23. if(arr[i]<small)
24. small=arr[i];
25. }
26. printf("The largest number=%d", large);
27. printf("\nThe smallest number=%d", small);
28. }

```

*Output:*

```

Enter the number of elements: 5
Enter elements of the array
25 87 45 32 98
The largest number = 98
The smallest number = 25

```

#### **10.4.7.2 Write a Program to Search for an Element Present in the Array, the Number of Times the Element is Present, and Print the Element's Positions**

##### **PROGRAM 10.8**

```

1. #include<stdio.h>
2. void main()
3. {
4. int a[100],i,n;
5. int ele,count=0;
6. printf("Enter how many elements you want: ");
7. scanf("%d",&n);
8. printf("Enter the array elements\n");
9. for (i=0; i<n; i++)

```

```
10. scanf("%d",&a[i]);
11.
12. printf("Enter the element to be search: ");
13. scanf("%d",&ele);
14. for (i=0; i<n; i++)
15. {
16. if(a[i]==ele)
17. {
18. printf("Element %d found in position=%d\n",ele,i+1);
19. count=count+1;
20. }
21. }
22. if (count==0)
23. printf("\nElement not found\n");
24. else
25. printf("\nThe element is found %d times", count);
26. }
```

*Output:*

Run 1

Enter how many elements you want: 5

Enter the array elements

12 23 12 34 23

Enter the element to be search: 23

Element 23 found in position=2

Element 23 found in position=5

The element is found 2 times

Run 2

Enter how many elements you want: 5

Enter the array elements

23 76 98 66 44

Enter the element to be search: 12

Element not found

**10.4.7.3 Write a Program to Print the Binary Equivalent of a Decimal Number Using an Array**

**PROGRAM 10.9**

```
1. #include<stdio.h>
2. void main()
3. {
4. int binary[20], i , m, n, r;
```

```

5. printf("Enter the number.....");
6. scanf("%d",&n);
7. m=n;
8. for(i=0 ; n>0 ; i++)
9. {
10. r=n%2;
11. binary[i]=r;
12. n=n/2;
13. }
14. printf("Binary equivalent of %d is : ", m);
15. i=i-1;
16. for(; i>=0 ; i--)
17. printf("%d ", binary[i]);
18. }

```

*Output:*

Run 1

Enter the number.....5

Binary equivalent of 5 is : 1 0 1

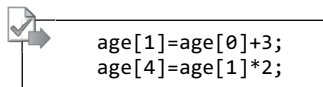
Run 2

Enter the number.....45

Binary equivalent of 45 is : 1 0 1 1 0 1

#### 10.4.8 Points to Note

- We can use a subscripted variable anywhere as a normal variable for writing C statements. For example:



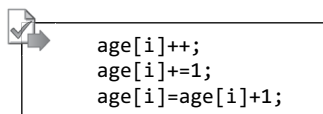
```

age[1]=age[0]+3;
age[4]=age[1]*2;

```

The above statements are perfectly valid and do not show any errors.

- For incrementing the *i*th element of a given array the following statement can also be used:

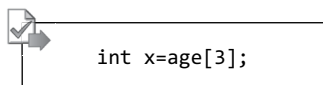


```

age[i]++;
age[i]+=1;
age[i]=age[i]+1;

```

- An array element can be assigned to a general variable as follows:



```

int x=age[3];

```

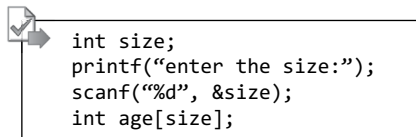
- An array has no bound checking capability, i.e., if we declare an array of size 3 and try to print the 4th element (`age[4]`), then the latter does not show any error and prints a garbage value. Program 10.10 shows the C code for clarity. The output of the program will print a garbage.

#### PROGRAM 10.10

```

1. #include<stdio.h>
2. void main()
3. {
4. int x[]={1,3,5};
5. printf("%d",x[4]);
6. }
```

- We must mention the dimension of the array during its declaration. It is impossible to declare an array using a variable. So, the following code does not work:



```

int size;
printf("enter the size:");
scanf("%d", &size);
int age[size];
```

## 10.5 2D Arrays

In the previous section, we discussed the concept of 1D arrays. We can write `int x[5]`; where 5 is the dimension. But in real life, we may come across a situation where we need to represent the data in a tabular format (with rows and columns). In mathematics, we use the concept of a matrix, where we represent the elements in rows and columns. So here we need a 2D array (e.g., `int X[3][4]`, where 3 and 4 indicate the dimension) to represent a matrix. An array may take three dimensions or more. We can also say that when an array takes more than a one-dimensional form, it is called a *multidimensional array*. In this section, we will discuss the 2D array.

In the 2D array, we require two subscripts to indicate one element. One subscript denotes the row number and the other subscript denotes the column number. Like 1D arrays, the subscript value starts from 0 (zero). Figure 10.11 shows an example to denote an element in a 2D array.

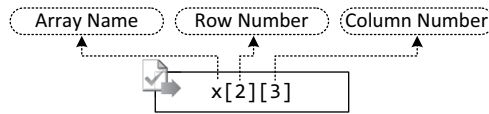
### 10.5.1 Introducing Matrices

A matrix  $mat[m][n]$  is an  $m$  by  $n$  table having  $m$  rows and  $n$  columns containing  $m \times n$  elements (refer to Figure 10.12). Each element of the matrix is represented by  $mat[i][j]$ .

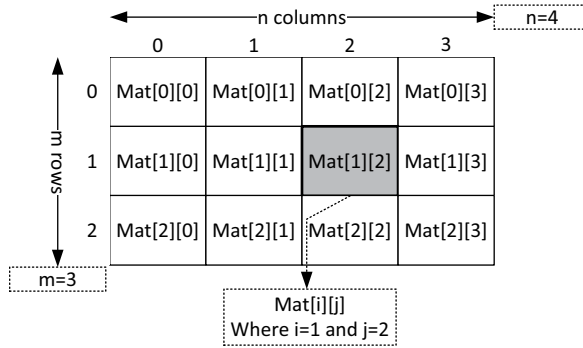
where,

$i$  represents the row number and varies from  $i = 0, 1, 2, \dots, m - 1$

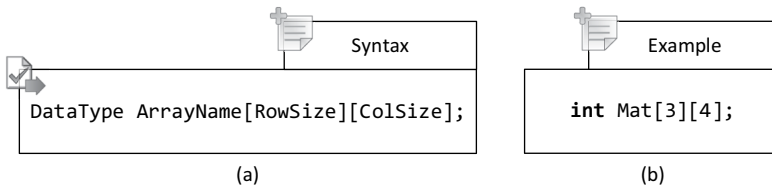
$j$  represents the column number and varies from  $j = 0, 1, 2, \dots, n - 1$



**FIGURE 10.11**  
An element of a 2D array with two subscripts.



**FIGURE 10.12**  
A matrix.



**FIGURE 10.13**  
Syntax of a 2D array declaration and example.

### 10.5.2 Declaration of a 2D Array

We use a general form shown in (Figure 10.13a) to declare a 2D array.

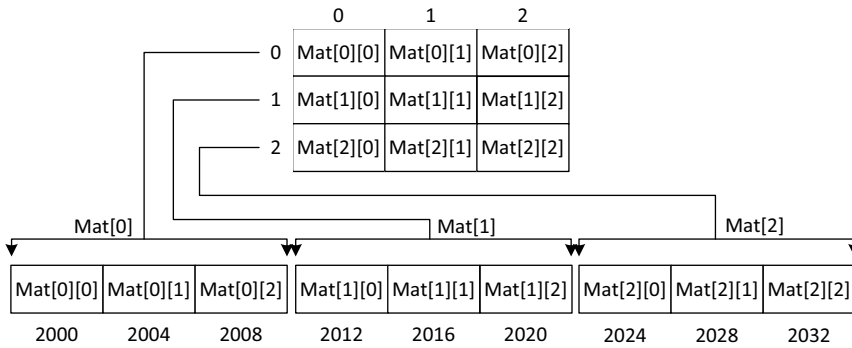
The example code (Figure 10.13b) will create an array of integers named `Mat` and can store  $4 \times 3 = 12$  elements of integer type. The element of the array can be accessed by `Mat[0][0]`, `Mat[0][1]`, . . . . `Mat[2][3]` and so on. According to its characteristics, all the elements of the matrix will be stored in contiguous memory locations.

### 10.5.3 Representation of a 2D Array in Memory

Since we know that array elements will be stored in contiguous memory locations, the 2D array is also stored in contiguous memory locations. In memory, whether a 1D or a 2D array, the array elements are stored in one continuous chain.

There are two ways of representing a 2D array inside memory:

1. Row major order;
2. Column major order.



**FIGURE 10.14** Row major order representation of a 2D array in memory.

### 10.5.3.1 Row Major Order

In row major order the first row occupies the first set of memory locations, the second occupies the next set, and so on (see Figure 10.14).

The address of an element in the above 2D array can be calculated by the following formula:

$$Address(A[i][j]) = Base + W(C \times i + j)$$

where,

- W = the size of each element;
- Base = the base address of the array;
- C = the number of columns present in the array.

**Quiz:** Assume that a 3 × 3 2D array is represented in row major order and stores integers. Let the integers take four bytes in memory. The base address is 2000. Find the address of A[1][2].

**Answer:**

$$Address(A[i][j]) = Base + W(C \times i + j)$$

$$Address(A[1][2]) = 2000 + 4(3 \times 1 + 2)$$

$$Address(A[1][2]) = 2000 + 20 = 2020$$

### 10.5.3.2 Column Major Order

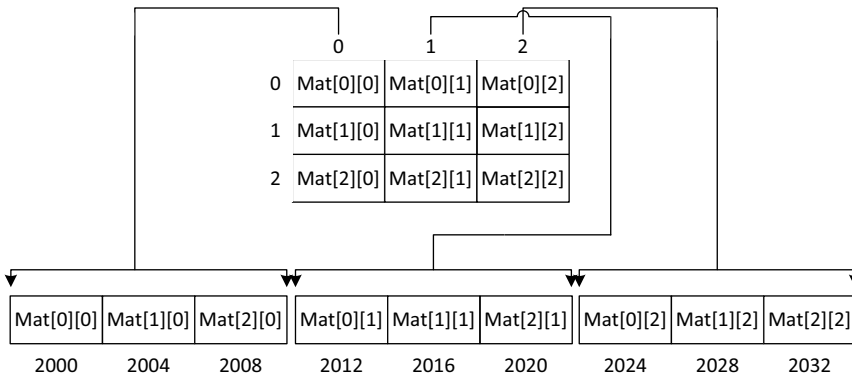
Here the first column of the array occupies the first set of memory locations, the second occupies the second set, and so on (see Figure 10.15).

The address of an element in the above 2D array can be calculated by the following formula:

$$Address(A[i][j]) = Base + W(R \times j + i)$$

where,





**FIGURE 10.15**  
Column major order representation of a 2D array in memory.

- W = the size of each element;
- Base = the base address of the array;
- R = the number of rows present in the array.

**Quiz:** Assume that a 3 × 3 2D array is represented in column major order and stores integers. The base address is 2000. Find the address of A[1][2].

**Answer:**

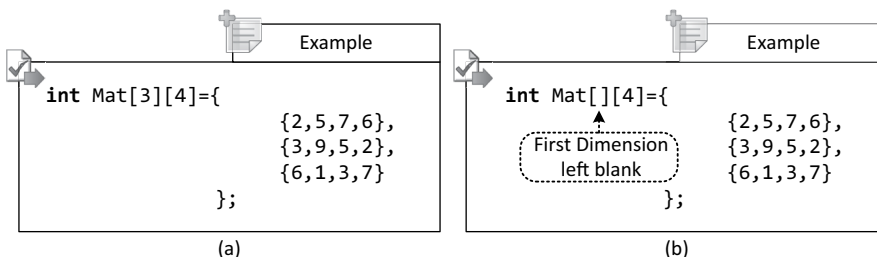
$$Address(A[i][j]) = Base + W(C \times i + j)$$

$$Address(A[1][2]) = 2000 + 4(3 \times 2 + 1)$$

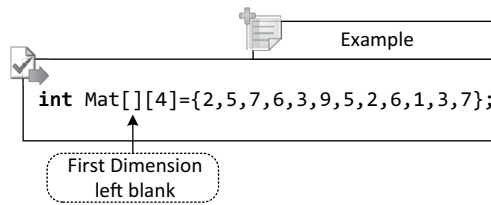
$$Address(A[1][2]) = 2000 + 28 = 2028$$

### 10.5.4 Initialization of a 2D Array

Initialization of a 2D array during declaration is done by specifying the elements in *row major order*. (The element of the first row is entered in a sequence, followed by the second row, the third row, and so on.) Figure 10.16a specifies an example. We may not specify the first dimension, but the second dimension is compulsory. One such example is shown in Figure 10.16b, where the compiler automatically reads four elements, treats them as row 1, then the second set of four elements, treats them as row 2, and so on.



**FIGURE 10.16**  
Example of 2D array initialization.

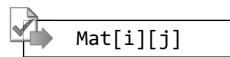
**FIGURE 10.17**

Example of 2D array initialization without dividing elements into groups.

If we want to initialize a 2D array using the rule followed in Figure 10.16b, then we can use a simpler way to write it. We can omit the inner bracket and provide the element in a continuous sequence (Figure 10.17 shows one such example).

### 10.5.5 Accessing the Elements of a 2D Array

The elements of a 2D array can be accessed in the same manner as we access a matrix, that is, we can access each element by providing the array name, the row position, and the column position, as follows:



where *i* refers to the row number and *j* refers to the column number.

Now that we understand the declaration and initialization of a 2D array, let us write a simple program that declares a 2D array, initializes it, and displays the array in a matrix format.

#### PROGRAM 10.11

```

1. #include<stdio.h>
2. void main()
3. {
4. int i,j;
5. int Mat[3][4]= {
6. {2, 5, 7, 6},
7. {3, 9, 5, 2},
8. {6, 1, 3, 7}
9. };
10. printf("%d ",Mat[0][0]);
11. printf("%d ",Mat[0][1]);
12. printf("%d ",Mat[0][2]);
13. printf("%d ",Mat[0][3]);

```

```

14. printf("\n");
15. printf("%d ",Mat [1] [0]);
16. printf("%d ",Mat [1] [1]);
17. printf("%d ",Mat [1] [2]);
18. printf("%d ",Mat [1] [3]);
19. printf("\n");
20. printf("%d ",Mat [2] [0]);
21. printf("%d ",Mat [2] [1]);
22. printf("%d ",Mat [2] [2]);
23. printf("%d ",Mat [2] [3]);
24. }

```

*Output:*

```

2576
3952
6137

```

In Program 10.11, for displaying 12 elements, we have used 12 `printf()` statements; but it can be reduced by using a for loop. We also have used two extra `printf()` statements at lines 14 and 19 that will help the array to be displayed like a matrix. If we remove these two `printf()` statements, then the elements will be printed in a row. Program 10.12 shows how to write the same program using a for loop.

#### PROGRAM 10.12

```

1. #include<stdio.h>
2. void main()
3. {
4. int i,j;
5. int Mat [3] [4]= {
6. {2, 5, 7, 6},
7. {3, 9, 5, 2},
8. {6, 1, 3, 7}
9. };
10. printf("The matrix is:\n");
11. for(i=0;i<3;i++)
12. {
13. for(j=0;j<4;j++)
14. {
15. printf("%d", Mat [i] [j]);
16. }

```

```

17. printf("\n");
18. }
19. }

```

*Output:*

The matrix is:

```

2 5 7 6
3 9 5 2
6 1 3 7

```

*Explanation:*

The inner for loop (lines 13 to 16) helps to print four elements in a row; line 17 will send the cursor to the next line. The outer for loop (line 11) helps the inner for loop to execute three times, because the matrix has three rows.

### 10.5.6 Entering Data in a 2D Array

We can initialize the 2D array, or we can enter the value at run time. For entering the value at run time, we may use `scanf()` for each element, or we may use a for loop. Both formats are given below.

Using `scanf()` for each input will increase the program size and it is not convenient.

```

1. int Mat[3][4];
2. printf("Enter the 0,0 element");
3. scanf("%d", &Mat[0][0]);
4. printf("Enter the 0,1 element");
 scanf("%d", &Mat[0][1]);
5. .
6. .
7. .
8. printf("Enter the 2,3 element");
9. scanf("%d", &Mat[0][1]);

```

Using the for loop we can reduce the code drastically; So, this is **recommended**.

```

1. int i,j;
2. int Mat[3][4];
3. printf("Enter the element");
4. for(i=0;i<3;i++)
5. {
6. for(j=0;j<4;j++)
7. {
8. scanf("%d",&Mat[i][j]);
9. }
10. }

```

*Explanation:*

Refer to the **recommended** code for this explanation. We use the concept of the nested for loop to read the input from the user. The `scanf()` function (line 8) reads the input and stores it in the corresponding row and column number, specified by the values of `i` and `j`. The inner for loop (line 6) executes the `scanf()` four times because our matrix contain four elements in each row. The outer for loop (line 4) will execute the inner for loop three times, because we have three rows in our matrix.

Let us write a complete program (Program 10.13) to enter some elements in a 2D array and print it in matrix format.

**PROGRAM 10.13**

```

1. #include<stdio.h>
2. void main()
3. {
4. int i,j;
5. int Mat[3][4];
6. printf("Enter the element: ");
7. for(i=0;i<3;i++)
8. {
9. for(j=0;j<4;j++)
10. {
11. scanf("%d",&Mat[i][j]);
12. }
13. }
14.
15. printf("The matrix is:\n");
16. for(i=0;i<3;i++)
17. {
18. for(j=0;j<4;j++)
19. {
20. printf("%d ",Mat[i][j]);
21. }
22. printf("\n");
23. }
24. }

```

*Output:*

Enter the element: 1 2 3 4 5 6 7 8 9 1 2 3

The matrix is:

```

1 2 3 4
5 6 7 8
9 1 2 3

```

The user may not always know how much row and column size is required for each run of the program. Till now, whatever program we have written, the matrix size was  $3 \times 4$ . But sometimes the user may need more than this size or less. So it is better to declare the matrix size with a bigger number and use a programming technique to get the number of the row and column size from the user at run time. Consider Program 10.14.

**PROGRAM 10.14**

```
1. #include<stdio.h>
2. void main()
3. {
4. int i,j, r, c;
5. int Mat[100][100];
6. printf("Enter the row and column size: ");
7. scanf("%d %d", &r,&c);
8. printf("Enter %d element: ",r*c);
9. for(i=0;i<r;i++)
10. {
11. for(j=0;j<c;j++)
12. {
13. scanf("%d",&Mat[i][j]);
14. }
15. }
16.
17. printf("The matrix is:\n");
18. for(i=0;i<r;i++)
19. {
20. for(j=0;j<c;j++)
21. {
22. printf("%d ",Mat[i][j]);
23. }
24. printf("\n");
25. }
26. }
```

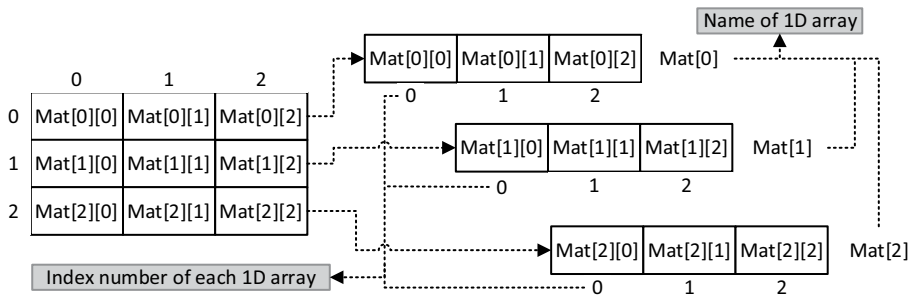
*Output:*

```
Enter the row and column size: 4 5
Enter 20 element: 12 13 14 15 16 23 33 43 56 23 56 78 98 32 23 12 23 45 23 76
The matrix is:

12 13 14 15 16
23 33 43 56 23
56 78 98 32 23
12 23 45 23 76
```

*Explanation:*

We have declared a large-sized matrix of  $100 \times 100$  (line 5). Using lines 6 and 7 we read the row size and column size from the user. After that we asked the user to enter the *rowSize*  $\times$  *columnSize* number of elements using line 8. Lines 9 to 15 help in reading the elements from the user's input and storing it in the allocated memory space. Finally, lines 17 to 25 print the content of the 2D array in matrix format as shown in the output.



**FIGURE 10.18**  
Representing a 2D array in the form of 1D arrays.

### 10.5.7 Exploration of a 2D Matrix

From the previous concept, we see that a 2D array is nothing but a table of data. We can also say that each row of this table is a 1D array (see Figure 10.18). We can give a name to each 1D array. The name can be the array name and the row number (for example we can say `Mat[0]` indicates the name of the first row, which is a 1D array, `Mat[1]` can be the name of the second 1D array, and so on).

To prove the above concept, we are going to use the concept of the 1D array: that the array name indicates the address of the first element of the array. That means if I print the value of `Mat [0]` or `Mat [1]`, then it should give an address. Consider the following program.

#### PROGRAM 10.15

```

1. #include<stdio.h>
2. void main()
3. {
4. int Mat [3] [4]={
5.
6. {2, 5, 7, 6},
7. {3, 9, 5, 2},
8. {6, 1, 3, 7}
9. };
10.
11. printf("\nThe address of 1st row= %u", Mat[0]);
12. printf("\nThe address of 2nd row= %u", Mat[1]);
13. printf("\nThe address of 3rd row= %u", Mat[2]);
14. }
```

*Output:*

```

The address of 1st row= 3738969104
The address of 2nd row= 3738969120
The address of 3rd row= 3738969136
```

*Explanation:*

We are able to print the addresses of each row using the matrix name with its first dimension. So, it is proved that a 2D array is a collection of 1D arrays. The format specifier %u indicates the format string for an unsigned integer. We know that an address never takes a negative value, so we take %u here. If you analyze the output, you can easily observe that the difference between the address of the first row and the second row is 16. Similarly, the second row and the third have the same difference. The matrix in Program 10.15 has 12 elements and each row contains 4 integer elements. Here the compiler takes four bytes to represent each integer element. Hence, the address of each row differs by 16. It also proves that the array elements are stored in a contiguous location in memory.

**10.5.8 Programming Examples****10.5.8.1 Write a Program to Add All the Elements Present in the Main Diagonal of a 2D Matrix****PROGRAM 10.16**

```

1. #include<stdio.h>
2. void main()
3. {
4. int a[10][10],i,j,sum=0,r,c;
5. printf("\nEnter the number of rows: ");
6. scanf("%d",&r);
7. printf("\nEnter the number of columns: ");
8. scanf("%d",&c);
9. if(r==c)
10. {
11. printf("\nEnter the elements of matrix: ");
12. for(i=0;i<r;i++)
13. for(j=0;j<c;j++)
14. scanf("%d",&a[i][j]);
15.
16. printf("\nThe matrix is\n");
17. for(i=0;i<r;i++)
18. {
19. for(j=0;j<c;j++)
20. {
21. printf("%d\t",a[i][j]);
22. }
23. printf("\n");
24. }

```



```

25. for(i=0;i<r;i++)
26. {
27. for(j=0;j<c;j++)
28. {
29. if(i==j)
30. sum=sum+a[i][j];
31. }
32. }
33. printf("\n\nSum of the diagonal elements of a matrix
 is: %d",sum);
34. }
35. else
36. {
37. printf("Column and row size must be the same";
38. }
39. }

```

*Output:*

Run 1

Enter the number of rows: 3

Enter the number of columns: 3

Enter the elements of matrix: 1 2 3 4 5 6 7 8 9

The matrix is

```

1 2 3
4 5 6
7 8 9

```

Sum of the diagonal elements of a matrix is: 15


Run 2

Enter the number of rows: 3

Enter the number of columns: 5

Column and row size must be the same

### 10.5.8.2 Write a Program to Add the Elements of Each Column and Print it in the Following Format

|   | Matrix  |                                                                                     |   |   | Output |    |
|---|---------|-------------------------------------------------------------------------------------|---|---|--------|----|
| 3 | 2     4 |                                                                                     | 3 | 2 | 4      | 9  |
| 7 | 1     3 |  | 7 | 1 | 3      | 11 |
| 6 | 8     7 |                                                                                     | 6 | 8 | 7      | 21 |

**PROGRAM 10.17**

```
1. #include<stdio.h>
2. void main()
3. {
4. int a[10][10],i,j,sum=0,r,c;
5. printf("\nEnter the number of rows: ");
6. scanf("%d",&r);
7. printf("\nEnter the number of columns: ");
8. scanf("%d",&c);
9. printf("\nEnter the elements of matrix: ");
10. for(i=0;i<r;i++)
11. for(j=0;j<c;j++)
12. scanf("%d",&a[i][j]);
13.
14. printf("\nThe matrix is\n");
15. for(i=0;i<r;i++)
16. {
17. for(j=0;j<c;j++)
18. {
19. printf("%d\t",a[i][j]);
20. }
21. printf("\n");
22. }
23. printf("\nTHE OUTPUT IS:\n");
24. for(i=0;i<r;i++)
25. {
26. sum=0;
27. for(j=0;j<c;j++)
28. {
29. printf("%d\t",a[i][j]);
30. sum=sum+a[i][j];
31. }
32. printf("| %d",sum);
33. printf("\n");
34. }
35. }
```

**Output:**

```
Enter the number of rows: 4
Enter the number of columns: 3
Enter the elements of matrix: 12 34 56 78 21 32 43 54 65 76 87 98
```

The matrix is

```
12 34 56
78 21 32
76 87 98
```

THE OUTPUT IS:

```
12 34 56 | 102
78 21 32 | 131
43 54 65 | 162
76 87 98 | 261
```

### 10.5.8.3 Write a Program to Add Two Matrices

#### PROGRAM 10.18

```
1. #include<stdio.h>
2. void main()
3. {
4. int a[10][10],b[10][10],c[10][10],i,j,r1,c1,r2,c2;
5. printf("\nEnter the number of rows and columns of 1st
6. matrix: ");
7. scanf("%d %d",&r1,&c1);
8. printf("\nEnter the number of rows and columns of 2nd
9. matrix: ");
10. scanf("%d %d",&r2,&c2);
11. if((r1==r2)&&(c1==c2))
12. {
13. printf("\nEnter the elements of 1st matrix: ");
14. for(i=0;i<r1;i++)
15. for(j=0;j<c1;j++)
16. scanf("%d",&a[i][j]);
17. printf("\nEnter the elements of 2nd matrix: ");
18. for(i=0;i<r2;i++)
19. for(j=0;j<c2;j++)
20. scanf("%d",&b[i][j]);
21. printf("\nThe 1st matrix is\n");
22. for(i=0;i<r1;i++)
23. {
24. for(j=0;j<c1;j++)
```

```
24. {
25. printf("%d\t",a[i][j]);
26. }
27. printf("\n");
28. }
29.
30. printf("\nThe 2nd matrix is\n");
31. for(i=0;i<r2;i++)
32. {
33. for(j=0;j<c2;j++)
34. {
35. printf("%d\t",b[i][j]);
36. }
37. printf("\n");
38. }
39. /* Addition of the two matrices*/
40. for(i=0;i<r1;i++)
41. for(j=0;j<c1;j++)
42. c[i][j]=a[i][j]+b[i][j];
43.
44. printf("\nThe addition of the two matrices
45. is\n");
46. for(i=0;i<r1;i++)
47. {
48. printf("\n");
49. for(j=0;j<c1;j++)
50. printf("%d\t",c[i][j]);
51. }
52. else
53. printf("ADDITION NOT POSSIBLE");
54. }
55.
56.
```

**Output:**

```
Enter the number of rows and columns of 1st matrix: 3 3
Enter the number of rows and columns of 2nd matrix: 3 3
Enter the elements of 1st matrix: 23 43 45 43 21 67 98 33 44
Enter the elements of 2nd matrix: 22 88 44 12 14 16 47 73 93
```

The 1st matrix is

```
23 43 45
43 21 67
98 33 44
```

The 2nd matrix is

```
22 88 44
12 14 16
47 73 93
```

The addition of the two matrices is

```
45 131 89
55 35 83
145 106 137
```

#### 10.5.8.4 Write a Program to Multiply Two Matrices

##### PROGRAM 10.19

```
1. #include<stdio.h>
2. void main()
3. {
4. int a[10][10],b[10][10],c[10][10],i,j,k,r1,c1,r2,c2;
5. printf("\nEnter the number of rows and columns of 1st matrix:
6. ");
7. scanf("%d %d",&r1,&c1);
8. printf("\nEnter the number of rows and columns of 2nd matrix:
9. ");
10. scanf("%d %d",&r2,&c2);
11. if(c1==r2)
12. {
13. printf("\nEnter the elements of 1st matrix: ");
14. for(i=0;i<r1;i++)
15. for(j=0;j<c1;j++)
16. scanf("%d",&a[i][j]);
17. printf("\nEnter the elements of 2nd matrix: ");
18. for(i=0;i<r2;i++)
19. for(j=0;j<c2;j++)
20. scanf("%d",&b[i][j]);
21. printf("\nThe 1st matrix is\n");
```

```
21. for(i=0;i<r1;i++)
22. {
23. for(j=0;j<c1;j++)
24. {
25. printf("%d\t",a[i][j]);
26. }
27. printf("\n");
28. }
29.
30. printf("\nThe 2nd matrix is\n");
31. for(i=0;i<r2;i++)
32. {
33. for(j=0;j<c2;j++)
34. {
35. printf("%d\t",b[i][j]);
36. }
37. printf("\n");
38. }
39. /* Code for matrix multiplication*/
40. for(i=0;i<r1;i++)
41. {
42. for(j=0;j<c2;j++)
43. {
44. c[i][j]=0;
45. for(k=0;k<c1;k++)
46. {
47. c[i][j]=c[i][j]+a[i][k]*b[k][j];
48. }
49. }
50. }
51. printf("\nThe multiplication of the two matrices
is\n");
52. for(i=0;i<r1;i++)
53. {
54. printf("\n");
55. for(j=0;j<c2;j++)
56. {
57. printf("%d\t",c[i][j]);
58. }
59. }
60. }
61. else
```

```
62. printf("MULTIPLICATION NOT POSSIBLE");
63. }
64.
65.
66.
```

*Output:*

```
Enter the number of rows and columns of 1st matrix: 3 3
Enter the number of rows and columns of 2nd matrix: 3 3
Enter the elements of 1st matrix: 23 45 22 33 55 77 31 94 76
Enter the elements of 2nd matrix: 33 66 11 23 45 67 98 32 21
```

The 1st matrix is

```
23 45 22
33 55 77
31 94 76
```

The 2nd matrix is

```
33 66 11
23 45 67
98 32 21
```

The multiplication of the two matrices is

```
3950 4247 3730
9900 7117 5665
10633 8708 8235
```

---

## 10.6 Multidimensional Arrays

As we have already discussed, any array which takes more than one dimension is known as a multidimensional array. In Section 10.5 we discussed the concept of 2D arrays. In this section, we introduce the concept of 3D arrays. In practice, 3D arrays are rarely used.

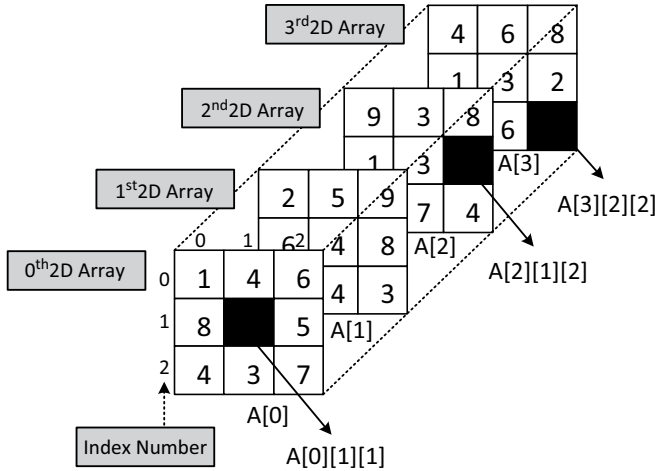
### 10.6.1 Declaration and Representation of 3D Arrays

Declaration of a 3D array is similar to other arrays; the only difference is it takes another dimension. The syntax is shown in Figure 10.19.

Where *size* represents the number of 2D arrays, *RowSize* and *ColSize* represent the number of rows and columns of each 2D array. The conceptual representation of a 3D array A[4]

```
Syntax
DataType ArrayName[size][RowSize][ColSize];
```

**FIGURE 10.19**  
3D array declaration syntax.



**FIGURE 10.20**  
Conceptual representation of a 3D array.

$A[3][3]$  is shown in Figure 10.20.  $A[4][3][3]$  represents four  $3 \times 3$  2D arrays placed back to back.

As discussed, a 3D array is a collection of 2D arrays and the elements are stored in a contiguous memory location. In the declaration, the first index represents the number of 2D arrays. Let us take an example to explain it properly.

```
Example
int A[2][3][4];
```

where,

- 2 indicates the number of 2D arrays;
- 3 represents the number of rows in each 2D array;
- 4 represents the number of columns in each 2D array.

According to the above declaration, array A can store 24 elements; 12 elements in each 2D array.



### 10.6.1.1 Write a Program to Declare a 3D Array, Input Some Numbers, and Display the 3D Array

#### PROGRAM 10.20

```

1. #include<stdio.h>
2. void main()
3. {
4. int A[2][3][4];
5. int i,j,k;
6. printf(" Enter 24 elements: ");
7. for(i=0;i<2;i++)
8. {
9. for(j=0;j<3;j++)
10. {
11. for(k=0;k<4;k++)
12. scanf("%d", &A[i][j][k]);
13. }
14. }
15. printf("The multidimensional array contains:\n");
16. for(i=0;i<2;i++)
17. {
18. for(j=0;j<3;j++)
19. {
20. for(k=0;k<4;k++)
21. printf(" %d ", A[i][j][k]);
22. printf("\n");
23. }
24. printf("\n");
25. }
26. };
27.

```

#### Output:

Enter 24 elements: 12 34 56 78 90 12 23 34 45 56 67 89 90 78 76 54 43 32 21 31 41 44 55  
77

The multidimensional array contains:

```

12 34 56 78
90 12 23 34
45 56 67 89

90 78 76 54
43 32 21 31
41 44 55 77

```

## 10.7 Character Arrays: Strings

Whenever we store several integer elements in an array, we call it an integer array. In the same manner, the collection of a character array is known as a string. We use strings to manipulate or process different words and sentences. Generally, a string is treated as variable-length data. For example, consider the name of a person. By nature, the names of persons vary from each other in their length. A string is a 1D array of characters that is terminated by an escape sequence known as a NULL (“\0”) character (i.e., to mark the end of the string, C uses the “\0” character).

A string is a 1D array of characters that is terminated by an escape sequence known as a NULL (“\0”) character.

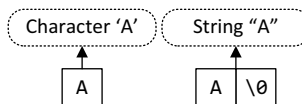
- Strings in C are enclosed within double quotes. Example: “C Programming Language”.
- The string is stored in memory as ASCII codes of the characters that make up the string, appended with 0 (ASCII value of “\0”).
- Normally each character of the string takes one byte in memory.
- Figure 10.21 shows you the difference between the character stored in memory and the one-character string stored in memory. The character requires only one memory location, but the one-character string requires two memory locations.

### 10.7.1 Declaration of a String

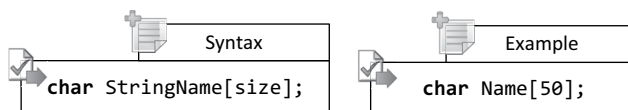
A string can be declared like a 1D array. The syntax of declaring a string and an example is shown in Figure 10.22.

### 10.7.2 Initialization of a String

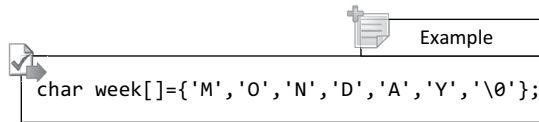
We can initialize a string in two ways. One way is to use the traditional initialization process as for a 1D array with an extra character “\0” appended at the end. One such example



**FIGURE 10.21**  
Representation of a character and a string in memory.

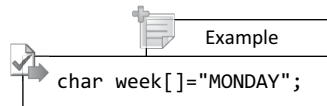


**FIGURE 10.22**  
String declaration syntax with an example.



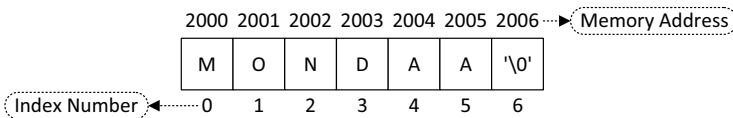
```
char week[]={ 'M', 'O', 'N', 'D', 'A', 'Y', '\0'};
```

**FIGURE 10.23**  
Initializing string example.



```
char week[]="MONDAY";
```

**FIGURE 10.24**  
Initializing a string without using the “\0” character.



**FIGURE 10.25**  
Conceptual representation of a string in memory.

is shown in Figure 10.23. In this declaration, the string *week* is initialized to MONDAY. You notice that a “\0” character is appended at the end to indicate the end of the string.

The C language provides another most uncomplicated way to initialize an array that does not require appending the “\0” character at the end. Figure 10.24 shows this initialization.

With this initialization process, the C compiler automatically adds a NULL character (“\0”) at the end of the string. This is the most convenient way to initialize a string. A string is stored in memory as ASCII codes of the character. The compiler takes care of storing the ASCII codes of the characters of the string in memory and also stores the null terminator at the end. Figure 10.25 shows a conceptual representation of the memory allocation.

### 10.7.3 Reading a String

To read anything from the user requires a `scanf()` function. For reading a string, we can also use the same function. But there is a problem since the `scanf()` function can be used to take input from the user, but a white space character delimits it. That means whenever input is done, `scanf()` recognizes a sequence of characters, and whenever space is encountered, it will stop.

#### 10.7.3.1 Disadvantages of the `scanf()` Function

As discussed above, a `scanf()` function only reads characters in a sequence until and unless whitespace is encountered. But, in general, most strings have whitespaces in them. For example, if we consider a name, we see it is a collection of words, and to separate the words we need whitespaces. So the disadvantage of using `scanf()` is it will read only the first word of your name and ignore the remaining word(s).

Consider Program 10.21 that will explain the problems associated with the `scanf()` function. In this example, we are trying to read a string from the user. Observe the output, and you will notice that we are not able to fulfill our objective of reading a complete string with the `scanf()` function. In the first run, the function gives us the output as desired, but in the second run, it only reads the first word and ignores the second. That's why the `printf()` function displays only the first word.

#### PROGRAM 10.21

```
1. #include<stdio.h>
2. void main()
3. {
4. char str[20];
5. printf("Enter the string : ");
6. scanf("%s",str);
7. printf("The string entered by you : %s",str);
8. }
```

*Output:*

Run 1

Enter the string: Programming  
The string entered by you: Programming

Run 2

Enter the string: Programming Language  
The string entered by you: Programming

*Explanation:*

In the above example, whenever the user enters **Programming** (Run 1) then the whole string will be printed, but at Run 2, whenever the user enters **Programming Language**, it will print only **Programming**, because, when space is encountered, `scanf()` will stop reading the data.

The C language provides an alternative function called `gets()` to avoid this problem. In the following section, we will discuss the `gets()` function.

#### 10.7.3.2 Reading Strings with the `gets()` Function

The best approach to string input is to use the library function called `gets()`. The `gets()` function will read the complete input line, including spaces, and store it in the memory area as a null-terminated string.

#### 10.7.4 Displaying the String

We use the `printf()` function to print something on the screen. Here also we can use the same function to print a string. Another function provided by the C language that does the same thing is known as the `gets()` function. Overall, we have two different functions available to print the content on the screen. For displaying the string, we can use:

- the `printf()` function;
- the `puts()` function.

Generally, `printf()` is convenient because we are acquainted with its format. All the programs in this chapter use the `printf()` function for output. Consider Program 10.22 that uses `printf()` to print the string. Program 10.23 shows the same program using the `puts()` function.

#### PROGRAM 10.22

```
1. #include<stdio.h>
2. void main()
3. {
4. char str[20];
5. printf("Enter the string : ");
6. gets(str);
7. printf("The string entered by you : %s", str);
8. }
```

*Output:*

Run 1

Enter the string: Programming

The string entered by you: Programming

Run 2

Enter the string: Programming Language

The string entered by you: Programming Language

*Explanation:*

For both cases (Run 1 and Run 2) the complete output will come without terminating the string after a space.

#### PROGRAM 10.23

```
1. #include<stdio.h>
2. void main()
3. {
4. char str[20];
5. printf("Enter the string : ");
6. gets(str);
7. printf("The string entered by you : ");
8. puts(str);
9. }
```

*Output:*

Enter the string: Programming Language

The string entered by you: Programming Language

## 10.7.5 Programming Examples

### 10.7.5.1 Find the Length of a String

We know that every string is ended with a NULL character (“\0”). We can easily find out the length of the string by counting the characters up to “\0” using a loop. The program is as follows:

#### PROGRAM 10.24

```
1. #include<stdio.h>
2. #include<conio.h>
3. void main()
4. {
5. char name[30];
6. int len=0,i=0;
7. printf("Enter your name: ");
8. gets(name);
9. while(name[i] !=NULL)
10. {
11. len=len+1;
12. i=i+1;
13. }
14. printf("Length of the string: %d",len);
15. }
```

*Output:*

```
Enter your name: C Programming Learn to code
Length of the String: 27
```

### 10.7.5.2 Count the Number of Words Present in a String

A string is a collection of words, and generally the words are separated by a space. For counting the number of words in a string, we need to search for the blank space present after each word:

- For counting the word, we have to find the blank space between the words;
- When we have one blank space, then the counter will be incremented to 1;
- If we find  $n$  blank spaces, then there must be  $n + 1$  words.

Consider the following program.

**PROGRAM 10.25**

```
1. #include<conio.h>
2. #include<stdio.h>
3. void main()
4. {
5. char str[30];
6. int word=0,i=0;
7. printf("Enter the string : ");
8. gets(str);
9. while(str[i]!='\0')
10. {
11. if(str[i]==' ')
12. {
13. word=word+1;
14. }
15. i=i+1;
16. }
17. printf("There are %d words present", word+1);
18. }
```

*Output:*

```
Enter the string: C programming learn to code
There are 5 words present
```

*Explanation:*

Lines 11–14 increment the value of the variable `word` when a blank space is found. The `word` variable is initialized to 0 in line 6. The while loop in line 9 scans each character until the end of the string. As shown in the output, there are four blank spaces; hence the number of words will be 5.

**10.7.5.3 Reverse the String**

There are several ways to do this. In our example, we take two strings. The first string will store the original string, and the other one will store its reverse. The process is simple:

- First, find the length of the given string. The length is required because we want to access the characters of the given string in the reverse manner. The length will be used as the last index of the given string.
- Use a loop that will start from the character in the last index, and move towards the beginning of the string. During this process, copy the characters to the second string.
- Finally, print the second string that will contain the reverse of the given string.

Consider the following program.

**PROGRAM 10.26**

```
1. #include<stdio.h>
2. #include<conio.h>
3. void main()
4. {
5. char str[30],copy[30];
6. int len=0,i,j;
7. printf("Enter a string: ");
8. gets(str);
9. i=0;
10. while(str[i]!=NULL)
11. {
12. len=len+1;
13. i=i+1;
14. }
15. for(i=0,j=len-1;i<len;i++,j--)
16. {
17. copy[i]=str[j];
18. }
19. copy[i]=NULL;
20. printf("\nThe original string is: %s ",str);
21. printf("\nThe reverse of the string is: %s",copy);
22. }
```

*Output:*

```
Enter a string: C Programming Learn to Code
The original string is: C Programming Learn to Code
The reverse of the string is: edoC ot nraeL gnimmargorP C
```

*Explanation:*

Lines 9–14 are used to find the length of the string. Lines 15–18 will read the character from the last index ( $len-1$ ) of the original string *str* and copy it to the second string *copy*. After copying all the characters, line 19 appends a NULL character (“\0”) at the end to mark the end of the string. Finally, lines 20 and 21 print the original and reverse strings, respectively.

**10.7.5.4 Check Whether the String is a Palindrome or Not**

This is the process of checking the similarity between a given string and its reverse string. For example, "MADAM" is a string, and if you find its reverse, it remains the same, so the string "MADAM" is a palindrome. Consider the following program.



**PROGRAM 10.27**

```
1. #include<stdio.h>
2. #include<conio.h>
3. void main()
4. {
5. char str[30];
6. int len=0,i,j;
7. printf("Enter your name: ");
8. gets(str);
9. i=0;
10. while(str[i]!=NULL)
11. {
12. len=len+1;
13. i=i+1;
14. }
15. }
16. for(i=0,j=len-1;i<j;i++,j--)
17. {
18. if(str[i]!=str[j])
19. break;
20. }
21. if(i==j || i==j+1)
22. printf("STRING PALLINDROME");
23. else
24. printf("NOT PALLINDROME");
25. }
```

*Output:*

Run 1

Enter your name: MADAM  
STRING PALLINDROME

Run 2

Enter your name: Programming  
NOT PALLINDROME

---

## 10.8 String Functions

For easy manipulation of the string, the C compiler provides a set of functions which is defined inside the header file *string.h*. To use these functions, the header file `<string.h>` must be included in the program. In the following section, we discuss some of the commonly used string functions.

### 10.8.1 strcpy (Destination, Source)

You can't just use `string1=string2` in C. You have to use the `strcpy()` function to copy one string to another.

**Example:**

```
S1 = "abc";
S2 = "xyz";
strcpy(S1, S2); /*S1="xyz"*/
```

### 10.8.2 strcat (Destination, Source)

This joins the destination and source strings and puts the joined string into the destination string.

**Example:**

```
S1 = "abc";
S2 = "xyz";
strcat (S1, S2); /*S1="abcxyz"*/
```

### 10.8.3 strcmp (First, Second)

Compare the first and second strings:

- If the first string is greater than the second string, then a number greater than 0 is returned;
- If the first string is less than the second, then a number less than 0 is returned;
- If the strings are equal, then 0 is returned.

Beside the above functions many other functions are also defined inside `<string.h>` that are shown in Table 10.1.

**TABLE 10.1**

List of String Functions

| Sl. No. | Functions                       | Descriptions                                    |
|---------|---------------------------------|-------------------------------------------------|
| 1       | <code>strlen(s1)</code>         | Returns the length of the string s1             |
| 2       | <code>strlwr(s1)</code>         | Converts string to lowercase.                   |
| 3       | <code>strupr(s1)</code>         | Converts the string to uppercase.               |
| 4       | <code>strncat(s1, s2, n)</code> | Appends n characters of string s2 to s1         |
| 5       | <code>strncpy(s1, s2, n)</code> | Copies n characters of string s2 to s1          |
| 6       | <code>strrev(s1)</code>         | Converts string to reverse                      |
| 7       | <code>strncmp(s1, s2, n)</code> | Compares first n characters of string s1 and s2 |

### 10.8.4 Programming Examples Using String Functions

#### PROGRAM 10.28: STRING COMPARE

```
1. #include <stdio.h>
2. #include <string.h>
3. void main()
4. {
5. char s1[100],s2[100];
6. printf("Enter string 1: "); gets(s1);
7. printf("Enter string 2: "); gets(s2);
8. if(strcmp(s1,s2)==0)
9. printf("1 and 2 are equal\n");
10. else if (strcmp(s1,s2)<0)
11. printf("1 less than 2\n");
12. else
13. printf("1 greater than 2\n");
14. }
```

#### Output:

Run 1

```
Enter string 1: Program
Enter string 2: Language
1 greater than 2
```

Run 2

```
Enter string 1: Program
Enter string 2: Program
1 and 2 are equal
```

Run 3

```
Enter string 1: Language
Enter string 2: Program
1 less than 2
```

#### PROGRAM 10.29: STRING LENGTH

```
1. #include<stdio.h>
2. #include<string.h>
3. void main()
4. {
5. char name[100];
6. int length;
7. printf("Enter the string: ");
8. gets(name);
```

```
9. length=strlen(name);
10. printf("\n Length of the string is= %d", length);
11. }
```

*Output:*

Enter the string: C Programming Learn to Code  
Number of characters in the string is= 27

---

## 10.9 Review Questions

### 10.9.1 Objective Questions

1. \_\_\_\_\_ is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index number to each location.
2. A character array is also known as \_\_\_\_\_.
3. \_\_\_\_\_ is known as a subscripted variable.
4. The starting index of an array begins with the number zero (0). True/false?
5. The elements present in an array are stored in a contiguous memory location. True/false?
6. To access an array element, we need to specify the \_\_\_\_\_ followed by the element's \_\_\_\_\_ within a square bracket.
7. In which representation of a 2D array does the first row occupy the first set of the memory location, the second occupy the next set, and so on?
8. In which representation of a 2D array does the first column occupy the first set of the memory location, the second occupy the next set, and so on?
9. A \_\_\_\_\_ is a 1D array of characters that is terminated by an escape sequence known as a NULL character.
10. Which predefined function is used to copy the content of one string to another?
11. Which function is used to find the length of a string?

### 10.9.2 Subjective Questions

1. What is an array? Classify arrays regarding the type and memory representation of its elements.
2. Why is an array called a subscripted variable? What are the characteristics of an array?
3. What are the advantages and disadvantages of an array?
4. Explain the initialization procedure of a 1D string and a 2D array with examples.
5. What is a 2D array? How are the elements of a 2D array represented in memory?

6. Explain the row major and column major order of 2D array element representation. Explain their formulas to find the address of any given element with an appropriate example.
7. Prove that a 2D array is a collection of many 1D arrays with appropriate programming examples.
8. Assume that a  $3 \times 5$  2D array named A is represented in row major order, and that it stores real numbers. Let the real number take four bytes. The base address is 2000. Find the address of A[3][4].
9. Assume that a  $3 \times 5$  2D array named A is represented in column major order, and that it stores real numbers. Let the real number take four bytes. The base address is 2000. Find the address of A[3][4].
10. What is a string? How is a string constant different from a character constant?
11. What is the difference between the `gets()` function and the `scanf()` function? Explain with an example.

### 10.9.3 Programming Exercises

1. Write a program to search for an element in an array.
2. Write a program to add all the elements in an array.
3. Write a program to find out the average of the elements present in an array.
4. Write a program to find the largest and smallest elements in an array.
5. Write a program to insert an element into an array.
6. Write a program to delete an element in an array.
7. Write a program to sort the elements in an array.
8. Write a program to copy one array to another array.
9. Write a program to swap two arrays.
10. Write a program to find out all the even and odd numbers present in an array.
11. Write a program to merge two arrays.
12. Write a program to delete all the duplicate elements present in an array and print the final array.
13. Write a program to reverse an array.
14. Write a program to print the numbers that are greater than the average.
15. Write a program to print all those numbers which are not divisible by 2 in an array of  $n$  integers.
16. Write a program to print all the array elements excluding the elements divisible by 2.
17. Write a program to print all the prime numbers present in an array.
18. Write a program to generate the Fibonacci series using an array.
19. Write a program to arrange the numbers in an array in such a way that the array will have the odd numbers followed by the even numbers.
20. Write a program to rearrange an array in reverse order without using a second array.
21. Write a program to swap the  $k$ th and  $(k+1)$ th elements in an integer array.  $k$  is given by the user.

22. Write a program to find the intersection of two sets of numbers.
23. Write a program to print the word associated with a corresponding number. For example, If 234 is entered through the keyboard then your program must print “two three four”.
24. Write a program to add two matrices.
25. Write a program to multiply two matrices.
26. Write a program to find the transpose of a matrix.
27. Write a program to add all the elements of a matrix.
28. Write a program to add the diagonal elements of a square matrix.
29. Write a program to add the elements of each row of a matrix and print them in the following format.

| Input Matrix  | Output             |
|---------------|--------------------|
| 2   3   4     | 2   3   4       9  |
| 4   6   7   ⇨ | 4   6   7       17 |
| 2   8   9     | 2   8   9       19 |

30. Write a program to add the elements of each column of a matrix and print them in the following format.

| Input Matrix  | Output      |
|---------------|-------------|
| 2   3   4     | 2   3   4   |
| 4   6   7   ⇨ | 4   6   7   |
| 2   8   9     | 2   8   9   |
|               | -----       |
|               | 8   17   20 |

31. Write a program to add the elements above the main diagonal.
32. Write a program to add the elements below the main diagonal.
33. Write a program to print the elements above the main diagonal.
34. Write a program to print the elements below the main diagonal.
35. Write a program to print the elements above the main diagonal including the diagonal elements.
36. Write a program to print the elements below the main diagonal including the diagonal elements.
37. Write a program to convert a string to upper case assuming that the string is entered in lower case explicitly.
38. Write a program to concatenate two strings without using a string function.
39. Write a program to count the number of vowels, consonants, and spaces in a line.
40. Write a program to alternate the case of every character in the input string. For example if the user enters aBCdEf GhiJK then the output will become AbcDEf gHIjk.

41. Write a program to accept a word from the user and print it in the following way.

For example if the word is PROGRAM, the program will print,

```
P
P R
P R O
P R O G
P R O G R
P R O G R A
P R O G R A M
```

42. Write a program to read a text and count all the occurrences of a particular letter given by the user.
43. Write a program to find the longest word in a string.
44. Write a program to copy a string to another string.
45. Write a program to read a word and rewrite it in alphabetical order.
46. Write a program to delete a word from a string.
47. Write a program to convert each character of a string into the next alphabetic letter and print the string.

# 11

## Pointers

### 11.1 Introduction

A pointer is something that indicates or points. From the layman's point of view a pointer is a long-tapered stick for indicating objects, as on a chart or a blackboard (Figure 11.1). Sometimes we use a laser pointer to point or show an object displayed on a screen (Figure 11.1). So, we can say a pointer is *indirectly pointing* to a device of our concerns.

To understand the way in which pointers operate, it is first necessary to understand the concept of *indirection*. You are familiar with this concept from your everyday life. For example, suppose you need to buy a new computer for your department. In the company that you work for, all purchases are handled by the purchasing department. So, you call Mr. X in purchasing and ask him to order the new computer for you. Mr. X, in turn, calls the local supply store to order the computer. This approach to obtain your new computer is actually an indirect one because you are not ordering the computer directly from the supply store yourself.

This same notion of indirection applies to the way pointers work in C. A pointer provides an indirect means of accessing the value of a particular data item. In C programming, a *pointer* is a variable that represents the *location* (rather than the *value*) of a data item, such as a variable or an array element. Pointers play an important role in the development process. A pointer has a number of useful applications. For example, pointers can be used to pass information back and forth between a function and its reference point. In particular, pointers provide a way to return multiple data items from a function via function arguments.

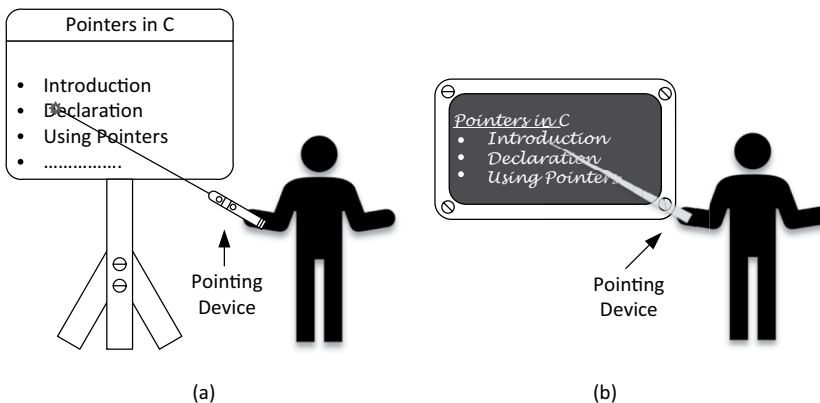


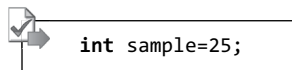
FIGURE 11.1  
Pointing device.



## 11.2 Basic Knowledge

Within the computer's memory, every stored data item occupies one or more contiguous memory cells (i.e., adjacent words or bytes). The number of memory cells required to store a data item depends on the type of data item. For example, a single character will typically be stored in one byte (eight bits) of memory; an integer usually requires four contiguous bytes; a floating-point number requires four contiguous bytes; and a double-precision quantity requires eight contiguous bytes.

Suppose, a sample is a variable that represents some particular data item. The compiler will automatically assign memory cells for this data item. For example, if we declare a variable sample as follows:



```
int sample=25;
```

Then, in computer memory, this variable sample takes a place and has an address as shown in Figure 11.2.

Now, we can access the value of the variable by using the variable name and we can access the address of the variable by using the **address of (&)** operator.

To explain, analyze Program 11.1.

### PROGRAM 11.1

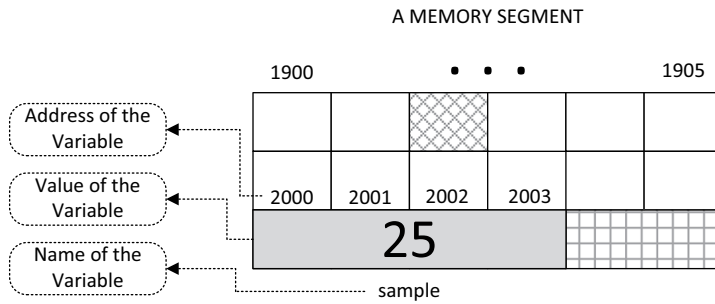
```
#include<stdio.h>
void main()
{
 int sample=25;
 printf ("\n Value of the variable=%d", sample);
 printf ("\n Address of the variable=%u", &sample);
}
```

*Output:*

```
Value of the variable=25
Address of the variable=2000
```

In the above example, we use %u for an unsigned integer, because an address cannot be negative. We may summarize this as:

- Every variable has an address;
- We can know the address of the variable using the address of the operator (&);
- The address cannot be negative;
- For accessing the value of the variable, we need the name of the variable.



**FIGURE 11.2**  
Memory allocation for the variable sample.

### 11.3 Pointer Variables

A pointer:

- Is a special type of variable;
- It stores the address of another variable rather than the value;
- The data type of the pointer variable is the same as the data type of the variable to which it points.

#### 11.3.1 Declaration of Pointer Variables

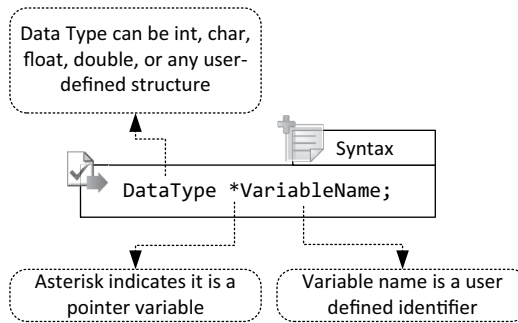
When a pointer variable is declared, the user must know the type of the variable to which the pointer is going to point. Declaration of the pointer is the same as variable declaration, the only difference is the variable name must be followed by an asterisk (\*). The asterisk indicates that the variable is a pointer variable. Figure 11.3 shows the syntax of declaring a pointer and its component.

Examples:

```
int *ptr; /* ptr is a pointer to an integer, which means it can only store the address of an integer*/
float *fptr; /* fptr is a pointer to a float, which means it can only store the address of a float*/
char *cptr; /* cptr is a pointer to a character, which means it can only store the address of a character*/
```

#### 11.3.2 Working with Pointers

In Section 11.2, we saw the *address of (&)* operator for getting the variable’s address. Now we know that a pointer is a special type of variable that can store another variable’s address. So, by using the *address of (&)* operator, we can easily store another variable’s address in a pointer variable. Program 11.2 shows the process of doing it:



**FIGURE 11.3**  
Syntax of pointer declaration.

### PROGRAM 11.2

```
void main()
{
 int sample = 25;
 int *ptr;
 ptr = &sample;
 printf ("\n Address of the variable = %u", &sample);
 printf ("\n Address of the variable = %u", ptr);
}
```

*Output:*

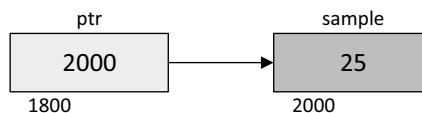
```
Address of the variable = 2000
Address of the variable = 2000
```

So, from the above output we understood that both outputs are the same and logically the whole relationship can be visualized as shown in Figure 11.4.

In Program 11.2, the variable `sample` and the pointer variable `ptr` both are of integer type. Whenever a pointer points to any variable, the data type of both variables (i.e., pointer variable and the variable to which it points) must be the same.

We can access the value of the variable by using a special type of pointer operator available in C which is `"*"`, called the **"value at address"** operator. This gives the value stored at a particular address. The **"value at address"** operator is also called an **"indirection"** operator.

Analyze Program 11.3 carefully.



**FIGURE 11.4**  
Illustration of pointer variable.

**PROGRAM 11.3**

```
#include<stdio.h>
void main()
{
 int sample = 25;
 printf ("\nAddress of sample = %u", &sample);
 printf ("\nValue of sample = %d", sample);
 printf ("\nValue of sample = %d", *(&sample));
}
```

*Output:*

```
Address of sample = 2000
Value of sample = 25
Value of sample = 25
```

Note that printing the value of `*(&sample)` is the same as printing the value of the `sample`. The expression `&sample` gives the address of the variable `sample` and the `*` operator gives the value present at that address. If we combine the concepts shown in Programs 11.2 and 11.3, then we can access the value of the variable through a pointer:

**11.3.3 Workout**

In this section we will see some more examples that will help in strengthening the concept of the pointer. Program 11.4 shows different ways to access the variables value through a pointer.

**PROGRAM 11.4**

```
#include<stdio.h>
void main()
{
 int sample = 25;
 int *ptr;
 ptr = &sample;
 printf ("\n Address of the variable = %u", &sample);
 printf ("\n Address of the variable = %u", ptr);
 printf ("\n Value of sample = %d", sample);
 printf ("\n Value of sample = %d", *(&sample));
 printf ("\n Value of sample = %d", *ptr);
}
```

*Output:*

```
Address of the variable = 2000
Address of the variable = 2000
Value of sample = 25
Value of sample = 25
Value of sample = 25
```

## PROGRAM 11.5

```

1. #include<stdio.h>
2. void main()
3. {
4. int x = 25;
5. int y = 30;
6. int *p;
7. int *q;
8. p = &x;
9. q = &y;
10. printf ("Value of x = %d", *p);
11. printf ("Value of y = %d", *q);

12. *p = y + x;
13. *q = *p + y;

14. printf ("Value of x = %d", x);
15. printf ("Value of y = %d", y);
16.}

```

To find the output of Program 11.5, let us represent the variables and pointers in graphical form:

- According to lines 4, 5, 6, 7, 8, and 9 the following logical structure (Figure 11.5) is created in memory. That means the pointer *p* points to variable *x* and the pointer *q* points to the variable *y* as shown in Figure 11.5.
- So the output of lines 10 and 11 will be:  
Value of *x* = 25  
Value of *y* = 30
- Line 12 can be calculated as shown in Figure 11.6. After execution *x* becomes 55.
- Similarly, line 13 can be calculated as shown in Figure 11.7. After execution *y* becomes 80.
- So the output of lines 14 and 15 will be:  
Value of *x* = 55  
Value of *y* = 80

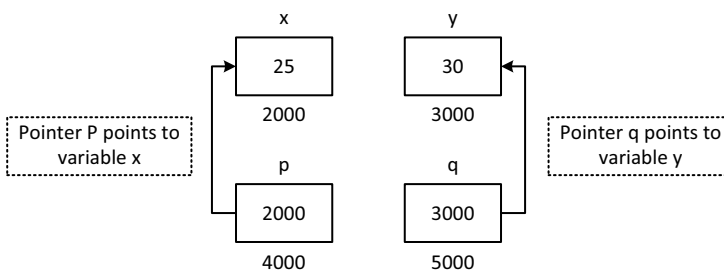
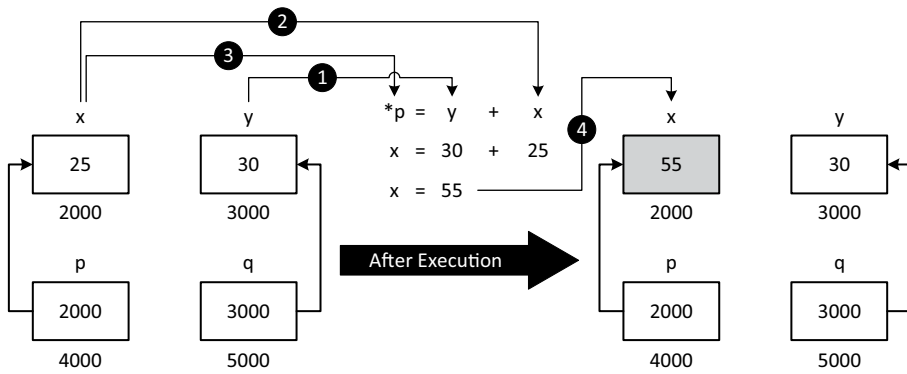
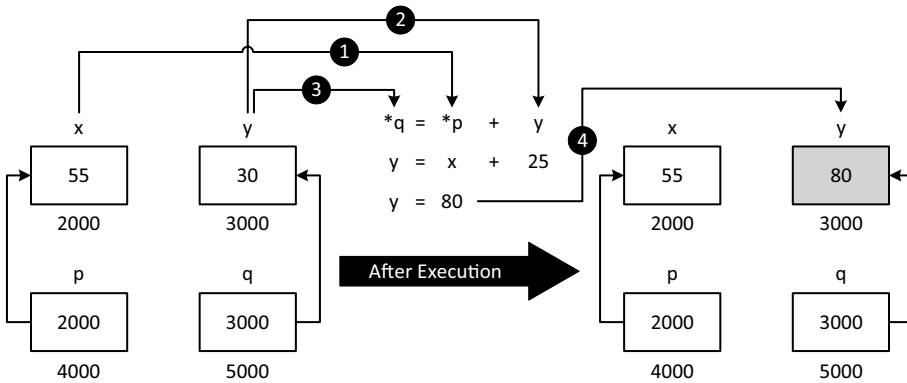


FIGURE 11.5

Logical representation of variables and pointers in memory.



**FIGURE 11.6**  
Execution of line 12.



**FIGURE 11.7**  
Execution of line 13.

### 11.4 Pointer to Pointer (Double Pointer)

In the previous section we saw the pointers which point to another variable. In this section we will see how a pointer can point to another pointer. To implement this concept, we need a double pointer or pointer to pointer.

The syntax for declaring a double pointer is shown below:

Syntax  
`DataType **VariableName;`

To further explain the working of a double pointer, let us take an example program (Program 11.6):

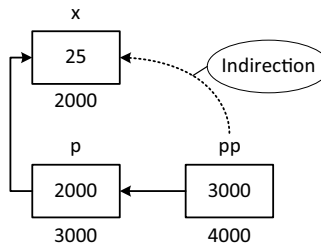
## PROGRAM 11.6

```

1.#include<stdio.h>
2.void main()
3.{
4. int x=25;
5. int *p;
6. int **pp;
7. p=&x;
8. pp=&p;
9. printf("\nx=%d", x);
10. printf("\nx=%d", *p);
11. printf("\nx=%d", **pp);
12.}

```

Line 6 declares a double pointer `pp`. In line 8 `pp` is assigned by the address of `p`, where `p` is a pointer and points to `x`, as given in line 7. So, the whole relationship can graphically be represented as shown in Figure 11.8.



**FIGURE 11.8**  
Double pointer.

The output of the above program will be:

```

x=25
x=25
x=25

```

## 11.5 Void Pointers

We know that pointers are used for pointing to different data types. A float pointer points to a float variable, an int pointer points to an integer variable, and so on. But sometimes we need a general-purpose pointer which can store the address of any type of variable, and that pointer is known as a *void pointer* or a *generic pointer*.

A void pointer can be declared as follows:

|                                  |
|----------------------------------|
| Syntax                           |
| <code>void *VariableName;</code> |

Pointers to void cannot be directly dereferenced like other pointer variables by using \*. This pointer must be properly typecast before they are going to be used. Let us take an example.

### PROGRAM 11.7

```

1. #include<stdio.h>
2. void main()
3. {
4. int x=25;
5. void *p;
6. p=&x;
7. printf("\nx=%d", *p);
8. }
```

Line 6 is a valid assignment, because a void pointer can store the address of any variable. This program will show an error in line 7. Before we use the pointer *p*, we must typecast it with an appropriate data type. Let us rewrite Program 11.7.

### PROGRAM 11.8

```

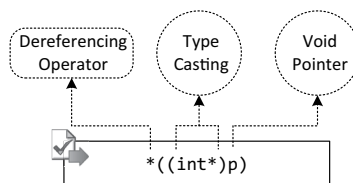
#include<stdio.h>
void main()
{
 int x=25;
 void *p;
 p=&x;
 printf("\nx=%d", *((int*)p));
}
```

Now the output of the program will be:

x=25

The expression `*((int*)p)` is explained below (Figure 11.9).

A void pointer must be correctly typecast before it is used. In Figure 11.9, `int*` converts the pointer *p* into an integer pointer. After that, we use the dereferencing operator to get the value of *x*, which is pointed to by *p*.



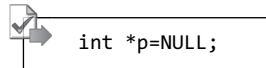
**FIGURE 11.9**  
Explanation of void pointer.



---

## 11.6 Null Pointers

The literal meaning of a null pointer is a pointer that points to nothing. A null pointer is declared as follows:

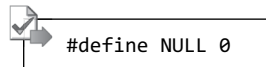


```
int *p=NULL;
```

Here `p` is pointing to an integer whose initial value is 0. Further we can say that `p` is a pointer which points to nothing.

### 11.6.1 What is the Meaning of NULL?

NULL is a macro-constant, and the definition is found in the header file `stdio.h`, `alloc.h`, `mem.h`, `stddef.h`, and `stdlib.h` as:



```
#define NULL 0
```

**Quiz:** What will be the output of the following C program?

#### PROGRAM 11.9

```
#include <stdio.h>
void main()
{
 if(!NULL)
 printf("I know preprocessor");
 else
 printf("I don't know preprocessor");
}
```

*Output:*

I know preprocessor

*Explanation:*

!NULL = !0 = 1

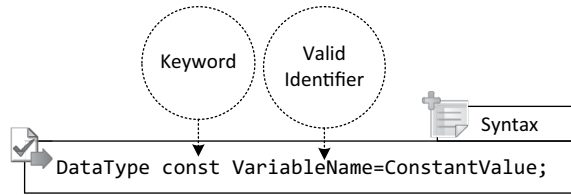
In an if condition, any non-zero number means true.

---

## 11.7 Constant Pointers

The fundamental concept of a constant pointer is similar to a constant variable. We know that a constant variable is a variable whose value cannot be changed during the program execution, and a constant variable must be assigned a value when it is declared.

The declaration of a constant variable can take the following form:



Let us take an example for this purpose shown in Program 11.10 (a-c).

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <div style="border: 1px solid black; padding: 5px; margin-bottom: 5px;">  Program 11.10 (a)                 </div> <pre style="border: 1px solid black; padding: 10px; font-family: monospace;">                 #include&lt;stdio.h&gt;                 void main()                 {                     int const x=5;                     printf("%d", x);                 }             </pre> <div style="border: 1px dashed black; border-radius: 50%; padding: 10px; width: fit-content; margin: 10px auto;">                 This Program will not show any error and the output is: 5             </div> | <div style="border: 1px solid black; padding: 5px; margin-bottom: 5px;">  Program 11.10 (b)                 </div> <pre style="border: 1px solid black; padding: 10px; font-family: monospace;">                 #include&lt;stdio.h&gt;                 void main()                 {                     int const x;                     x=5;                     printf("%d", x);                 }             </pre> <div style="border: 1px dashed black; border-radius: 50%; padding: 10px; width: fit-content; margin: 10px auto;">                 Error: Constant Variable x must be initialized             </div> | <div style="border: 1px solid black; padding: 5px; margin-bottom: 5px;">  Program 11.10 (c)                 </div> <pre style="border: 1px solid black; padding: 10px; font-family: monospace;">                 #include&lt;stdio.h&gt;                 void main()                 {                     int const x=5;                     x=5+5;                     printf("%d", x);                 }             </pre> <div style="border: 1px dashed black; border-radius: 50%; padding: 10px; width: fit-content; margin: 10px auto;">                 Error: Cannot modify a constant object             </div> |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

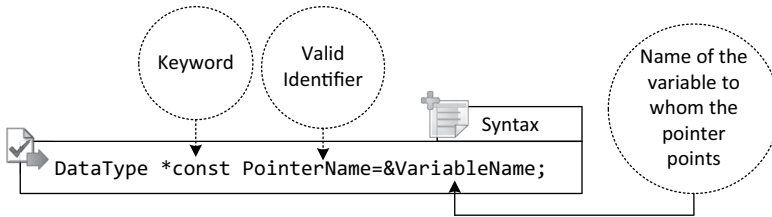
Similar to the above concept, a constant pointer also follows the same rule:

- A constant pointer must be initialized when it is declared.
- The value of a constant pointer cannot be changed during program execution. That means if a constant pointer is pointing to a variable, let us say x, then throughout the program the constant pointer must point to x.

**NOTE**

Constant pointers are those which cannot change the address they are pointing to.

The declaration of a constant pointer takes the following form:

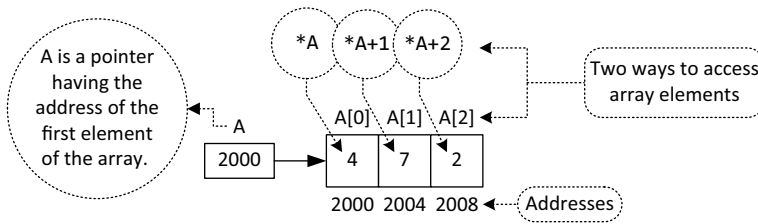


Let us explain the above concept using some sample programs:

|                                                                                                                                         |                                                                                                                                                |                                                                                                                                                            |
|-----------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>Program 11.11 (a)</p> <pre>#include&lt;stdio.h&gt; void main() {     int x=5;     int *const p=&amp;x;     printf("%d", *p); }</pre> | <p>Program 11.11 (b)</p> <pre>#include&lt;stdio.h&gt; void main() {     int x=5;     int *const p;     p=&amp;x;     printf("%d", *p); }</pre> | <p>Program 11.11 (c)</p> <pre>#include&lt;stdio.h&gt; void main() {     int x=5, y=6;     int *const p=&amp;x;     p=&amp;y;     printf("%d", *p); }</pre> |
| <p>This Program will not show any error and the output is:<br/>5</p>                                                                    | <p>Error: Constant Variable p must be initialized</p>                                                                                          | <p>Error: Cannot modify a constant object</p>                                                                                                              |

**NOTE**  
An array name is a constant pointer. The detail of this is discussed in Section 11.10.

Let us take a small example to explain the above note. In Program 11.12(a), we would like to print the value of array A using the indirection operator with the array name. \*A indicates the value at address A. The actual memory representation of array A is shown in Figure 11.10. According to the memory representation, A is a pointer that stores the address of the first element. When we write printf(“%d”, \*A), which means the value at address A, it prints 4. Similarly, printf(“%d”, \*(A+1)) will print the next element, which is 7, and so on.



**FIGURE 11.10**  
Representation of array in memory.

| Program 11.12 (a)                                                                                                                                  | Program 11.12 (b)                                                                                                                                                | Program 11.12 (c)                                                                                                                                                                     |
|----------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>#include&lt;stdio.h&gt; void main() {     int A[3]={4,7,2};     printf("%d", *A);     printf("%d", *(A+1));     printf("%d", *(A+2)); }</pre> | <pre>#include&lt;stdio.h&gt; void main() {     int A[3]={4,7,2};     printf("%d", *A);     A=A+1;     printf("%d", *A);     A=A+1;     printf("%d", *A); }</pre> | <pre>#include&lt;stdio.h&gt; void main() {     int A[3]={4,7,2};     int *p;     p=A;     printf("%d", *p);     p=p+1;     printf("%d", *p);     p=p+1;     printf("%d", *p); }</pre> |
| <p>This Program will not show any error and the output is:<br/>4 7 2</p>                                                                           | <p>Error:<br/>Lvalue Required</p>                                                                                                                                | <p>This Program will not show any error and the output is:<br/>4 7 2</p>                                                                                                              |

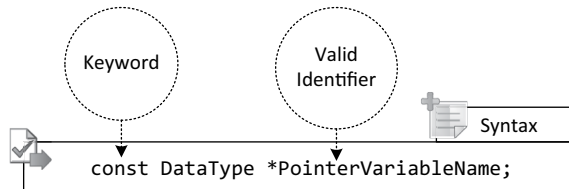
Now consider Program 11.12(b). This program will show you an error “Lvalue Required”. This is because, in the line `A=A+1`, we are trying to assign a new address to `A`. Assigning a new value to `A` is not possible because `A` is a constant pointer and we cannot change the content of `A` throughout its lifetime.

However, if we take a normal integer pointer for this case, then the program will not show any error. Program 11.12(c) shows the code for it. What we do here is, we first assign the value (here it is an address) of `A` to an integer pointer `p`. Later, we increment the value of `p` using the line `p=p+1`, and continue printing the value at address `p`. As `p` is a normal integer pointer (not a constant pointer), we can assign a new address to it. Hence, it is proved that the array name is a constant pointer. We cannot assign a new address to this pointer.

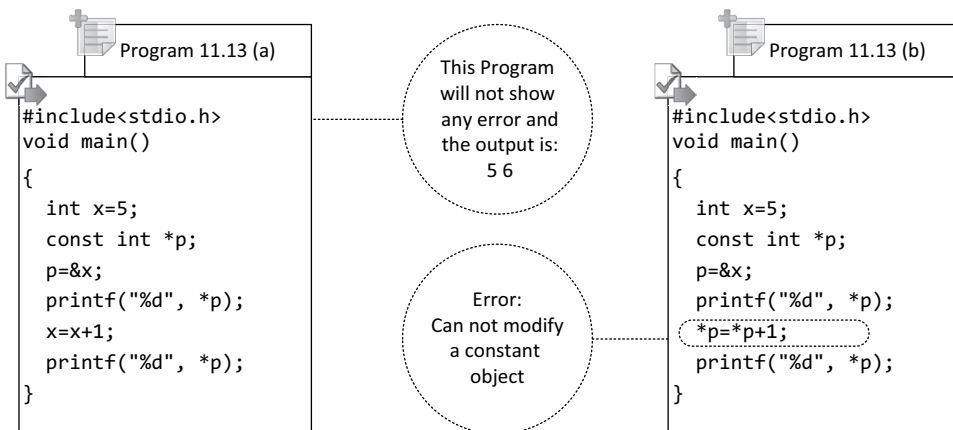
### 11.7.1 Pointers to Constants

When a pointer is not able to change the value of a variable to which it points, it is known as a pointer to a constant. Suppose a pointer `p` points to an integer variable `x`, but `p` cannot change the value of `x`, then `p` is called a pointer to a constant.

The declaration of pointers to constants takes the following form:



Let us take an example:



Program 11.13 (a) will not show any error. The first `printf()` statement will print 5, because we are printing the value at address `p` (`*p`), and `p` contains the address of `x`. The second `printf()` statement will print 6, because the previous line `x=x+1` will increment the value of `x`. However, Program 11.13 (b) will show you an error. This is because, in line `*p=*p+1` we are trying to modify the content of `x` through `p`, which is not allowed.

## 11.8 Pointer Arithmetic

Pointer arithmetic refers to the different arithmetic operations that can be applied on pointers. A limited number of arithmetic operations can be performed upon pointers. The number of bytes a pointer can access in memory depends upon the type of variable it points to. For example, a pointer to an integer accesses four bytes of memory, a pointer to a character accesses one byte of memory, and so on. Some of the operations that are applied to pointer variables are:

- Increment and decrement;
- Addition and subtraction;

- Comparison;
- Assignment.

The increment (++) operator increases the value of a pointer by the size of the data object the pointer points to. For example, if the pointer refers to the first element in an array, the ++ makes the pointer refer to the second element in the array. That means if the array is an integer array then the pointer is increased by four bytes and if the array is a float array then the pointer is increased by four bytes. This is shown in Figure 11.11.

#### PROGRAM 11.14

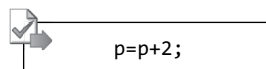
```
#include<stdio.h>
void main()
{
 int A[5]={10, 50, 20, 30, 90};
 int *p;
 p=&A[0];
 printf("%d ", *p);
 p++;
 printf("%d ", *p);
}
```

*Output:*  
10 50

#### *Explanation:*

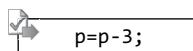
This was shown in Figure 11.11. Similarly, the decrement (--) operator decreases the value of a pointer by the size of the data object the pointer refers to. For example, if the pointer refers to the second element in an array, the -- makes the pointer refer to the first element in the array.

You can add an integer to a pointer, but you cannot add a pointer to a pointer. If the pointer *p* points to the first element in an array, the following expression causes the pointer to point to the third element in the same array:



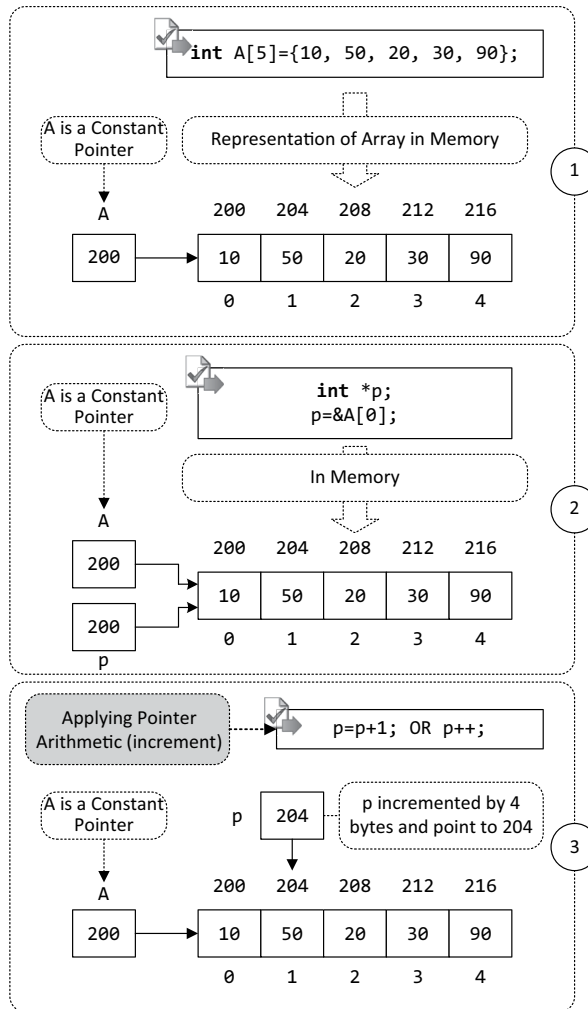
```
p=p+2;
```

Similarly, if a pointer *p* points to the fourth element of an array, the following expression causes the pointer to point to the first element in the same array:



```
p=p-3;
```

This concept is shown in Figure 11.12.

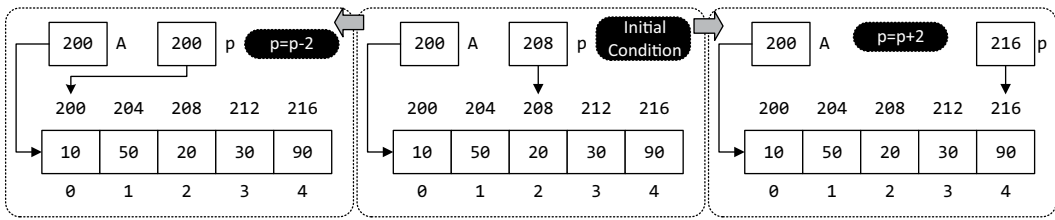


**FIGURE 11.11**  
Applying increment operation on pointer variables.

If you have two pointers that point to the same array, you can subtract one pointer from the other. This operation yields the number of elements in the array that separate the two addresses that the pointers refer to. Program 11.15 explains this.

### PROGRAM 11.15

```
#include<stdio.h>
void main()
{
 float A[5]={1.4, 5.8, 2.3, 6.4, 9.1};
 float *p, *q;
 int size;
```

**FIGURE 11.12**

Applying pointer arithmetic (addition and subtraction).

```

p=&A[2];
q=&A[4];
size=q-p;
printf("%d", size);
}

```

*Output:*  
2

You can compare two pointers with the following operators: ==, !=, <, >, <=, and >=. Pointer comparisons are defined only when the pointers point to elements of the same array. Pointer comparisons using the == and != operators can be performed even when the pointers point to elements of different arrays. Some operations should not be applied to pointers like:

#### PROGRAM 11.16

```

#include<stdio.h>
void main()
{
 float A[5]={1.4, 5.8, 2.3, 6.4, 9.1};
 float *p, *q;
 int size;
 p=&A[2];
 q=&A[2];
 size=p==q;
 printf("%d ", size);
}

```

*Output:*  
1

#### *Explanation:*

In this program, as both pointers point to the same location so the comparison operator (==) returns 1 and the output will be 1.



**PROGRAM 11.17**

```
#include<stdio.h>
void main()
{
 int A[5]={4, 8, 3, 64, 91};
 int B[7]={99, 66, 33, 64, 22, 66, 55};
 int *p, *q;
 int size;
 p=&A[2];
 q=&B[2];
 size=p==q;
 printf("%d ", size);
}
```

*Output:*

0

*Explanation:*

In this program, both the pointers point to the third element. But the output is 0 because the pointers are pointing to two different arrays.

Some operations should not be applied to pointers like:

- Non-integer values should not be added to any pointer;
- Assignment of one pointer to another should be avoided, if the data type of the pointers mismatch;
- Addition, multiplication, and division of two pointers is not possible.

---

## 11.9 Pointers and Functions

By now, we are well familiar with how to call functions. Whenever we call a function and pass something to it, we have always passed the “values” of variables to the called function. Instead of passing the value of a variable, can we pass the location number (also called an address) of the variable to a function? The answer is yes.

So finally, we can say, the passing of arguments to a function can be done in two ways:

- Pass by value;
- Pass by address.

### 11.9.1 Pass by Value

In this method the “value” of each of the actual arguments in the calling function is copied into corresponding formal arguments of the called function. With this method the changes

made to the formal arguments in the called function have no effect on the values of actual arguments in the calling function. The following program illustrates “pass by value.”

#### PROGRAM 11.18

```
#include<stdio.h>
void display(int p)
{
 p=p+10;
 printf("\nInside Function: %d",p);
}
void main()
{
 int x=25;
 display(x);
 printf("\nInside Main: %d",x);
}
```

In this program, I have passed the value of *x* to the function `display()`, which receives that value through a variable *p*. Inside the `display` function the value of *p* is increased by 10 which does not affect the value of *x* in the function `main`, because both variables have a different scope. The whole process of control flow is shown in Figure 11.13, and the output of this program is:

```
Inside Function: 35
Inside Main: 25
```

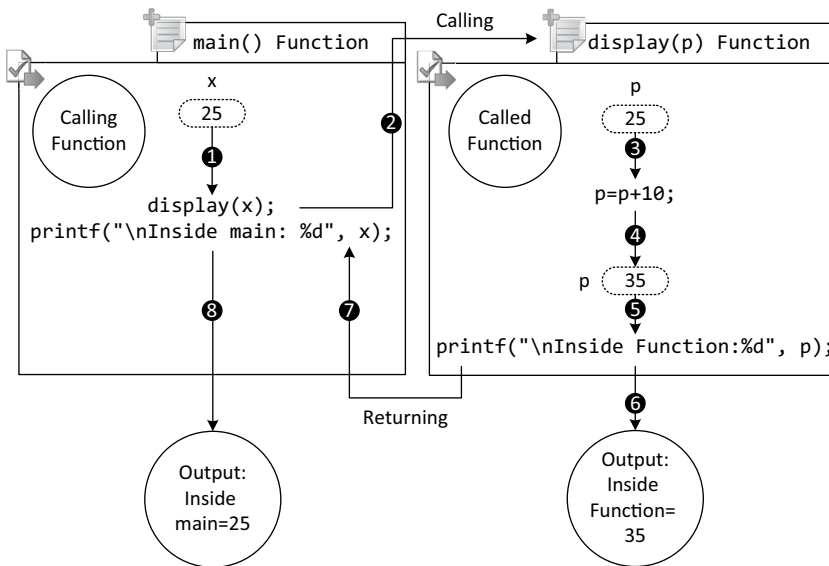
Now by examining the pass by value concept we may conclude that both variables (*p* and *x*) have different scopes so any changes made to one variable do not affect the other variable. Now let us see in the case of pass by reference what will happen.

#### 11.9.2 Pass by Reference or Address

In this method, the “address” of the actual arguments (in the calling function) is copied into the corresponding formal arguments (the parameters in the called function). Hence, the changes made to the formal arguments in the called function affect the values of the actual arguments in the calling function.

#### PROGRAM 11.19

```
#include<stdio.h>
void display(int *p)
{
 *p=*p+10;
 printf("\nInside Function: %d",*p);
}
```



**FIGURE 11.13**  
Pass by value.

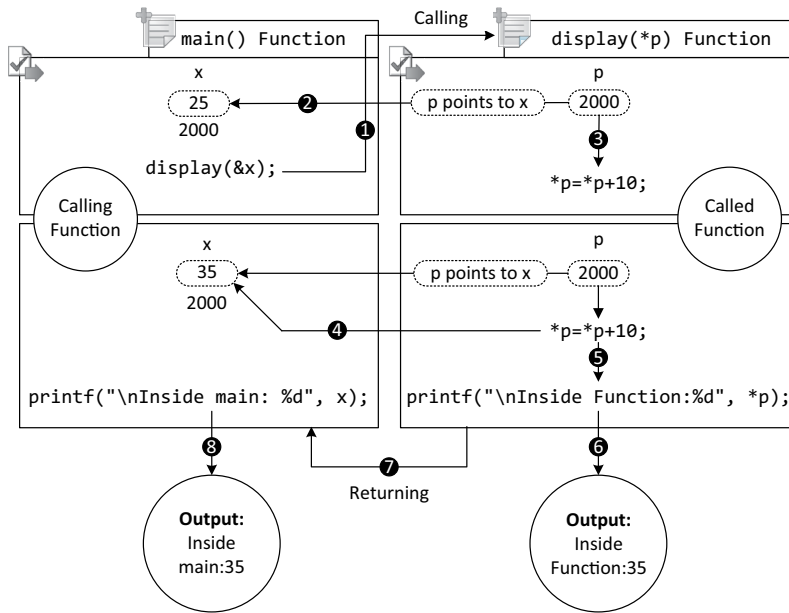
```
void main()
{
 int x=25;
 display(&x);
 printf("\nInside Main: %d",x);
}
```

This means that using these addresses we would have access to the actual arguments and hence we would be able to manipulate them. The following program illustrates this fact.

Referring to Figure 11.14 and Program 11.19, execution starts from `main()`. The first line `int x=25` allocates memory for variable `x` and stores 25 in it. ❶ The second line `display(&x)` calls the function by passing the address of `x`, which in turn assigns the address to `p`. ❷ Now, `p` points to `x`. ❸ On executing the first line (`*p=*p+10`) of the function `display()`, the value of `x` is updated to 35 because `p` points to `x`. ❹ Then the next line (`printf("\nInside Function: %d",*p)`) prints the value of `*p` which is nothing but the value of `x`, i.e., 35. ❺ Now, control returns to the `main()` function and ❻ starts executing the last `printf()` function, which prints the value of `x`, i.e., 35. So, in the pass by reference method the output of the program will be:

```
Inside function: 35
Inside Main: 35
```

If we compare both outputs (Program 11.18 and 11.19), they are seen to differ by the value of `x` inside the function `main()`. To get the same output we have to rewrite the pass by value function as follows:



**FIGURE 11.14**  
Pass by reference.

**PROGRAM 11.20**

```
#include<stdio.h>
int display(int p)
{
 p=p+10;
 printf("\nInside Function: %d",p);
 return p;
}
void main()
{
 int x=25;
 x=display(x);
 printf("\nInside Main: %d",x);
}
```

The `display()` function should return the value of `p` to the `main` function, so that `x` will be assigned by the updated value of `p`. In this case the output becomes:

```
Inside function: 35
Inside Main: 35
```

Now, the question is, if we are able to get the same output in both cases, then what is the necessity for using the pass by reference method? To answer this question, we need to solve the following simple problem.

### 11.9.2.1 Problem: Write a Program to Swap Two Numbers Using Functions

Let us first solve the problem using the *pass by value method* as shown in Program 11.21.

#### PROGRAM 11.21

```
#include<stdio.h>
void swap(int x, int y)
{
 int t;
 t = x;
 x = y;
 y = t;
 printf("\nx= %d y= %d", x, y);
}
void main()
{
 int a=10, b=20;
 swap(a,b);
 printf("\na= %d b= %d",a,b);
}
```

The output of the above program will be:

```
x=20 y=10
a=10 b=20
```

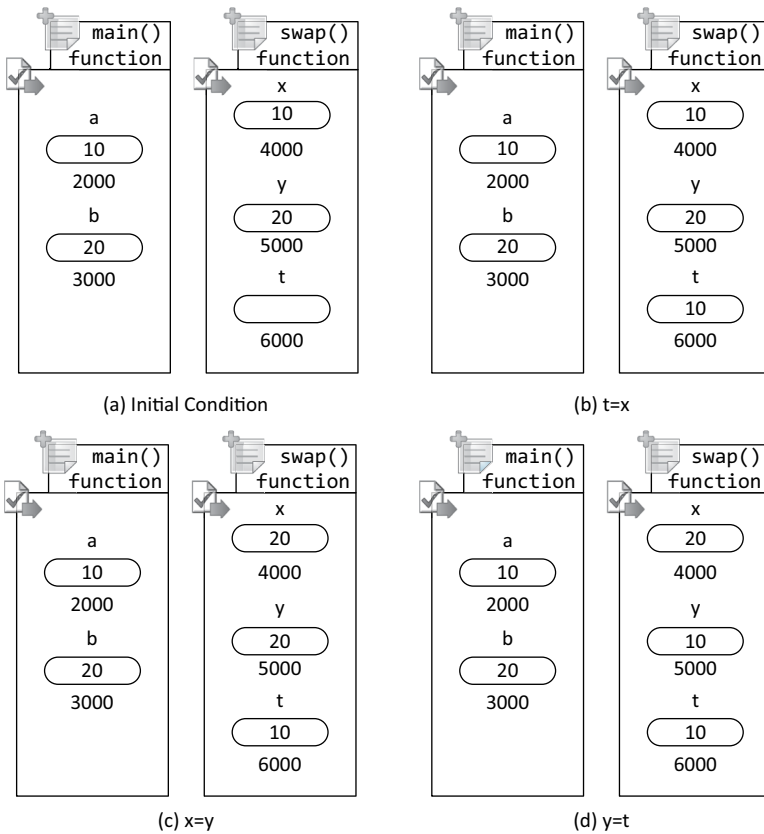
Note that the values of *a* and *b* remain unchanged even after exchanging the values of *x* and *y*. The whole process is shown in Figure 11.15, where panel (a) represents the initial state of each variable. In panel (b), the value of *x* is assigned to the temporary variable *t*, and in panel (c), the value of *y* is assigned to *x*. In the last step in panel (d), the value of *t* is assigned to *y*. That means the swapping is done on formal arguments (*x* and *y*) rather than on actual arguments (*a* and *b*).

But our problem demands the swapping of *a* and *b* rather than *x* and *y*. Again, we know that a function cannot return two values at the same time. Finally, we can say that by using the *pass by value method* this problem cannot be solved. So, we need the *pass by reference method* to solve this problem.

Let us solve the problem using the *pass by reference method* as shown in Program 11.22.

#### PROGRAM 11.22

```
#include<stdio.h>
void swap(int *x, int *y)
{
 int t;
 t = *x;
```



**FIGURE 11.15**  
Pass by value for swapping two numbers.

```

 *x = *y;
 *y = t;
 }
void main()
{
 int a = 10, b = 20;
 swap(&a, &b);
 printf ("\na = %d b = %d", a, b) ;
}

```

In this program, rather than passing the values of a and b, we have passed the addresses of both variables. To catch this address in the function, we have declared two pointers \*x and \*y in the called function. Now, as the pointer points to a and b, so any changes made on x and y will affect the actual values that are a and b. So, after executing the statements the value of the actual arguments (a and b) are swapped. Figure 11.16 shows the execution process.

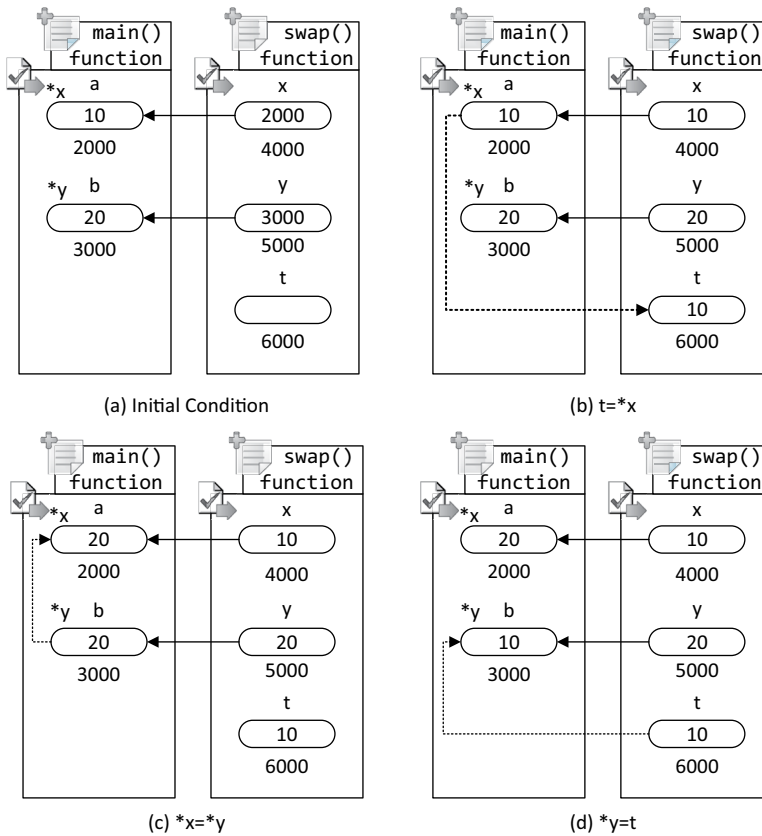


FIGURE 11.16

Pass by reference for swapping two numbers.

## 11.10 Pointers and Arrays

In this section, I would like to show you how the array and pointer are related to each other. We know that the array name is itself a *constant pointer*. That means when we print the first element's address, or we want to print the array name, both will give same output. Let us consider Program 11.23 for this purpose.

### PROGRAM 11.23

```
#include<stdio.h>
void main()
{
 int A[3]={3,6,9};
 printf("\n Address of A=%u", &A[0]);
 printf("\n Address of A=%u", A);
}
```

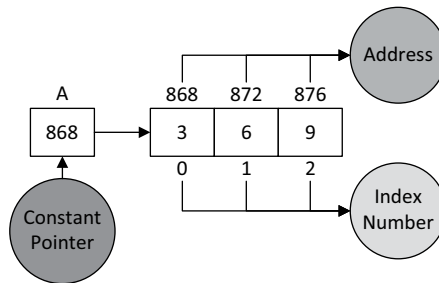
The output of the above program will be:

Address of A = 868  
Address of A = 868

According to the output, it is shown that the array name is the first address of the array. That means if we write the following statement in the above program:

```
printf("\n Value=%d", *A);
```

then the output will be 3, because the `*` operator is the *value at address operator* and shows the *value at address A*, which is 3. So, to access the other elements of the array we use `A`. The concept can be visualized as shown in Figure 11.17.



**FIGURE 11.17**  
Array representation.

According to Figure 11.17, `A` is a pointer which points to the first address of the array. Now let us consider Program 11.24 to print all the elements of the array using this pointer.

#### PROGRAM 11.24

```
#include<stdio.h>
void main()
{
 int A[3]={3,6,9};
 int i;
 printf(" The elements are: ");
 for(i=0;i<3;i++)
 printf("%d ", *(A+i));
}
```

*Output:*

The elements are: 3 6 9



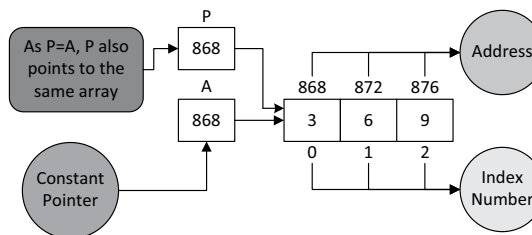
There are many exciting points associated with this concept; let us discuss some of them. From the above, we have found that the array name is a constant pointer, and we can assign the value of this pointer to any other pointer.

For example:

```
int *P;
P=A; /*Assume that A is an Integer Array*/
```

From the above assignment, Figure 11.18 can be constructed.

Now, as P points to the same array, we can print the array elements using P also. The program for this concept is shown in Program 11.25.



**FIGURE 11.18**  
Pointer P pointing to the array A.

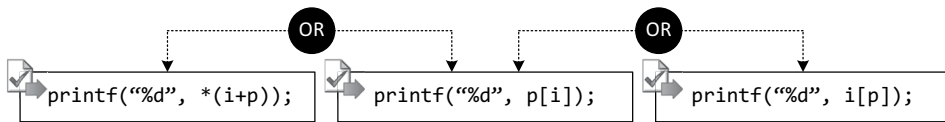
### PROGRAM 11.25

```
1. #include<stdio.h>
2. void main()
3. {
4. int A[3]={3,6,9};
5. int i;
6. int *p;
7. p=A;
8. printf(" The elements are: ");
9. for(i=0;i<3;i++)
10. printf("%d ", *(P+i));
11. }
```

*Output:*

The elements are: 3 6 9

Line 10 of Program 11.25 can take different forms as shown below; the output remains same.



The question is, how is it that `i[p]` and `p[i]` are the same? Actually, the compiler converts `p[i]` to `*(p+i)` and accordingly the output is displayed. When we supply `i[p]`, the compiler automatically converts that statement to `*(i+p)` which is the same as `*(p+i)`. Let us rewrite the above program and see the output.

### PROGRAM 11.26

```
#include<stdio.h>
void main()
{
 int A[3]={3,6,9};
 int i;
 int *p;
 p=A;
 printf("\nThe elements are: ");
 for(i=0;i<3;i++)
 printf("%d ", p[i]);
 printf("\nThe elements are: ");
 for(i=0;i<3;i++)
 printf("%d ", i[p]);
}
```

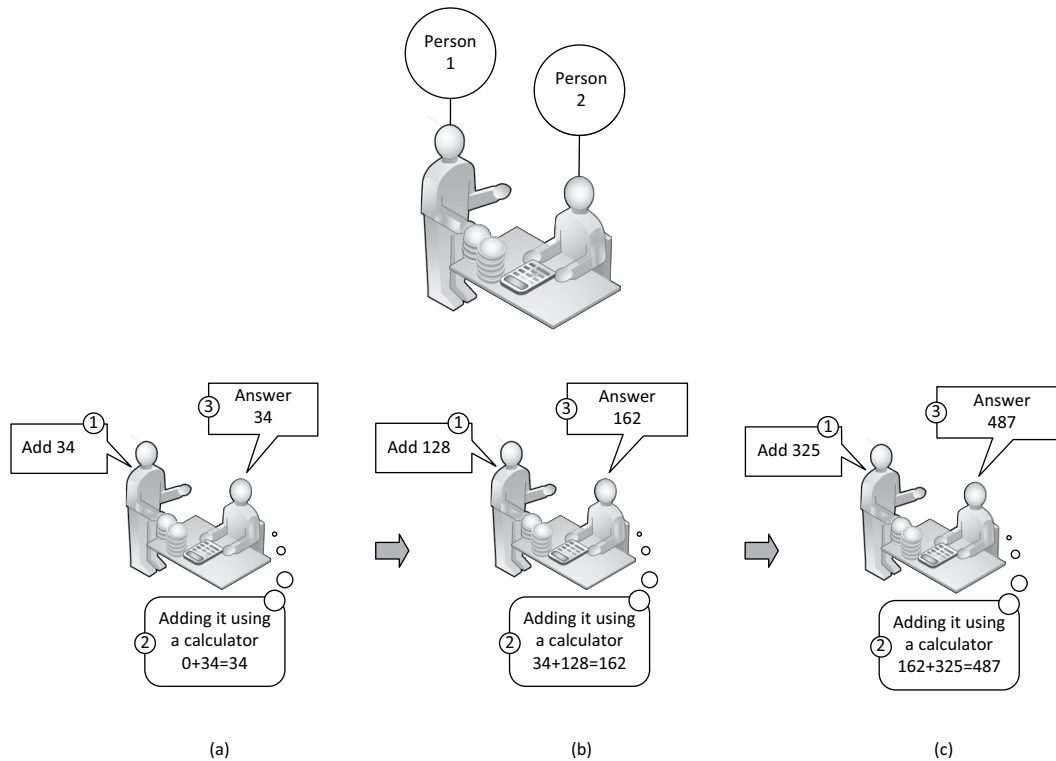
*Output:*

```
The elements are: 3 6 9
The elements are: 3 6 9
```

## 11.11 Passing Arrays to Functions

An array is passed to a function in a similar way like passing a variable. But the problem here is the array does not contain a single value. To pass an entire array to a function we need the help of a pointer. We can also pass each element of an array to a function individually. But to pass each element individually takes more overhead and control transformation. Let us first take an example to pass each element of an array individually (without using a pointer) to a function and analyze the consequences. Program 11.27 shows the code.

The program illustrates the addition of all the elements present in the array using a function. The approach is quite simple. We take a for loop and, in each iteration, we pass one element to the function. The function uses a variable `sum` which is initialized to 0. When the function receives an element, it adds it to `sum` value. Finally, we return the `sum`.

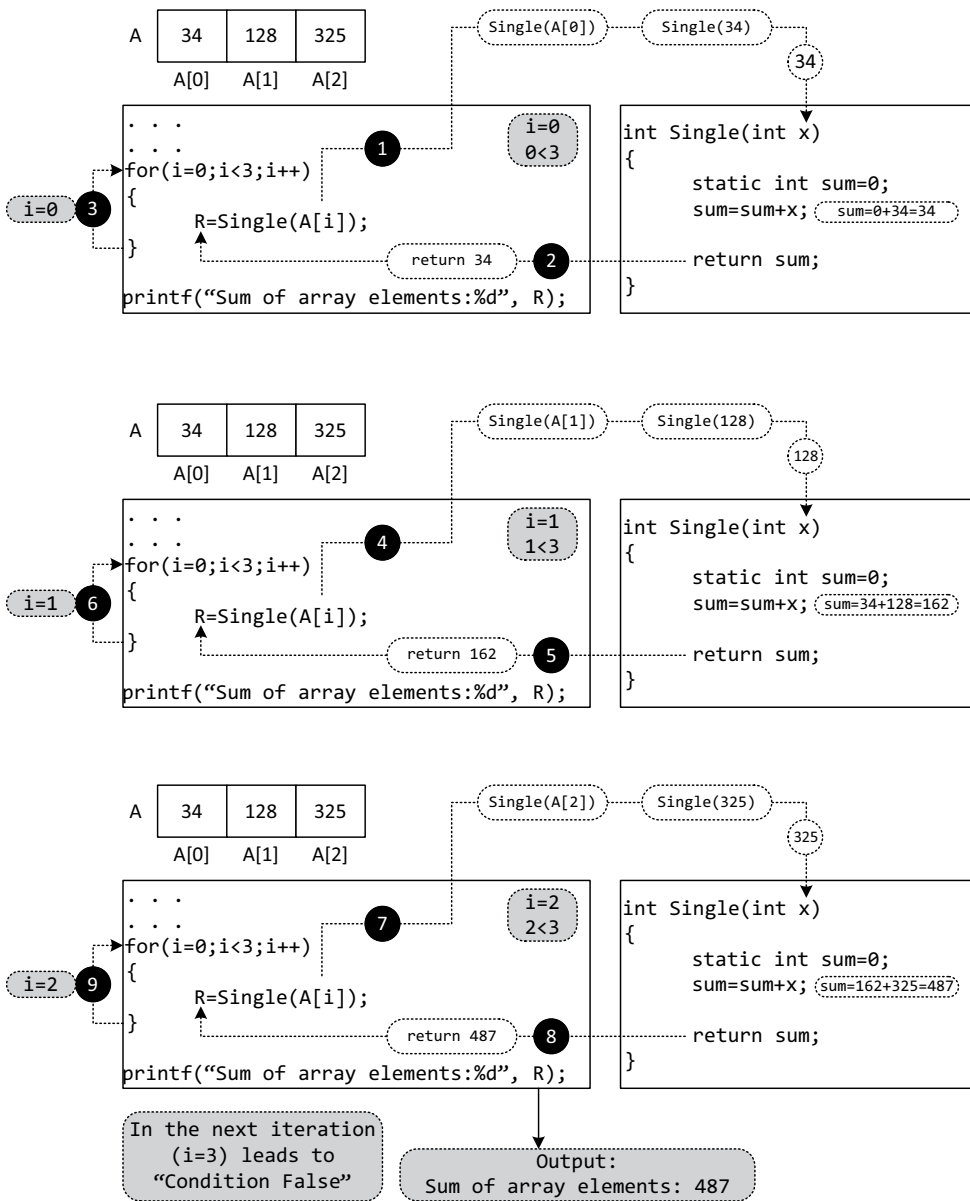


**FIGURE 11.19**  
Analogy that describes pass by value.

I would like to present you with an analogy related to this program, shown in Figure 11.19. Let there be two persons. The second person has a calculator and the first person dictates numbers one by one and the second person adds them using the calculator and speaks it out loudly. Finally, the second person returns the result to the first person.

In Figure 11.19, the first person asks the second person to add 34. The second person adds 34 to 0 and speaks out the answer, 34. Then the first person speaks out the second number (128). The second person adds it to the previous value (34) and replies with the answer 162. Finally, the second person adds the previous value 162 with 325 (the new number) and replies with the answer 487. One important point to note here is that the second person needs to remember the last result. In programming terms this can be done using a static keyword. Program 11.27 shows the complete code.

According to the execution of the Program 11.27, control will transfer from the `main()` function to the subfunction `Single()` repeatedly. Control transformation depends on the number of elements present in the array. According to the example the array contains three elements and so control must transfer three times; the execution step is shown in Figure 11.20. At  $i=0$ , the first function call takes place, passing the first element 34 to the function `Single()`, which returns 34 ( $0+34$ ). At  $i=1$ , the second function call occurs,



**FIGURE 11.20**  
Execution step for Program 11.27.

passing 128 to the function `Single()`, which returns 162 (34+128). At `i=2`, the third function call occurs, passing 325 to the function `Single()`, which returns 487 (162+325).

The other way to do the same thing is to pass the array using a reference; or we can say pass the address of the first element. This way of passing an address is called the passing by reference method. In this method the entire array is passed to the

**PROGRAM 11.27**

```
#include<stdio.h>
int Single(int x)
{
 static int sum=0;
 sum=sum+x;
 return sum;
}
void main()
{
 int A[100], R, n, i;
 printf("Enter how many elements you want: ");
 scanf("%d", &n);
 printf("Enter %d elements: ", n);
 for(i=0;i<n;i++)
 scanf("%d", &A[i]);
 for(i=0;i<n;i++)
 {
 R=Single(A[i]);
 }
 printf("Sum of array elements: %d", R);
}
```

The explanation of the above program is shown in Figure 11.20 and the output is shown below. The explanation will be easily understood if you relate the execution with respect to the analogy described in Figure 11.19.

*Output:*

```
Enter how many elements you want: 3
Enter three elements: 34 128 325
Sum of array elements: 487
```

subfunction. If we explain this using our analogy in Figure 11.19, instead of dictating the numbers, the first person may give a list of numbers to the second person and ask him to add it and speak out the result. The second person reads the list of numbers, adds them, and finally replies with the result. Let us write the complete code to implement the use of pass by reference. Program 11.28 shows this implementation.

**PROGRAM 11.28**

```
1. #include<stdio.h>
2. int Single(int *x, int N)
3. {
4. static int sum=0, i;
5. for(i=0;i<N;i++)
6. {
7. sum=sum+x[i];
8. }
9. return sum;
10. }
11. void main()
12. {
13. int A[100], R, n, i;
14. printf("Enter how many elements you want: ");
15. scanf("%d", &n);
16. printf("Enter %d elements: ", n);
17. for(i=0;i<n;i++)
18. scanf("%d", &A[i]);
19. R=Single(A, n);
20. printf("Sum of array elements: %d", R);
21. }
```

*Output:*

```
Enter how many elements you want: 3
Enter three elements: 34 128 325
Sum of array elements: 487
```

*Explanation:*

Line 19 calls the function `Single` by passing `A` and `n` as the arguments. We know that `A` contains the first address of the array elements. We can also write the same line as: `R=Single(&A[0], n)`. Here `&A[0]` also represents the first address. Line 2 declares a pointer `x` to receive the address. We can also declare this as: `int x[]`. After storing the first address, the pointer variable `x` can now point to the actual array and easily access its content by sequentially traversing through the array.

This type of problem-solving approach requires less overhead and is more flexible. Therefore, to pass an array we should always follow the concept of the pass by reference method. For greater understanding let us write another program.

**11.11.1 Write a Program to Pass an Array to a Function and Find the Largest and Smallest Numbers Present in that Array****PROGRAM 11.29**

```
#include<stdio.h>
void bigSmall(int A[], int n)
{
 int large=A[0], small=A[0], i;
 for(i=0;i<n;i++)
 {
 if(A[i]>large)
 large=A[i];
 }
 for(i=0;i<n;i++)
 {
 if(A[i]<small)
 small=A[i];
 }
 printf("The largest number=%d", large);
 printf("\nThe smallest number=%d", small);
}
void main()
{
 int arr[100],i,N;
 printf("Enter the number of elements: ");
 scanf("%d",&N);
 printf("Enter elements of the array\n");
 for(i=0;i<N;i++)
 {
 scanf("%d",&arr[i]);
 }
 bigSmall(arr, N);
}
```

**Output:**

```
Enter the number of elements: 10
Enter elements of the array
13 46 82 945 654 234 7867 21 89 23
The largest number=7867
The smallest number=13
```

---

## 11.12 Pointers and 2D Arrays

The concept of the 2D array was discussed in Chapter 10. In this section, we will discuss how a 2D array is accessed through pointers. We know that a 2D array can either be represented in row major order or column major order. The C language supports row major order representation. We also know that an array is stored by contiguous memory location. Hence, if we get the first address of the 2D array, then we can easily access the remaining elements of the array through the pointer. To understand this let us write the code for accessing the array element using a pointer. If you remember, a 2D array is a collection of 1D arrays.

We are going to write two programs for accessing the elements. Program 11.30 shows how to access the elements row-wise because each row is a 1D array. Program 11.31 shows how to access the element from the beginning to the end.

### PROGRAM 11.30

```
1. #include<stdio.h>
2. void main()
3. {
4. int i,j, *p;
5. int Mat[3][4];
6. printf("Enter 12 elements: ");
7. for(i=0;i<3;i++)
8. {
9. for(j=0;j<4;j++)
10. {
11. scanf("%d",&Mat[i][j]);
12. }
13. }
14. for(p=Mat[0];p<=&Mat[0][3];p++)
15. {
16. printf("%d ", *p);
17. }
18. printf("\n");
19. for(p=Mat[1];p<=&Mat[1][3];p++)
20. {
21. printf("%d ", *p);
22. }
23. printf("\n");
24. for(p=Mat[2];p<=&Mat[2][3];p++)
25. {
26. printf("%d ", *p);
27. }
28. }
```



*Output:*

```

Enter 12 elements: 12 23 34 45 56 67 78 89 90 98 87 76
12 23 34 45
56 67 78 89
90 98 87 76

```

*Explanation:*

In this program we take a  $3 \times 4$  matrix. Lines 4 to 13 create the 2D array and read 12 elements from the user and stores them in allocated memory. To print the contents of the array we use a pointer. Consider lines 14 to 17. Here `Mat[0]` is assigned to pointer `p`. If you recall, `Mat[0]` is the name of the first 1D array (see Section 10.5.7 and Program 10.15). Similarly, `Mat[1]` and `Mat[2]` are the names of the second and third 1D arrays, respectively. Again, an array name is a constant pointer. In line 14, `p=Mat[0]` means the address of the first element of the first row is assigned to `p`. We can also write `p=&Mat[0][0]` in place of `p=Mat[0]`. The for loop (line 14) will continue up to the last element of the first 1D array, which last element is `Mat[0][3]`, mentioned by the condition `p <= Mat[0][3]` inside the for loop. The pointer traverses through all elements of the first 1D array and prints the items present in the array using line 16. After finishing printing the first row, line 18 sends the cursor to the next line. A similar process continues for the remaining two rows of the 2D matrix.

Now consider a second program that will not print the array row-wise. We use a single for loop to print the entire contents of the 2D array directly.

**PROGRAM 11.31**

```

1. #include<stdio.h>
2. void main()
3. {
4. int i,j, *p;
5. int Mat [3] [4];
6. printf("Enter 12 elements: ");
7. for(i=0;i<3;i++)
8. {
9. for(j=0;j<4;j++)
10. {
11. scanf ("%d", &Mat [i] [j]);
12. }
13. }
14. for(p=&Mat [0] [0] ;p<=&Mat [2] [3] ;p++)
15. {
16. printf("%d ", *p);
17. }
18. }

```

*Output:*

```
Enter 12 elements: 12 23 34 45 56 67 78 90 89 123 234 345
12 23 34 45 56 67 78 90 89 123 234 345
```

*Explanation:*

Lines 14–17 print the contents of the 2D array. The pointer `p` is assigned to the first address of the 2D array (line 14) and traverses through the entire array using the condition `p<=Mat [2] [3]`, where `Mat[2][3]` is the last element of the 2D array. The `printf()` function (line 16) prints the contents one by one.

You can observe one issue with the output. The entire content is printed in a single line. But according to the concept of a 2D array, it should be printed in matrix format. I leave this for the student to solve.

**Quiz:** Rewrite Program 11.31, so that the output is printed in matrix form.

### 11.13 Pointers and Strings

A string is a character array. So, every rule we have discussed for a 1D array is also applicable to a character array. For the sake of completeness, I would like to introduce some programs related to strings and pointers in this section.


#### PROGRAM 11.32

```
#include<stdio.h>
void main()
{
 char Str[5]="C Programming Learn to code";
 printf("\n Address of Str=%u", &Str[0]);
 printf("\n Address of Str=%u",Str);
}
```

The output of the above program will be:

```
Address of Str=1148753859
Address of Str=1148753859
```

Both outputs are the same, which means the name of the string is a pointer to the character array. If we write the following statement in the above program:

```
 printf("\n First character=%c", *Str);
```

then the output will be `C`, because the `*` operator is the *value at address operator* and shows the *value at address* `Str`, which is `C`. So, to access other elements of the array we can use `Str`.

### 11.13.1 Passing a String to a Function

We can pass a string similarly to the way we pass a 1D array by using the pass by reference method. For example, if your string name is `Str`, then we either pass `Str` directly, or we can pass `&Str[0]`, which indicates the address of the first character of the string. Inside the function, we use a character pointer to catch the address. After receiving the address, the character pointer can traverse through the string to process or manipulate it.

Program 11.33 writes a function to reverse a string. We have already shown the process of reversing a string in Program 10.26. Here, we will show the string passing method.

### 11.13.2 Write a Program to Reverse a String Using a Function

#### PROGRAM 11.33

```
1. #include<stdio.h>
2. #include<conio.h>
3. void stringReverse(char *p)
4. {
5. int i, j, len=0;
6. char rev[50];
7. i=0;
8. while(p[i]!=NULL)
9. {
10. len=len+1;
11. i=i+1;
12. }
13. for(i=0,j=len-1;i<len;i++,j--)
14. {
15. rev[i]=p[j];
16. }
17. rev[i]=NULL;
18. printf("\nThe original string is: %s ",p);
19. printf("\nThe reverse of the string is: %s",rev);
20. }
21. void main()
22. {
23. char str[30];
24. printf("Enter a string: ");
25. gets(str);
26. stringReverse(str);
27. }
```

*Output:*

```
Enter a string: C Programming Learn to Code
The original string is: C Programming Learn to Code
The reverse of the string is: edoC ot nraeL gnimmargorP C
```

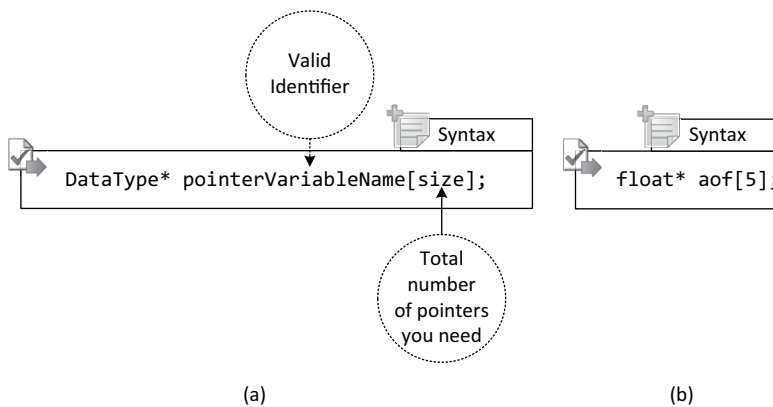
*Explanation:*

Execution starts from `main()`. Line 26 calls the function `stringReverse(str)` by passing `str` as the argument. `str` is a constant pointer because it is the name of the character array. We can also pass `&str[0]` as an argument in place of `str`, because `&str[0]` also indicates the address of the first element of the string. The function declaration starts from line 3. We have used a pointer `p` to receive the address passed by the `main()` function and obviously `p` must be a character pointer. Now, after `p` points to the string `str` we can access each character of the string by traversing through it. Lines 8 to 12 are used to find the length of the string. Lines 13 to 17 are used to read the characters from the string and copy them to `rev[50]` in reverse order. Finally, lines 18 and 19 print the string.

### 11.14 An Array of Pointers

So far, we know about character arrays, integer arrays, and float arrays. In this section, we will introduce a new array that can hold many pointers; we call it an array of pointers. The declaration takes the form shown in Figure 11.21a.

Before declaring an array of pointers, we should know how many pointers there are and their data type. One example is shown in Figure 11.21b, which declares five pointers of type `float`. Each pointer will hold the address of a variable of `float` type. Let us write a program to show how to use an array of pointers (Program 11.34).



**FIGURE 11.21** Syntax and example of an array of pointers.

## PROGRAM 11.34

```

1. #include<stdio.h>
2. void main()
3. {
4. float F[]={2.4, 6.9, 34.5, 56.23, 1.7};
5. float* aofp[5];
6. int i;
7. /* Assigning address to aofp*/
8. for(i=0;i<5;i++)
9. {
10. aofp[i]=&F[i];
11. }
12. /*Printing the array F through aofp*/
13. printf("Float array contains: ");
14. for(i=0;i<5;i++)
15. {
16. printf("%f ", *aofp[i]);
17. }
18. }

```

*Output:*

Float array contains: 2.400000 6.900000 34.500000 56.230000 1.700000

*Explanation:*

Line 4 declares a float array of five elements. The compiler allocates memory blocks for it as shown in Figure 11.22. Line 5 declares an array of pointers that can store five addresses, which must point to float variables (Figure 11.22b).

Lines 8 to 11 assign the addresses of float array F to an array of pointers aofp in a sequence. Similarly, aofp[0] is assigned to &F[0], aofp[1] is assigned to &F[1], and so on. After assignment each pointer in aofp will point to individual elements in F, as shown in Figure 11.22c. Finally, we print the elements of F through aofp pointers using lines 14 to 17.

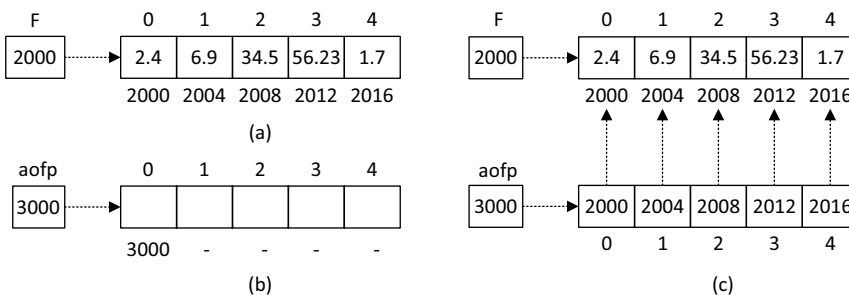
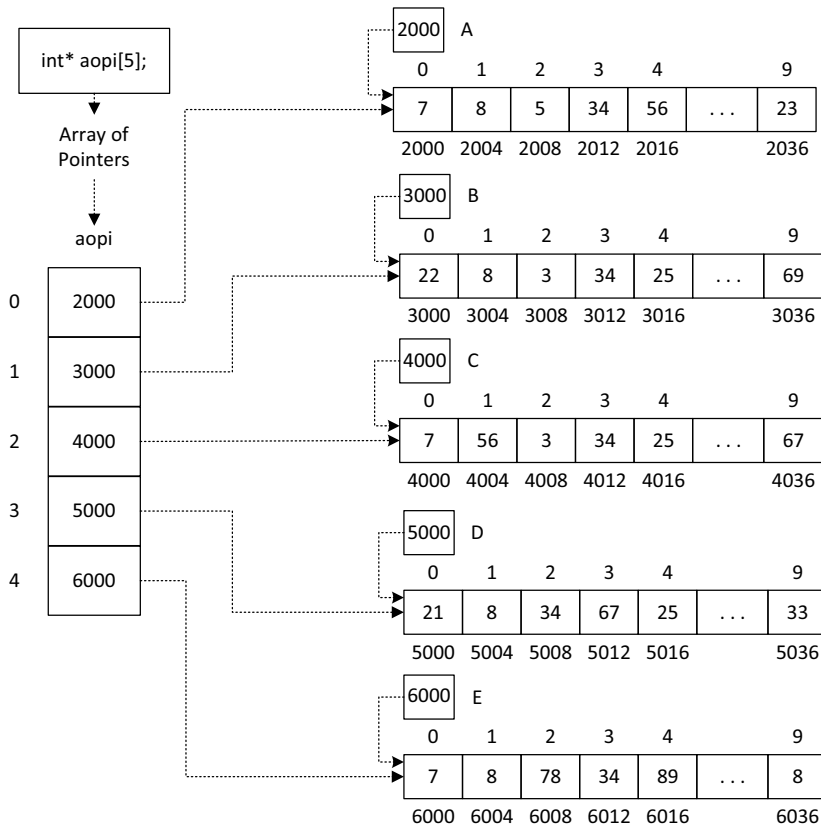


FIGURE 11.22

Illustration of Program 11.34.



**FIGURE 11.23**  
Array of pointers to different arrays.

We can extend this concept and assign the addresses of multiple arrays to arrays of pointers. Say you have five arrays and each array contains ten elements. We can declare an array of pointers of size 5 and assign the address of each array to these pointers. Later on, we can access these arrays through this array of pointers. The overall assignment is as shown in Figure 11.23.

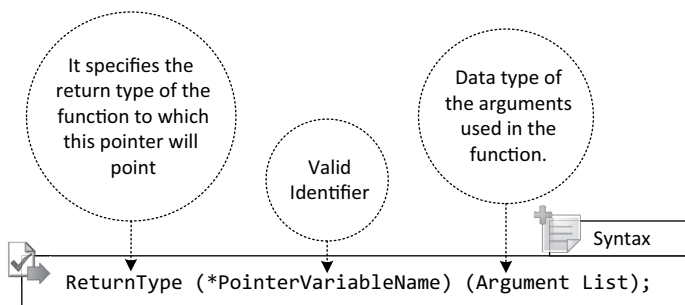
Students are encouraged to implement the above concept using C code. An array of pointers can also be created for strings. Assume that there are several strings, and each is accessed through the array of pointers. In this case, the array of pointers must be declared as character type: `char* aopc[size]`. There are several other uses of this technique too. The scope of this book does not allow us to explain all the details. Students are encouraged to explore more on this topic.

### 11.15 Pointers to Functions

We have seen pointers that store the addresses of integers; let us name them: a pointer to an integer. Similarly, the pointer which points to a float or character, we name: a pointer to a float or a pointer to a character, respectively. The essential rule in pointer declaration is: the data type of the pointer and the variable to which it points must be the same.

In this section, we will introduce another pointer that points to a function and which is called a pointer to a function. Now, how is the rule discussed above applicable to the declaration of a pointer to a function? Every function has a signature. The signature of a function specifies three things: (1) the name of the function, (2) the type and the number of arguments, and (3) the return type of the function. Hence, a pointer to function declaration requires knowledge of the signature of the function. The general syntax for declaring a pointer to a function is shown in Figure 11.24.

The syntax purely depends on the signature of the function. As shown, the return type and the argument list will be borrowed from the function to which the pointer is going to point. Let us take an example to explain this in detail. Suppose we have a function that finds the largest number among three numbers as shown in Figure 11.25.



**FIGURE 11.24**  
Syntax for declaring a pointer to a function.

Function

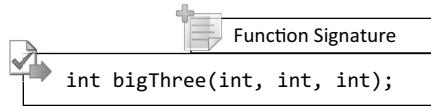
```

int
{
 if(x>y)
 {
 if (x>z)
 return x;
 else
 return z;
 }
 else
 {
 if(y>z)
 return y;
 else
 return z;
 }
}

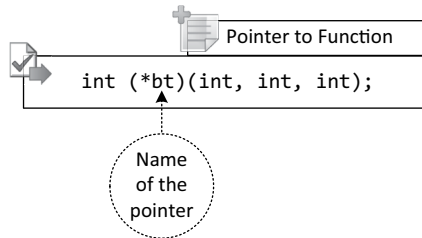
```

**FIGURE 11.25**  
A function to find the largest number among three numbers.

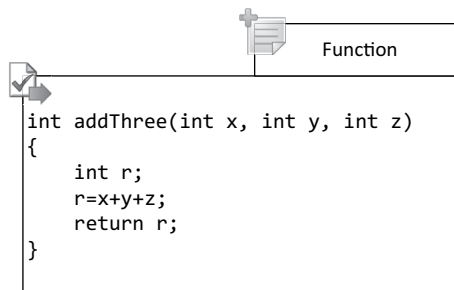
As discussed above, the function signature of the above function is shown as:



Now, we can easily declare a pointer to this function.



Here, `bt` is a user-defined pointer name and is declared to point to a function which takes three integer arguments and returns one integer value. That means it points to not only the `bigThree()` function declared above, but also any function that has the same signature. Suppose we have another function that adds three numbers and returns the result. The function code is shown below. We can also use the above pointer to point to `addThree()` function too because it has the same signature.



Program 11.35 shows the way we use the pointer to function concept.

### PROGRAM 11.35

```
1.#include<stdio.h>
2.int bigThree(int x, int y, int z)
3.{
4. if(x>y)
5. {
6. if (x>z)
7. return x;
8. else
9. return z;
10. }
```



```

11. else
12. {
13. if(y>z)
14. return y;
15. else
16. return z;
16. }
18.}
19.void main()
20.{
21. int a, b, c, big;
22. int (*bt)(int, int, int);
23. printf("Enter three numbers: ");
24. scanf("%d%d%d", &a, &b, &c);
25. bt=&bigThree;
26. big=bt(a,b,c);
27. printf("The biggest number is: %d", big);
28.}

```

*Output:*

```

Enter three numbers: 25 46 34
The biggest number is: 46

```

*Explanation:*

Execution starts from `main()`. Line 22 declares the pointer to function variable `bt`. Line 25 assigns the address of the function to `bt` and finally we call the function through `bt` in line 26 by passing three arguments `a, b, c`. After the function executes, the result is returned to the `main()` function which is received in the variable `big`. Finally, line 27 prints the value of `big` which is the largest number among three numbers.

Program 11.36 will show you that `bt` can assign the addresses of both functions declared above: `bigThree()` and `addThree()`. The program is self-explanatory.

## 11.16 Review Questions

### 11.16.1 Objective Questions

1. A \_\_\_\_\_ is a special variable that stores the address of another variable.
2. The \_\_\_\_\_ operator is used to declare a pointer, and the \_\_\_\_\_ operator is used to find a variable's address.
3. If `p` is a pointer which points to a variable `x`, then `*p` represents the value of `x`. True/false?

**PROGRAM 11.36**

```
#include<stdio.h>
int bigThree (int x, int y, int z)
{
 if (x > y)
 {
 if (x > z)
 return x;
 else
 return z;
 }
 else
 {
 if (y > z)
 return y;
 else
 return z;
 }
}
int addThree(int x, int y, int z)
{
 int r;
 r=x+y+z;
 return r;
}
void main()
{
 int a, b, c, big, sum;
 int (*bt) (int, int, int);
 printf ("Enter three numbers: ");
 scanf ("%d%d%d", &a, &b, &c);
 bt = &bigThree;
 big = bt (a, b, c);
 printf ("\nThe biggest number is: %d", big);
 bt = &addThree;
 sum = bt (a, b, c);
 printf ("\nThe addition result is: %d", sum);
}
```

*Output:*

Enter three numbers: 56 78 32

The biggest number is: 78

The addition result is: 166

4. If `p` is a pointer which points to a variable `x`, then `*p` and `*(&x)` refer to the value of `x`. True/false?
5. If `p` is a pointer which points to a variable `x`, then printing the value of `p` and `&x` produces the same output. True/false?
6. If a pointer `p` points to another pointer `q`, and pointer `q` points to a variable `x`, then which one is a double pointer?
7. If a pointer `p` points to another pointer `q`, and pointer `q` points to a variable `x`, what does `*(&p)` refer to?
8. Let `A` be an array containing three elements: 6, 9, and 25. If we print the value of `*A`, what is the output?
9. Write the syntax for declaring an array of pointers.
10. Write the syntax for declaring a pointer to a function.

### 11.16.2 Subjective Questions

1. What is a null pointer? Explain the uses of a null pointer.
2. What is a void pointer? How is it different from other pointers? How do we use a void pointer?
3. How do we declare a triple pointer? Write a program to show how to use a triple pointer.
4. Differentiate between a constant pointer and a pointer to a constant.
5. Describe pointer arithmetic.
6. Describe the need for pass by reference with a suitable example. Write a program to show the difference between pass by value and pass by reference.
7. How do we pass an array to a function? Compare pass by value and pass by reference with respect to array passing.
8. How do we pass a 2D array to a function. Write a program to add two matrices by passing both matrices to the function.
9. What is an array of pointers? Assume that you have ten strings, write a program to create an array of pointers that points to these ten strings and print the strings.
10. What is a pointer to a function? How do we declare a pointer to a function? Write a suitable programming example to explain how it works.

### 11.16.3 Programming Exercises

- 1 Find the error or output for the following C code.

```
a.#include<stdio.h>
void main()
{
 int A[3]={3,6,9};
 int i;
 int *p;
 p=A;
```

```
 for(i=0;i<3;i++)
 printf("%d %d %d %d ", p[i], *(p+i), i[p]);
}
```

```
b.#include<stdio.h>
void main()
{
 int A[3]={3,6,9};
 int i;
 int *p;
 p=A;
 printf("%d", *p);
 p=p+1;
 printf("%d", *p);
 p=p+1;
 printf("%d", *p);
}
```

```
c.#include<stdio.h>
void main()
{
 int A[3]={3,6,9};
 int i;
 int *p;
 p=A;
 printf("\n%d", *p+2);
 printf("\n%d", *(p+2));
}
```

```
d.#include<stdio.h>
void main()
{
 int A[3]={3,6,9};
 int i;
 printf("\n%d", *A+2);
 printf("\n%d", *(A+2));
}
```

```
e.#include<stdio.h>
void main()
{
 int A[3]={3,6,9};
 int i;
 printf("%d", *A);
}
```

```
A=A+1;
printf("%d", *A);
A=A+1;
printf("%d", *A);
}
```

```
f.#include<stdio.h>
void main()
{
 int const x=25;
 int *p;
 p=&x;
 printf("%d", *p);
}
```

```
g.#include<stdio.h>
void main()
{
 float A[5]={1.4, 5.8, 2.3, 6.4, 9.1};
 float *p, *q;
 int size;
 p=&A[2];
 q=&A[2];
 size=p>=q;
 printf("%d ", size);
}
```

```
h.#include<stdio.h>
void main()
{
 int A[5]={4, 8, 3, 64, 91};
 int B[7]={99, 66, 33, 64, 22, 66, 55};
 int *p, *q;
 int size;
 p=&A[2];
 q=&B[2];
 q=p+1;
 printf("%d ", *q);
}
```

2. Write a program to sort an array in descending order by passing the array to a function.
3. Write a program to compare two strings by passing them to a function. The function will return 1 if both the strings are same, else it returns 0.

# 12

## Structures and Unions

### 12.1 Introduction

Before we start this chapter, let us pick up some real-life problems and think about whether we can solve these problems using the programming technique learned till now.

**Problem 1:** Write a program to add together two times. We represent time with three parameters: hour, minute, and second. For example, time  $T1 = 2:35:40$ , time  $T2 = 5:10:35$ . Find time  $T3$  such that  $T3 = T1 + T2$  (see Figure 12.1).

**Problem 2:** Write a program to add the height of two persons, given in feet and inches. For example, height  $H1 = 5'10''$ , height  $H2 = 5'8''$ . Find the height  $H3$  such that  $H3 = H1 + H2$  (see Figure 12.2).

Thinking about Problem 1, do we have any data type available to represent the time as a single entity? I mean, can we write `int t1` to store the hour, minute, and second simultaneously? The answer is no. Similarly, Problem 2 cannot be solved using the existing data type that we have been using. Hence, we need a different data type for this, and yes the C language provides us with the features to define the new data type for these specific problems. A structure is one such technique that helps us in solving the problem discussed above.

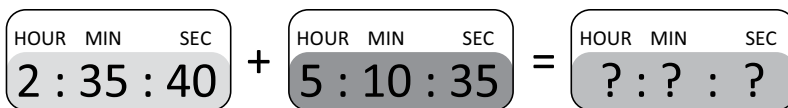


FIGURE 12.1  
Problem 1 visualization.

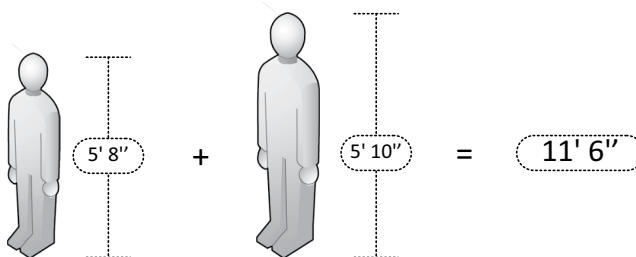
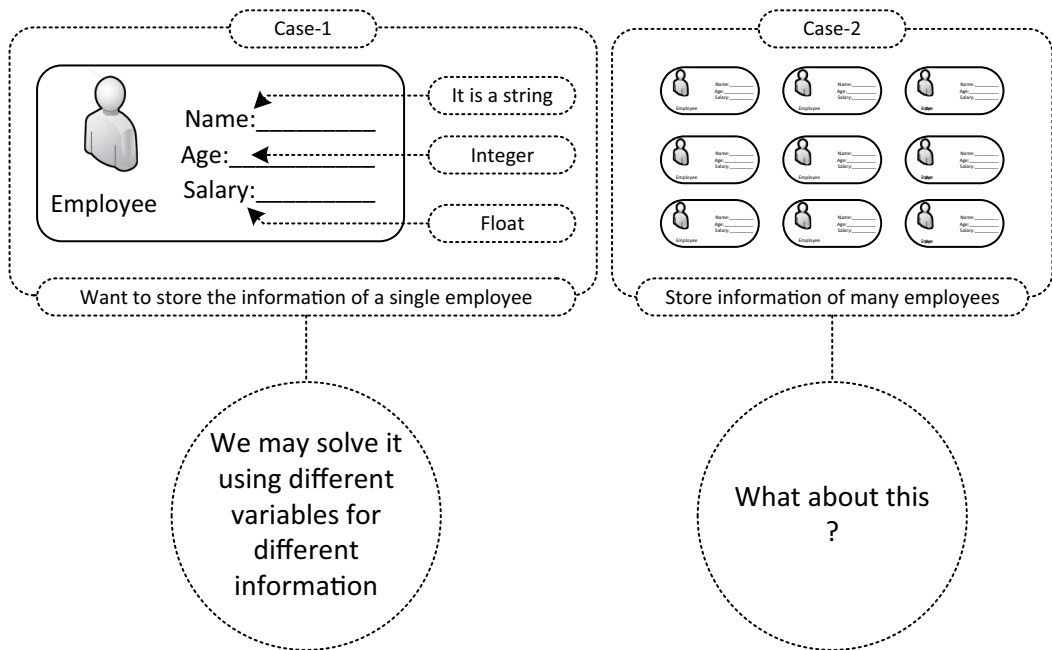


FIGURE 12.2  
Problem 2 visualization.



**FIGURE 12.3**  
Visualization of information storage problem.

Let us think in another way. We have already used an array in Chapter 9, which represented a group of data items having the same data type. But, if you have data with different data types, where do you store it? For example, you want to store the information of an employee (Case 1 of Figure 12.3). Employee information consists of a name, age, and salary. The name of the employee is a string, age is an integer, and the salary is of float type. Hence, we cannot use an array to store these pieces of information. We can store it using three different types of variables: `char Name [50]` to store the name, `int age` to store the age, and `float salary` to store the salary. But, assume that there are many employees whose information you want to store (Case 2 of Figure 12.3). Then for each employee, you need to declare separate variables, and managing them is quite tricky. The visualization of this situation is shown in Figure 12.3.

From the discussion, we conclude that the basic data types cannot solve these problems. We need something special. C provides a better solution for the above problems, what is known as a *structure*. A structure is a user-defined data type and can group one or more variables of different data types under a single name. Similarly, another user-defined data type is union. This is also used for the same purpose; it is like a structure but with a little difference as discussed in Section 12.12. This entire chapter is dedicated to a discussion of these user-defined data types.

After completing this chapter, the student will be:

- Able to define a structure, a union, and the difference between them;
- Able to declare and use structure and union variables for information storing and processing;

- Able to describe an array of structures, nested structures, and other user-defined types like typedef;
- Able to understand the way a pointer is used to access the members of a structure;
- Able to define pointers inside a structure as a member;
- Able to define bitfields and enumerations.

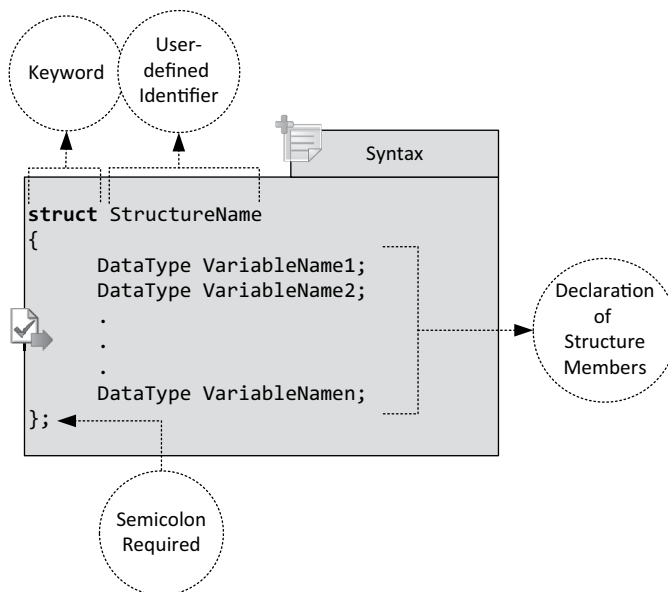
## 12.2 Declaring a Structure

A structure can be defined as a collection of one or more variables of the same or different data types, grouped together under a single name. Like other data types, a structure must be declared before it is used inside a program. C provides two different ways to declare a structure: tagged structure and structure declaration using typedef.

A structure can be defined as a collection of one or more variables of the same or different data types, grouped together under a single name.

### 12.2.1 Tagged Structure Declaration

To declare a structure using this method, we use the keyword *struct*. In this declaration, the *struct* keyword is followed by a structure name (tag) and the general syntax for declaring the structure is shown in Figure 12.4.

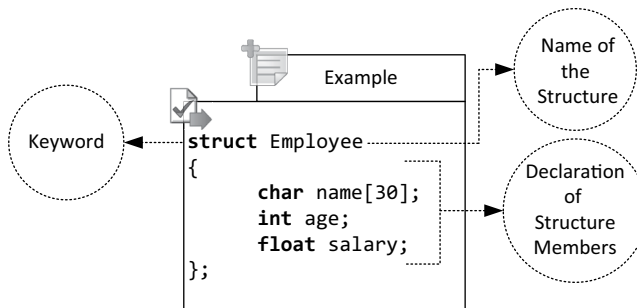


**FIGURE 12.4**  
Structure declaration syntax.



## NOTE

Structure members should not be initialized inside the structure declaration.



**FIGURE 12.5**  
Structure declaration example.

Figure 12.5 shows an example of a structure declaration. Here, `Employee` is a structure with three data members. After this declaration, `Employee` acts as a user-defined data type, and we can declare any number of variables using this data type. The following section will discuss how to declare variables using this new data type.

### 12.2.2 Structure Declaration Using typedef

To declare a structure using this method, we use the keyword *typedef*. In this declaration, the *typedef* keyword is followed by a *struct* keyword and the name of the structure is declared after the closing braces of structure declaration and before the semicolon. The general syntax for declaring the structure is shown in Figure 12.6.

Figure 12.7 shows an example that uses the syntax described in Figure 12.6.

#### Points to remember:

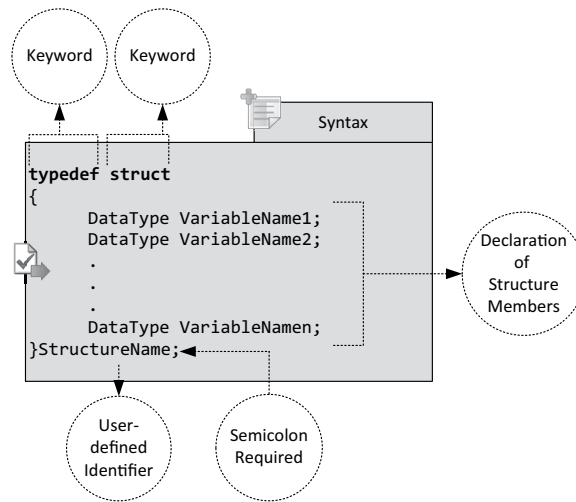
- All member names within a particular structure must be different;
- The individual members of a structure may be any of the common data types such as `int`, `float`, pointers, array, or even other structures;
- No memory space is allocated when the structure is declared.

### 12.2.3 Declaring Structure Variables

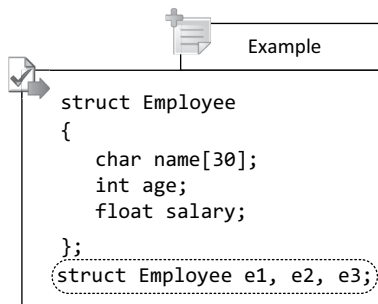
Structure variables can be declared in two ways: either using the name of the structure, or after the closing braces of the structure declaration and before the semicolon.

#### 12.2.3.1 Declaring Structure Variables Using the Structure Name

In this method, a structure variable can be declared using the structure name. First the structure is declared and then the variables for the structure are declared as follows:

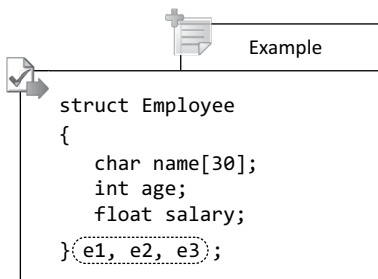


**FIGURE 12.6**  
Structure declaration using typedef.

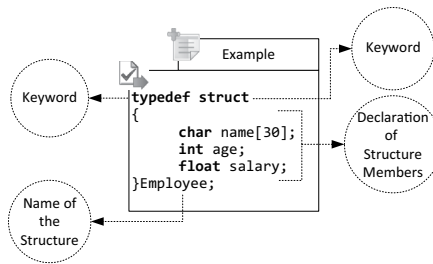


### 12.2.3.2 Declaring Structure Variables after the Closing Braces

In this method, all the structure variables can be specified after the closing braces of the structure and before the semicolon as shown below:



Once the declaration is over, we get three different variables and each variable consists of three data members, i.e., name, age, and salary. Now we can say that e1, e2, and e3 are variables of Employee type.



**FIGURE 12.7**  
Example of Structure using the keyword typedef.

### Points to remember:

- Memory spaces are allocated for each variable of the structure after it is declared. For example, separate memory spaces are allocated for the Employee variables e1, e2, and e3.
- Generally, the structure declaration is made before the main function, and the structure variables for the structure are declared inside the main function.
- The size of each structure variable is the addition of the size of each member variable. See Figure 12.8.

Consider the following programming example to see the size of the structure variable.

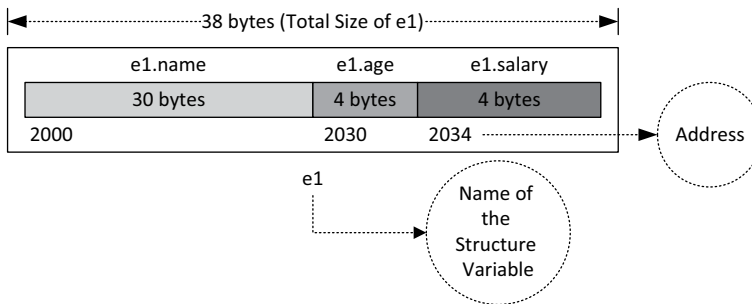
### PROGRAM 12.1

```
#include<stdio.h>
struct Employee
{
 char name[30];
 int age;
 float salary;
};
void main()
{
 struct Employee e1;
 printf("Size of e1 = %d ", sizeof(e1));
}
```

*Output:*  
Size of e1 = 38

*Explanation:*

In the above program, we have declared the structure above the main function and inside the latter we have declared the structure variable (e1) inside the main function. The same program can be written in many ways. Program 12.2 shows the declaration of a structure inside the `main()` function.



**FIGURE 12.8**  
Memory representation of `e1`.

### 12.3 Initializing a Structure

A structure is a collection of structure members, and each member may be of different data types. So for initializing a structure, we need to specify values for each member. The initialization of a structure is similar to array initialization.

Let us consider the following structure:

Example

```

struct Employee
{
 char name[30];
 int age;
 float salary;
};

```

The variable of this structure can be initialized during its declaration as:

Initialization

```

struct Employee e1={ "Aakash", 32, 45345.67 };
struct Employee e3={ "Ramesh", 35, 47348.77 };
struct Employee e3={ "Nilesh", 25, 23374.82 };

```

According to the above initialization each structure variable (`e1`, `e2`, or `e3`) is initialized with three data members, because each structure variable (`e1`, `e2`, `e3`) contains three member variables (`name`, `age`, `salary`).

## PROGRAM 12.2

```

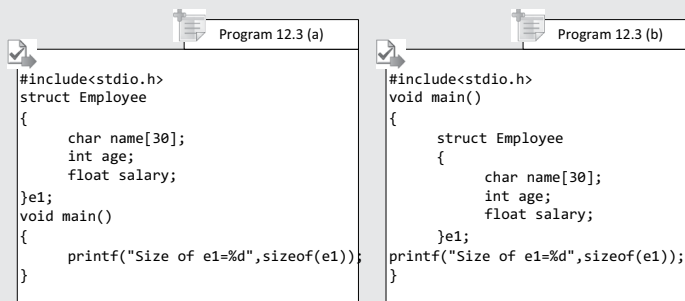
#include<stdio.h>
void main()
{
 struct Employee
 {
 char name[30];
 int age;
 float salary;
 } ;
 struct Employee e1;
 printf("Size of e1 = %d ", sizeof(e1));
}

```

*Output:*

Size of e1 = 38

Another way of writing the same program may take the forms shown in Program 12.3a and Program 12.3b. The purpose of writing the same program in different forms is to show that you can code your program in any of the forms and that the compiler does not show any errors.



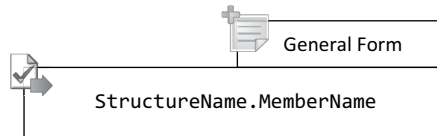
The outputs of the above Programs 12.3a and b are also the same as Programs 12.1 and 12.2. We are familiar with the declaration of structures and structure variables. Another way of declaring a structure is available: using the *typedef* keyword. In Section 12.8, we will discuss the use of the typedef keyword in structure declaration.

## 12.4 Accessing Structure Members

C provides two different operators to access the member of a structure independently: the *dot* (.) operator and the *arrow* (->) operator. These are some of the special operators provided by C. In the following section, we discuss the dot (.) operator for accessing the data member. The arrow (->) operator is discussed in Section 12.9.

### 12.4.1 Accessing Members Using the dot (.) Operator

To access the members of a structure using the dot (.) operator takes the following form:



Let us consider the example shown in Figure 12.9. To access each individual member of this structure we need a dot (.) operator along with the structure variable name.

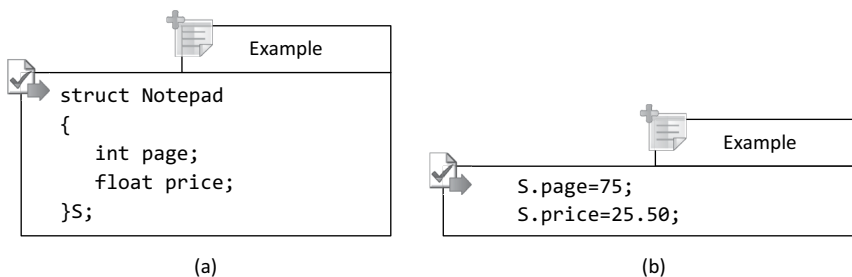
The above statements involve the conceptual representation of a structure variable S as shown in Figure 12.10.

Now we are at the stage of writing a complete program using the concept of structure. Program 12.4 declares a structure notepad with two data members: page and price. The program will read the values for these data members and print the detail of this notepad.

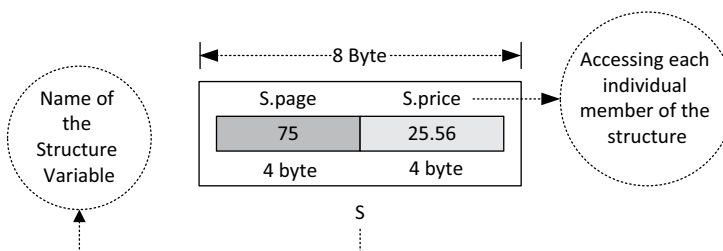
#### NOTE

The name of the structure (notepad) is not a variable name. It is just a user defined data type with which we can declare many variables. So, the following statement shows an error:

```
notepad. page = 75; /*ERROR*/
```



**FIGURE 12.9**  
Member accessing example using a dot operator.



**FIGURE 12.10**  
Conceptual representation of structure variable S.

**PROGRAM 12.4**

```
#include<stdio.h>
struct notepad
{
 int page;
 float price;
};
void main()
{
 S.page=75;
 S.price=25.56;
 printf("Number of pages: %d", S.page);
 printf("\nPrice: %f", S.price);
}
```

*Output:*

```
Number of pages: 75
Price: 25.56
```

If we want to read the value for each individual member then we can also use the `scanf()` function for this purpose. The following program illustrates this concept.

**PROGRAM 12.5**

```
#include<stdio.h>
struct notepad
{
 int page;
 float price;
};
void main()
{
 struct notepad S; /* Declaration of Structure Variable*/
 /*Reading the values for structure members*/
 printf("Enter the number of pages: ");
 scanf("%d", &S.page);
 printf("Enter the price: ");
 scanf("%f", &S.price);
 /* Printing the values stored in each structure member*/
 printf("Number of pages: %d", S.page);
 printf("\nPrice: %f", S.price);
}
```

*Output:*

```
Enter the number of pages: 35
Enter the price: 20.35
Number of pages: 35
Price: 20.35
```

---

## 12.5 Learn to Code Examples

In this section, we discuss some of the problems that can only be solved by using the concept of a structure. Let us begin with Problem 2 that was discussed in Section 12.1: the addition of the heights of two people.

**Write a program to declare a structure named HEIGHT with two data members: feet and inches. Write the code to add both heights.**

### PROGRAM 12.6

```
#include<stdio.h>
#include<conio.h>
struct height
{
 int feet;
 int inches;
};
void main()
{
 struct height h1, h2, h3;
 printf("Enter the height in feet and inches for height1: ");
 scanf("%d%d", &h1.feet, &h1.inches);
 printf("Enter the height in feet and inches for height2: ");
 scanf("%d%d", &h2.feet, &h2.inches);
 /* Addition of the height*/
 h3.feet = h1.feet + h2.feet;
 h3.inches = h1.inches + h2.inches;
 if(h3.inches >=12)
 {
 h3.feet=h3.feet+1;
 h3.inches =h3.inches-12;
 }
 printf("Addition of the Height is: %d feet %d inches",
 h3.feet, h3.inches);
}
```



*Output:*

```
Enter the height in feet and inches for height1: 7 9
Enter the height in feet and inches for height2: 6 8
Addition of the Height is: 14 feet 5 inches
```

**Write a program to declare a structure named POINT with two data members: x-coordinate and y-coordinate. Write the code to find the distance between the points.**

**PROGRAM 12.7**

```
#include<stdio.h>
#include<math.h>
struct point
{
 int xco;
 int yco;
};
void main()
{
 struct point p, q;
 int x, y, dist;
 printf("Enter x and y co-ordinate value for first point: ");
 scanf("%d%d", &p.xco, &p.yco);
 printf("Enter x and y co-ordinate value for second point: ");
 scanf("%d%d", &q.xco, &q.yco);
 x=q.xco-p.xco;
 y=q.yco-p.yco;
 dist= sqrt(x*x+y*y);
 printf("\nDistance between the points: %d", dist);
}
```

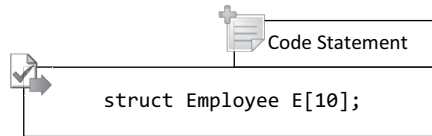
*Output:*

```
Enter x and y co-ordinate value for first point: 4 6
Enter x and y co-ordinate value for second point: 2 8
Distance between the points: 2
```

**12.6 Arrays of Structures**

Like an array of integers or an array of floats, an array of structures can also be declared. The need for an array of structures arises when we need to consider storing multiple entities. Suppose we want to store information about ten employees, then we need an array of structures.

In C, you can declare an array of structures by preceding the array name with the structure name. For instance, given a structure with the tag name of `Employee`, the following statement declares an array, called `E`, of `struct Employee`. The array has ten elements, each element being a single instance of `struct Employee`.



Let us take an example:

**Write a program to declare a structure `emp_info` with the following data members: `Emp_id`, `Name`. Write the code for storing and displaying the details of `N` employees.**

### PROGRAM 12.8

```
#include <stdio.h>
#include <conio.h>
struct emp_info
{
 int emp_id;
 char nm[50];
};
void main()
{
 struct emp_info emp[10];
 int i,n;
 printf("Enter no of employee info to store: ");
 scanf("%d", &n);
 for(i=0;i<n;i++)
 {
 printf("\n\t Enter Employee ID : ");
 scanf("%d",&emp[i].emp_id);
 printf("\n\t Enter Employee Name : ");
 scanf("%s",emp[i].nm);
 }
 printf("\nDetail of employee");
 printf("\n-----");
 for(i=0;i<n;i++)
 {
 printf("\n\t Employee ID : %d",emp[i].emp_id);
 printf("\n\t Employee Name : %s",emp[i].nm);
 }
}
```

*Output:*

```

Enter no of employee infor to store: 3
Enter Employee ID: 100
Enter Employee Name: Raman
Enter Employee ID: 101
Enter Employee Name: Swagat
Enter Employee ID: 102
Enter Employee Name: Aakash
Detail of employee

Employee ID: 100
Employee Name: Raman
Employee ID: 101
Employee Name: Swagat
Employee ID: 102
Employee Name: Aakash

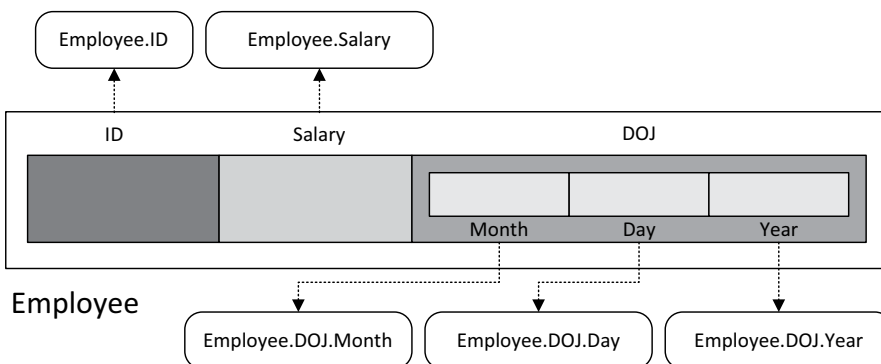
```

## 12.7 Structures within Structures (Nested Structures)

When one structure is a member of another structure, we say it is a *nesting of structures* or a *structure within a structure*. Let us take an example. Suppose we have a structure named `Employee` with the following data members: `ID`, `Salary`, and `Date of Join (DOJ)`. Here `DOJ` is itself a structure with data members: `Day`, `Month`, and `Year`. This structure design is shown in Figure 12.11

### 12.7.1 Declaration of Nested Structures

Declaration of nested structures can take different forms. Either we can declare the nested structures with all their data members inside the main structure, or we can declare each structure separately and then group them inside the main structure.



**FIGURE 12.11**  
Nested structures.

**NOTE**

When we declare the structures separately, the innermost structure must be declared first, then the next level, and finally the main structure.

Let us take the Employee example and declare the structure in all possible ways.

**12.7.1.1 Declare the Structure with One Declaration**

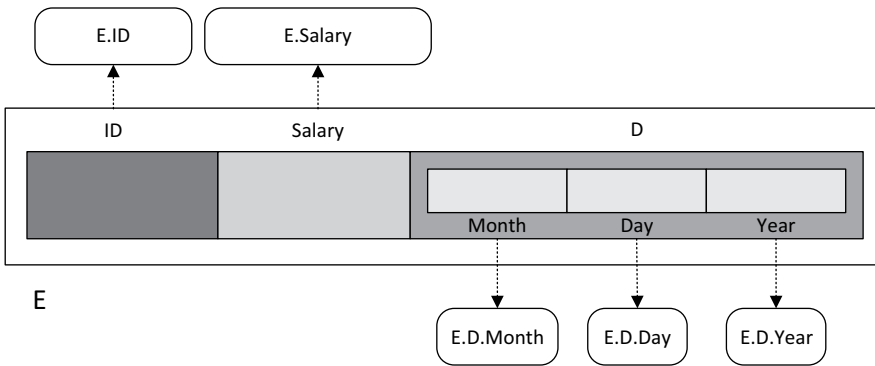
```
struct Employee
{
 int ID;
 float Salary;
 struct DOJ
 {
 int Day;
 int Month;
 int Year;
 }D;
}E;
```

**12.7.1.2 Declare the Structure Separately**

```
struct DOJ
{
 int Day;
 int Month;
 int Year;
};
struct Employee
{
 int ID;
 float Salary;
 struct DOJ D;
}E;
```

**12.7.2 Accessing the Members of a Nested Structure**

A member of a nested structure can be accessed through a structure variable and a dot (.) operator. To obtain a particular member, we can use the highest-level structure variable followed by a dot operator, then the next level structure variable and a dot operator, and finally the member's variable name. The complete set of references for the structure Employee is shown in Figure 12.12.




**FIGURE 12.12**  
Referring to individual structure members.

### 12.7.3 Nested Structure Initialization

The initialization of a nested structure is the same as the initialization of a simple structure. Each structure must be initialized entirely before proceeding to the next member. Each structure is enclosed in a set of braces.

The following example shows the initialization of the Employee structure:

|                                                                                                  |
|--------------------------------------------------------------------------------------------------|
|  Code Statement |
| <pre>struct Employee E = {101, 25356, {6, 10, 2012}};</pre>                                      |

Program 12.9 shows a complete program for the Employee structure.

#### PROGRAM 12.9

```
#include<stdio.h>
struct DOJ
{
 int Day;
 int Month;
 int Year;
};
struct Employee
{
 int ID;
 float Salary;
 struct DOJ D;
};
void main()
```

```

{
 struct Employee E;
 printf("Enter the ID, Salary and date of join: ");
 scanf("%d%f%d%d%d", &E.ID, &E.Salary, &E.D.Day, &E.D.Month,
 &E.D.Year);
 printf("\nDetail of Employee");
 printf("\n-----");
 printf("\n Employee ID: %d", E.ID);
 printf("\n Salary: %d", E.Salary);
 printf("\n Date of Join: %d-%d-%d", E.D.Day, E.D.Month,
 E.D.Year);
}

```

*Output:*

```

Enter the ID, Salary and Date of join: 101 35565 10 6 2012
Detail of Employee

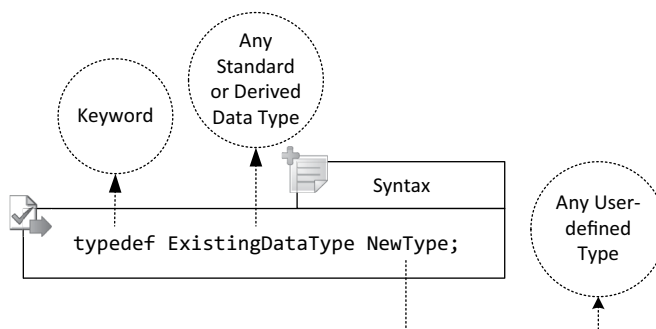
Employee ID: 101
Salary: 35565.000000
Date of Join: 10-6-2012

```

## 12.8 User-defined Data Type: *typedef*

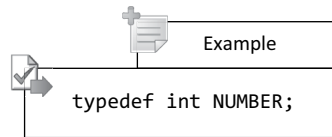
A type definition (*typedef*) is used to give a new name to an existing data type, which can again be used as a new type. We have already used this keyword to declare structures. In this section, we discuss some of the additional facilities provided by *typedef*.

We can use the type definition with any type. For example, we can redefine `int` to `NUMBER`. The general syntax is shown in Figure 12.13.



**FIGURE 12.13**  
General syntax for using `typedef`.

Example:



As shown in the example, NUMBER is a new data type (a synonym for int), and the user can use this data type to declare any number of integer variables. Program 12.10 illustrates this concept.

### PROGRAM 12.10

```
#include<stdio.h>
void main()
{
 typedef int NUMBER; /*Declaring new data type NUMBER*/
 NUMBER a, b; /* Declaring new variables using NUMBER data
 type*/
 printf("Enter two numbers: ");
 scanf ("%d%d", &a,&b);
 if(a>b)
 printf("A is greater than %d", a);
 else
 printf("B is greater than %d", b);
}
```

*Output:*

```
Enter two numbers: 20 45
B is greater than 45
```

### NOTE

The typedef identifier is traditionally coded in uppercase. This alerts the reader that there is something unusual about the type.

#### 12.8.1 Uses of typedef

- It simplifies the declaration and thus is more readable for the programmer.
- It is used for defining new data types like structure. This has already been discussed in Section 12.2.

---

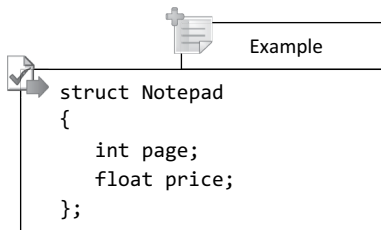
## 12.9 Pointers and Structures

A pointer is associated with a structure in many ways. A pointer can be a member of a structure; a pointer may point to a structure; or we can declare a structure pointer to access the members of a structure. In this section, we discuss some of these associations.

### 12.9.1 Accessing Structure Members Using a Pointer

We know that a pointer only points to those variables whose data type matches the data type of the pointer. So, to access a member of a structure, we need to declare a pointer variable of the same structure type. Again, we have to use the arrow operator (->) along with this pointer to access the member of a structure.

The whole scenario will look as follows. Suppose we have a structure `notepad` with the members `page` and `price`.



As shown in the example, to access each member of **Notepad**, we need to declare a pointer variable of this **Notepad** type, and we must use the arrow operator (->) to access each of the members (Program 12.11).

#### PROGRAM 12.11

```
#include<stdio.h>
struct notepad
{
 int page;
 float price;
};
void main()
{
 struct notepad S; /* Declaration of Structure Variable*/
 struct notepad *p; /*Declaration of Structure Pointer*/
 p=&S; /*Assigning the address of S to p*/
 printf("Enter the number of pages: ");
 scanf("%d", &p->page);
 printf("Enter the price: ");
```



```

scanf("%f", &p->price);
printf("Number of pages: %d", p->page);
printf("\nPrice: %f", p->price);
}

```

*Output:*

```

Enter the number of pages: 75
Enter the price: 25.56
Number of pages: 75
Price: 25.56

```

Graphically the above concept can be represented as shown in Figure 12.14.

### 12.9.2 A Pointer as a Member of a Structure

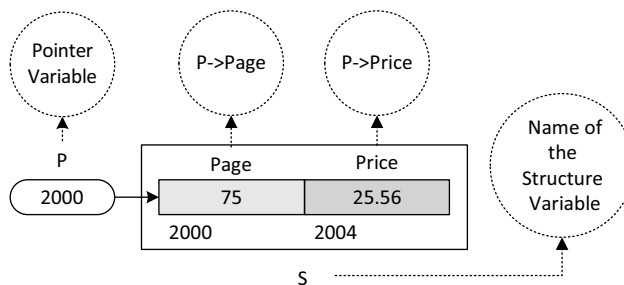
A pointer can also be a member of a structure. These members can also be accessed like a general member. These pointer members can point to any other variable depending upon its data type. Let us consider the Program 12.12.

Conceptually the memory representation of the above structure is shown in Figure 12.16.

There may be situations where an external pointer points to a specific member of a structure and a pointer member (a pointer which is a member of the structure) of a structure can point to another member of that structure. The scenario can be visualized as shown in Figures 12.17 and 12.18. The corresponding programs are shown in Programs 12.13 and 12.14.

### 12.9.3 Self-referential Structures

In computer science, one of the concepts that is used throughout is known as a *list*. A list is a collection of related data. Sometimes this list is called a *linked list*. We can define a linked list as a collection of *nodes* linked together, and each node has two parts: the first part stores the data, and the second part stores an address to the next node. The relationship is shown in Figure 12.19.



**FIGURE 12.14**

Accessing structure members using a structure pointer.

**PROGRAM 12.12**

```
1. #include<stdio.h>
2. struct notepad
3. {
4. int page;
5. float MRP;
6. float *p;
7. };
8. void main()
9. {
10. struct notepad S;
11. float Selling_Price;
12. printf("Enter the number of pages: ");
13. scanf("%d", &S.page);
14. printf("Enter the MRP: ");
15. scanf("%f", &S.MRP);
16. printf("Enter the Selling Price: ");
17. scanf("%f", &Selling_Price);
18. S.p=&Selling_Price;
19. printf("Number of pages: %d", S.page);
20. printf("\nMaximum Retail Price: %f", S.MRP);
21. printf("\nSelling Price: %f", *S.p);
22. }
```

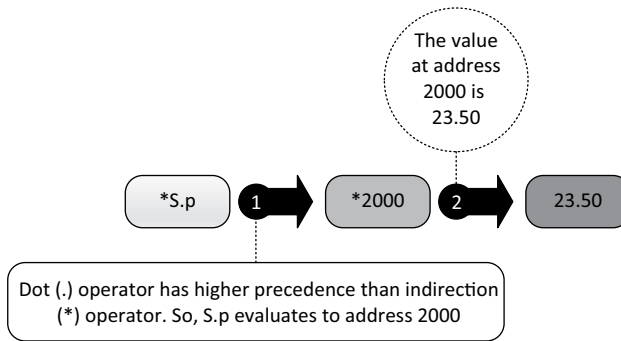
*Output:*

```
Enter the number of pages: 75
Enter the MRP: 25.56
Enter the Selling Price: 23.50
Number of pages: 75
Maximum Retail Price: 25.560000
Selling Price: 23.500000
```

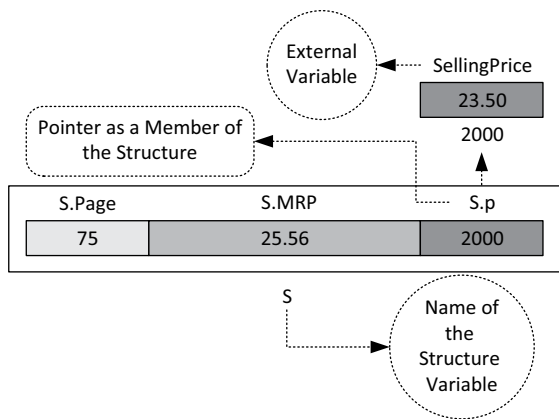
*Explanation:*

Here, *p* is a pointer which can be accessed by the structure *S* and a dot (.) operator. As *p* is a float pointer, it can point to a variable whose data type is float. To show how this works, we have taken a variable (*Selling\_Price* at line number 11) which is not a member of this structure *S*. In line 18, we have assigned the address of this external variable to *p*. At line 21 we have printed this association and the evaluation is given in Figure 12.15.

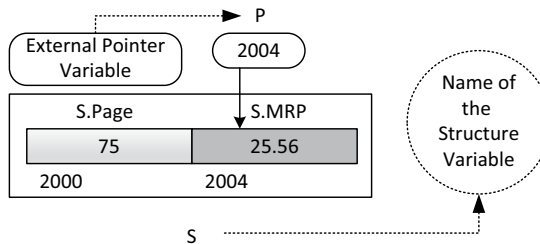
As we can see, a node is a collection of two different parts: the first part is data that is of any data type (int, float, etc.), and the second part is an address pointing to another node. So here, we can say the data type of the second part (link) must be the same as the data type of the node. Now we have to consider: what will be the data type of the node?



**FIGURE 12.15**  
Execution of line 21 (Program 12.12).



**FIGURE 12.16**  
Pointers as members of a structure.



**FIGURE 12.17**  
An external pointer pointing to a structure member.

A node can be implemented with the help of a structure because a node consists of two different types of data. According to the concept under discussion, the structure declaration of a node can be as follows:

## PROGRAM 12.13

```

#include<stdio.h>
struct notepad
{
 int page;
 float MRP;
};
void main()
{
 struct notepad S;
 float *p;
 printf("Enter the number of pages: ");
 scanf("%d", &S.page);
 printf("Enter the MRP: ");
 scanf("%f", &S.MRP);
 p=&S.MRP;
 printf("Number of pages: %d", S.page);
 printf("\nMaximum Retail Price: %f", *p);
}

```

*Output:*

```

Enter the number of pages: 75
Enter the MRP: 25.50
Number of pages: 75
Maximum Retail Price: 25.500000

```

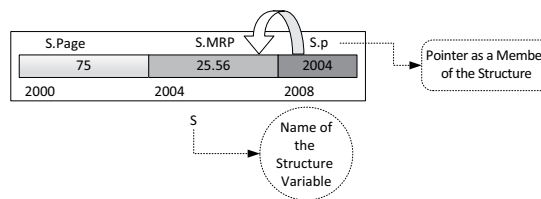


FIGURE 12.18

Pointer member pointing to a structure member.

```

struct NODE
{
 int Data;
 struct NODE *Link;
};

```

This type of node in a linked list is called a *self-referential structure*. In a self-referential structure, each variable contains at least one pointer which can point to another variable of the same structural type. Let us take an example as shown in Program 12.15.

**PROGRAM 12.14**

```
#include<stdio.h>
struct notepad
{
 int page;
 float MRP;
 float *p;
};
void main()
{
 struct notepad S;
 printf("Enter the number of pages: ");
 scanf("%d", &S.page);
 printf("Enter the MRP: ");
 scanf("%f", &S.MRP);
 S.p=&S.MRP;
 printf("Number of pages: %d", S.page);
 printf("\nMaximum Retail Price: %f", *S.p);
}
```

*Output:*

```
Enter the number of pages: 75
Enter the MRP: 25.50
Number of pages: 75
Maximum Retail Price: 25.500000
```

---

**12.10 Structures and Functions**

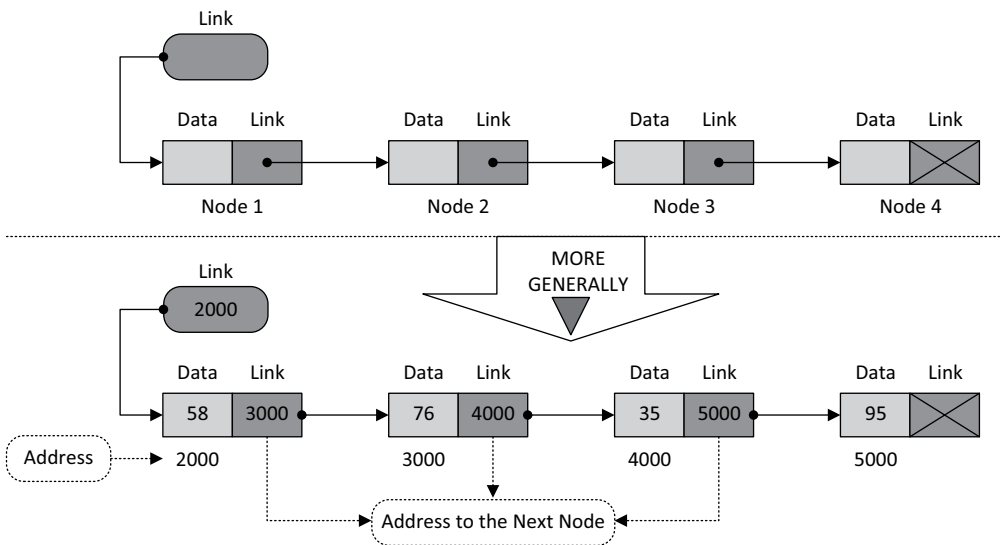
Like a general variable, a structure can also be passed to a function. The objective of doing this is to give all the functionality to a user-defined data type. We can say a structure can be fully useful if we are able to pass the structure as an argument to a function. There are different ways to pass a structure to a function:

- Passing individual members of a structure;
- Passing the whole structure using the pass by value concept;
- Passing the whole structure using the pass by address concept.

The following section explains how a structure is actually passed to a function based on the above three categories.

**12.10.1 Passing Individual Members of a Structure**

Passing individual members of a structure is equivalent to passing an individual variable to a function. To do this, we have to take parameters equal to the number of members



**FIGURE 12.19**  
Representation of a list.

### PROGRAM 12.15

```

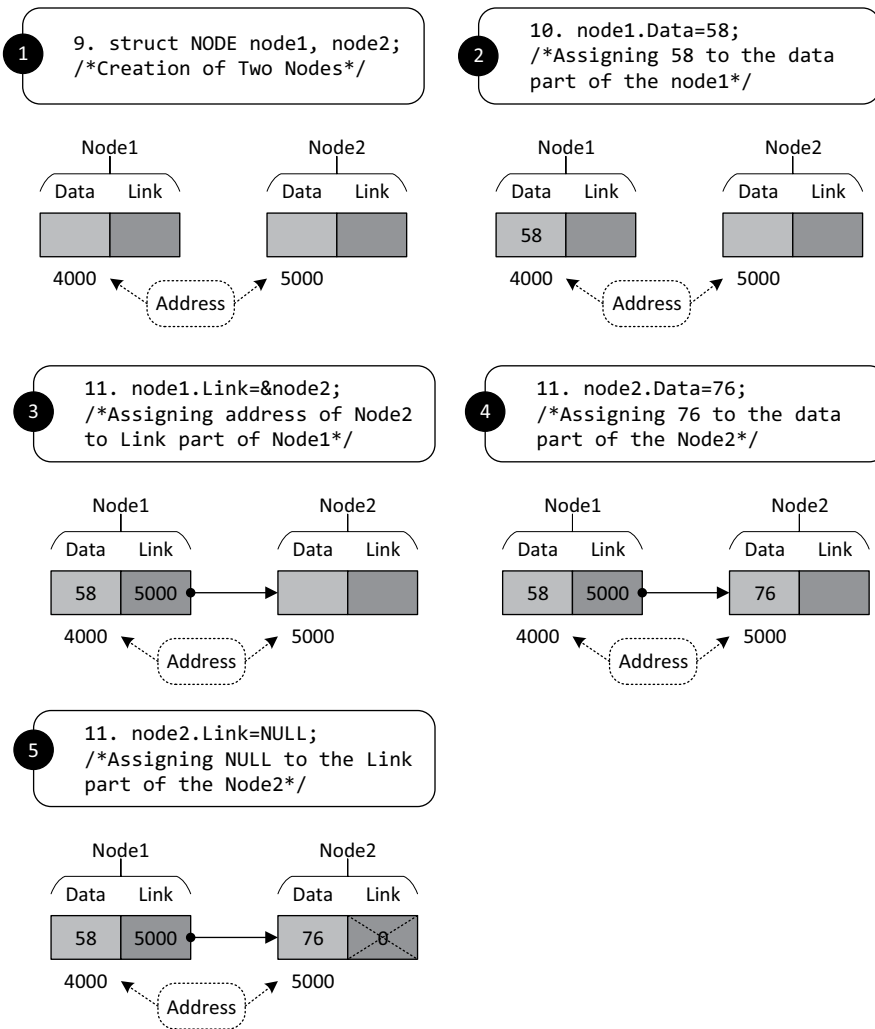
1. #include<stdio.h>
2. struct NODE
3. {
4. int data;
5. struct NODE *Link;
6. };
7. void main()
8. {
9. struct NODE node1, node2;
10. node1.data=58;
11. node1.Link=&node2; /*Address of node2 is assigned to link part
of node1*/
12. node2.data=76;
13. node2.Link=NULL; /* Link part of node2 assigned to NULL */
14. printf("\n|%d|%u|", node1.data, node1.Link);
15. printf(" -> |%d|%u|", node2.data, node2.Link);
16. }

```

*Output:*

```
|58|8680| -> |76|0|
```

The step-by-step execution of lines 9 to 13 is shown in Figure 12.20.



**FIGURE 12.20**  
Execution of Program 12.15 (lines 9 to 13).

present in the structure. The value of each individual member is assigned to a specific parameter. For example, if our structure contains three members, then our function should also include three parameters to store the value of individual members. But remember, the data type of the parameter should match with the data type of the structure members.

Let us take an example to show how to pass individual members of a structure to a function (Program 12.16).

### 12.10.2 Passing the Whole Structure Using the Pass by Value Concept

In this case, the whole structure can be passed to a function. To understand how this concept works, let us first discuss some of the related issues associated with structure assignment.

**PROGRAM 12.16**

```

#include<stdio.h>
struct notepad
{
 int page;
 float price;
};
void display(int, float); /* Function Prototype*/
void main()
{
 struct notepad N;
 printf("Enter page and price of notepad: ");
 scanf("%d %f", &N.page, &N.price);
 display(N.page, N.price); /* Function Calling by passing
 individual members*/
}
void display(int x, float y)
{
 printf("\nNumber of pages: %d", x);
 printf("\nPrice: %f", y);
}

```

*Output:*

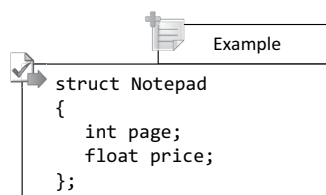
```

Enter page and price of notepad: 75 35.50
Number of pages: 75
Price: 35.500000

```

A structure variable can be assigned to another structure variable if and only if both are the variable of the same structure.

Consider the following structure:

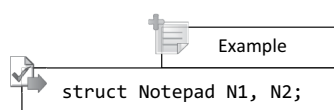


```

struct Notepad
{
 int page;
 float price;
};

```

Suppose we declare two variables of the above structure type:



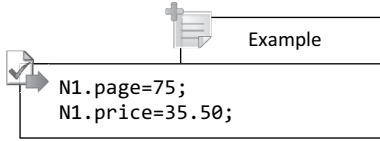
```

struct Notepad N1, N2;

```

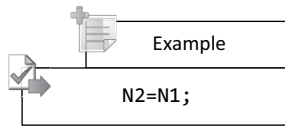


Now, let us assign some value to structure N1 using the following lines:



After the above assignment, the basic structure is shown in Figure 12.21.

According to the above description, N1 and N2 are variables of notepad type. So, we can assign the value of N1 to N2 directly using the following line and automatically 75 will be assigned to page of N2 and 35.50 is assigned to the price of N2:

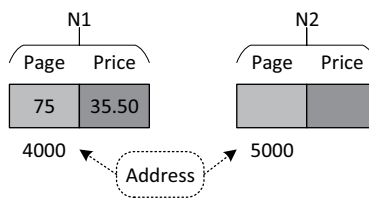


The representation after the assignment is shown in Figure 12.22.

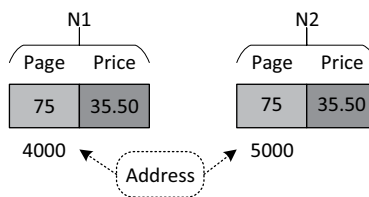
By using the above concept, we can pass a whole structure to a function by using the pass by value concept. Program 12.17 shows a complete program for this purpose.

### 12.10.3 Passing the Whole Structure Using the Pass by Address Concept

This is similar to the pass by value method, but the only difference is here we are passing the address of the structure variable. However, the function parameter should be a pointer of the same structure type. Let us rewrite Program 12.18 to show how this concept works:



**FIGURE 12.21**  
Representation of N1 and N2.



**FIGURE 12.22**  
Representation of N1 and N2 after assignment.

**PROGRAM 12.17**

```
#include<stdio.h>
struct notepad
{
 int page;
 float price;
};
void display(struct notepad);
void main()
{
 struct notepad N;
 printf("Enter page and price of notepad: ");
 scanf("%d %f", &N.page, &N.price);
 display(N); /* Passing the whole structure to the function*/
}
void display(struct notepad x) /*x is notepad type, so N can be
assigned to x*/
{
 printf("\nNumber of pages: %d", x.page);
 printf("\nPrice: %f", x.price);
}
```

**Output:**

```
Enter page and price of notepad: 75 35.50
Number of pages: 75
Price: 35.500000
```

Recall Problem 1 discussed in Section 12.1. Let us solve the problem using the concept of passing a structure to a function. Declare a structure TIME with data members: hour, minute, and second. Write a function that adds two different times using the pass by value method.

---

**12.11 Unions**

A union can be defined as a collection of one or more variables of the same or different data types grouped together under a single name. As per the definition, a union is the same as a structure with slight differences. The difference between structures and unions will be discussed in Section 12.12. The major difference is: a union provides a way by which the memory space can be shared by its members.

**12.11.1 Declaration of a Union**

A union follows the same syntax as the declaration of a structure. In fact, a union uses a keyword *union* for declaration rather than the *struct* keyword. The syntax and an example are shown in Figure 12.23.

**PROGRAM 12.18**

```
#include<stdio.h>
struct Time
{
 int hour;
 int min;
 int sec;
};
void addTime(struct Time T3, struct Time T4)
{
 struct Time T5;
 T5.hour=T3.hour+T4.hour;
 T5.min=T3.min+T4.min;
 T5.sec=T3.sec+T4.sec;
 if(T5.sec>60)
 {
 T5.min=T5.min+1;
 T5.sec=T5.sec-60;
 }
 if(T5.min>60)
 {
 T5.hour=T5.hour+1;
 T5.min=T5.min-60;
 }
 printf("Addition of the two times=%d:%d:%d",T5.hour,T5.min,T5.
sec);
}
void main()
{
 struct Time T1, T2;
 printf("Enter the time1(hour min sec): ");
 scanf("%d %d %d", &T1.hour, &T1.min, &T1.sec);
 printf("Enter the time2(hour min sec): ");
 scanf("%d %d %d", &T2.hour, &T2.min, &T2.sec);
 addTime(T1, T2);
}
```

**Output:**

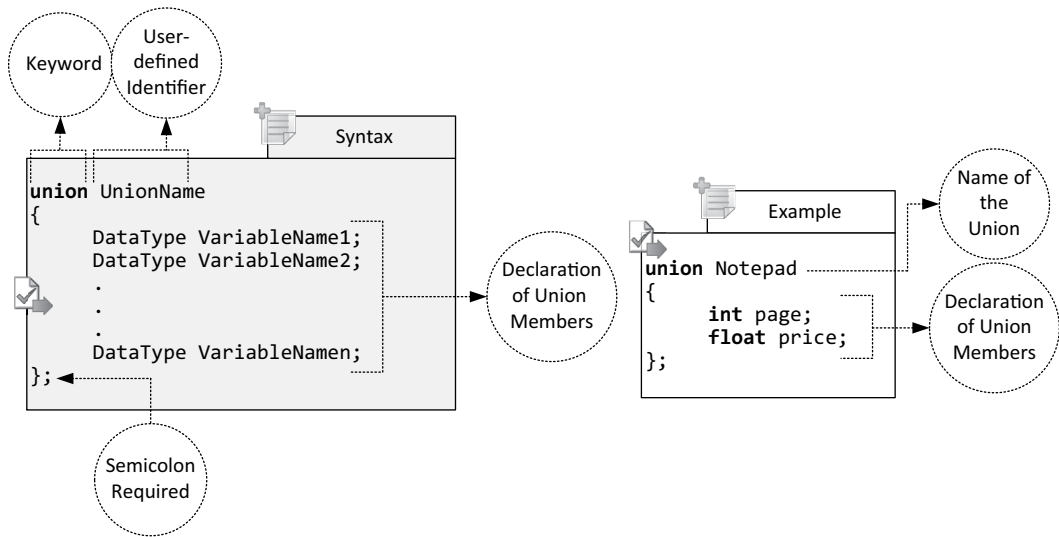
```
Enter the time1 (hour min sec): 4 30 40
Enter the time2 (hour min sec): 5 40 35
Addition of the two times = 10 : 11 : 15
```

**PROGRAM 12.19**

```
#include<stdio.h>
struct Time
{
 int hour;
 int min;
 int sec;
};
void addTime(struct Time *T3, struct Time *T4)
{
 struct Time T5;
 T5.hour=T3->hour+T4->hour;
 T5.min=T3->min+T4->min;
 T5.sec=T3->sec+T4->sec;
 if(T5.sec>60)
 {
 T5.min=T5.min+1;
 T5.sec=T5.sec-60;
 }
 if(T5.min>60)
 {
 T5.hour=T5.hour+1;
 T5.min=T5.min-60;
 }
 printf("Addition of the two times = %d:%d:%d",T5.hour,T5.
min,T5.sec);
}
void main()
{
 struct Time T1, T2;
 printf("Enter the time1(hour min sec): ");
 scanf("%d %d %d", &T1.hour, &T1.min, &T1.sec);
 printf("Enter the time2(hour min sec): ");
 scanf("%d %d %d", &T2.hour, &T2.min, &T2.sec);
 addTime(&T1, &T2);
}
```

**Output:**

```
Enter the time1 (hour min sec): 4 30 40
Enter the time2 (hour min sec): 5 40 35
Addition of the two times = 10 : 11 : 15
```



**FIGURE 12.23**  
Declaration and example of a union.

### 12.11.2 Member Accessing

Union members can also be accessed similarly to a structure. To access each member, we can use the dot (.) operator or the arrow (->) operator. For the example shown in Figure 12.23, if we declare a union variable as follows:

```

Example
union Notepad N;

```

then, each member of this union can be accessed by the following valid statement:

```

Example
N.page
N.price

```

#### NOTE

Most of the concepts that we have discussed previously for structures are also applicable to unions.

## 12.12 Structures vs. Unions

There are many aspects where a union is different from a structure. In this section, we concentrate on some similarities and differences between structures and unions.

### 12.12.1 Size of Unions and Structures

As we know, the size of a structure is the sum of the size of all the members present in the structure. But the size of a union is equivalent to the largest member of the union. Figure 12.24 shows the size difference between a union and a structure.

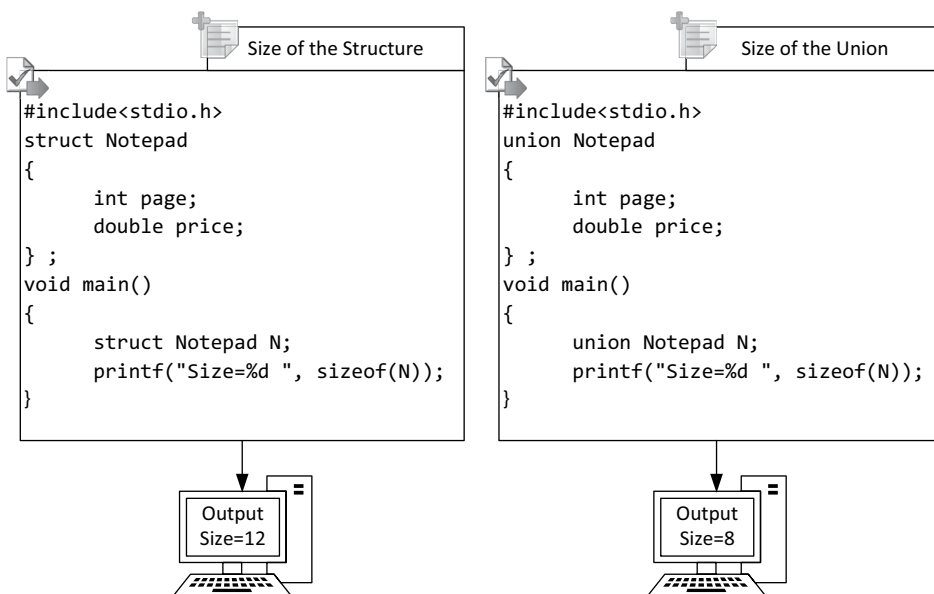
### 12.12.2 Sharing Memory and Member Accessing

With a union, at one instance of time, we can access only one member because the union allocates memory space for the largest member only. Other members can only share it from time to time. That means at one instance of time, only one member is available in the memory. For example, let's say there are three members in a family, but they have only one chair. So at any instance, only one member can sit on it. The analogy is shown in Figure 12.25.

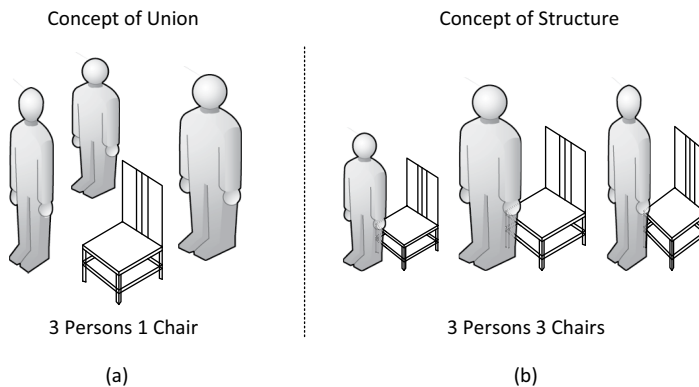
With a structure, memory is allocated for all the members. Hence, at any instance of time, any member can be accessed. For example, let's say there are three members in a family, and they all have their chairs. See Figure 12.25.

Let us take a programming example to show you the effect of accessing members at the same time for both unions and structures. Figure 12.26 shows these programs.

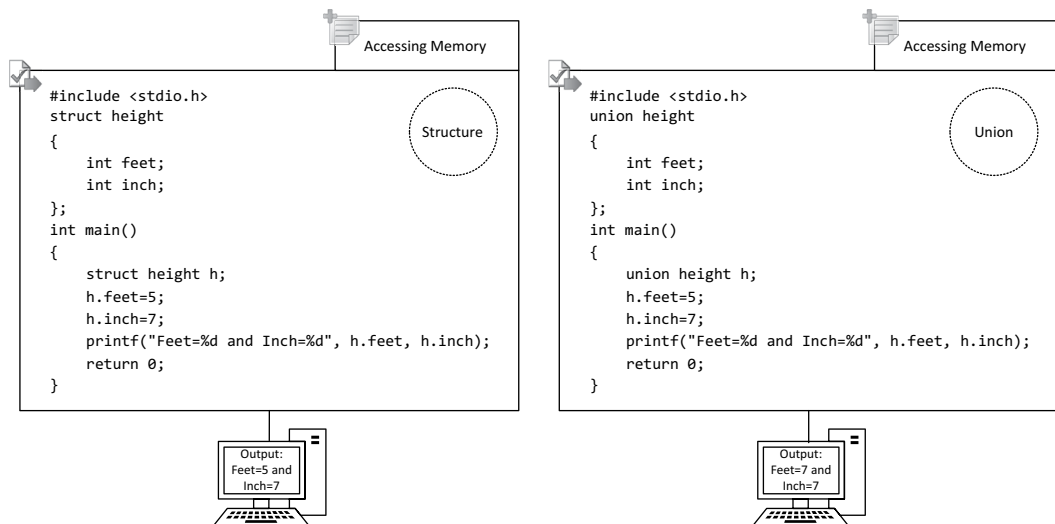
In the case of structures, memory is allocated for both feet and inches (Figure 12.27). So when we access its content through `printf()`, it gets displayed. But in the case of unions, one space is allocated for both members: feet and inches. When we write `h.feet=5`, 5 is assigned to that location and, later on, when `h.inch=7` gets executed, the previous value (5) is replaced with 7. Finally, when we print the content of that location, 7 is printed for both cases: `h.feet`, `h.inch`. For easy understanding see Figure 12.27 that shows the execution steps for union memory allocation.



**FIGURE 12.24**  
Size of a structure and a union.



**FIGURE 12.25**  
Analogy of memory access in a union and a structure.

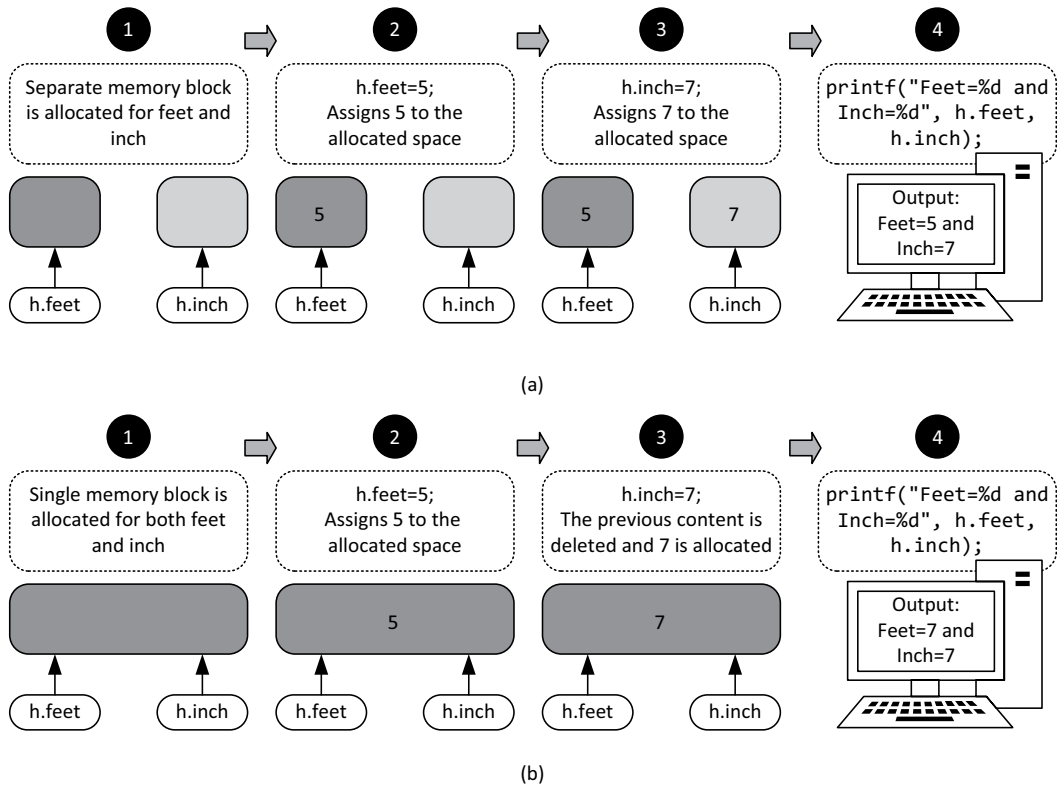


**FIGURE 12.26**  
Difference between structures and unions with respect to memory access.

## 12.13 Bitfields

Before we begin the discussion on bitfields, we must know the compiler and the system configuration on which we are running our programs because the size of the data types is different from system to system. Until now, we have taken integer to be four bytes, but this is not always true for every system. When we run our programs using a 32-bit computer, then integer takes two bytes but, when you run the same program on a 64-bit computer, it takes four bytes. It is important to know the system you are using. For our discussion on bitfields, we use a 64-bit computer that takes four bytes for integer allocation.

Consider Program 12.20 to clarify your doubts regarding the size of different data types. We will run this program on a 64-bit computer with a c99 compiler. We recommend that



**FIGURE 12.27** Execution steps of the memory access program.

before you proceed, you run this program on your computer and see what your system shows you.

**NOTE**

For our discussion on bitfields, we use a 64-bit computer that takes four bytes for integer allocation.

The compiler allocates memory for all the members present inside a structure. For example, let a structure have two members: an integer and a float member, then a total of eight bytes is allocated: four bytes for integers, and four bytes for floats. Assume that we want to allocate as many bits as we want. For instance, we want to allocate three bits for some members, five bits for some other members, and so on. Can we do this? Is it possible? Yes, it is possible. Mark the bits here, we are not talking about bytes. It is possible to allocate memory at the bit level for the members of a structure. The concept of bitfields provides this facility. We use a bitfield to allocate a distinct number of bits for the members of a structure irrespective of their data types.



**PROGRAM 12.20**

```

#include <stdio.h>
int main()
{
 printf("\nSize of int= %d", sizeof(int));
 printf("\nSize of short int= %d", sizeof(short int));
 printf("\nSize of long int= %d", sizeof(long int));
 printf("\nSize of char= %d", sizeof(char));
 printf("\nSize of float= %d", sizeof(float));
 printf("\nSize of double= %d", sizeof(double));
 printf("\nSize of long double= %d", sizeof(long double));
 return 0;
}

```

*Output:*

```

Size of int= 4
Size of short int= 2
Size of long int= 8
Size of char= 1
Size of float= 4
Size of double= 8
Size of long double= 16

```

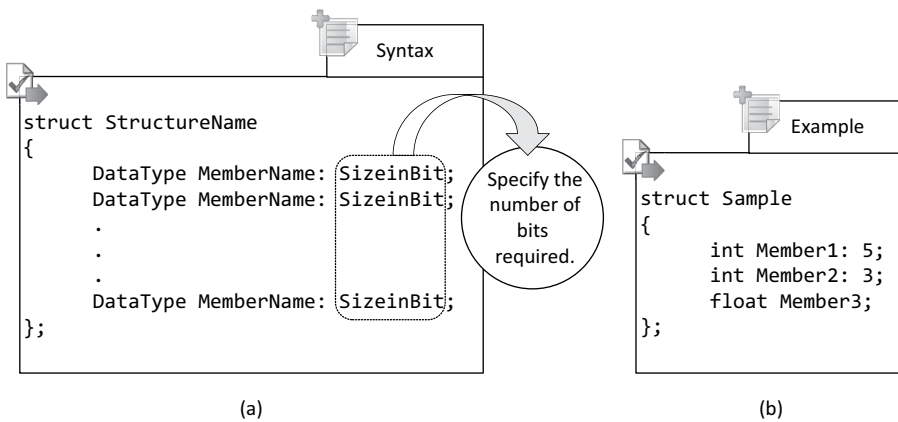
We use bitfield to allocate a distinct number of bits for the members of a structure irrespective of their data types.

**12.13.1 Declaration of a Bitfield**

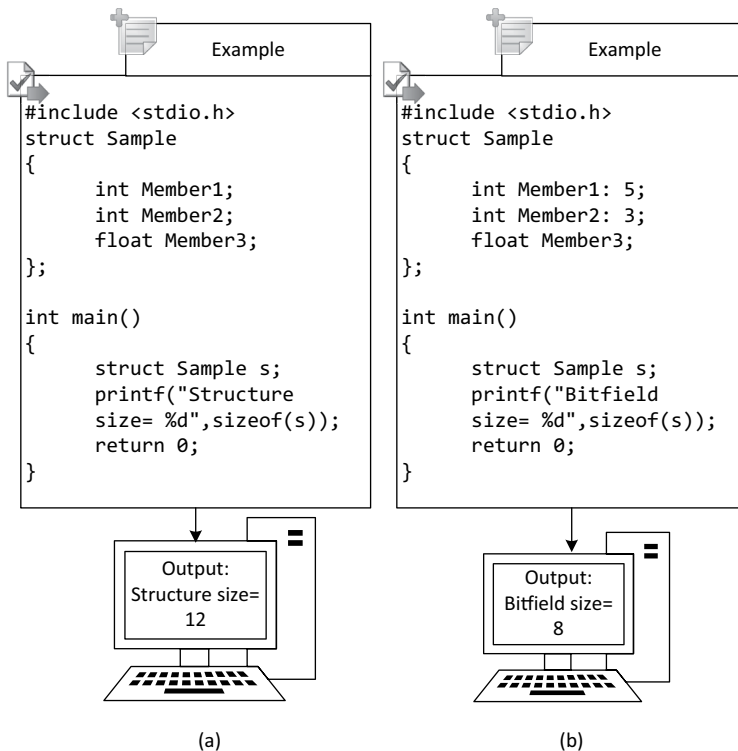
Bitfield declaration takes the form, shown in Figure 12.28 with an example.

Consider the programs shown in Figure 12.29a. Here, the compiler adds the size of each member of the structure, and finally prints the size as 12; the size of each member is four bytes. Refer to Program 12.20 to see the size of the different data types. In Figure 12.29b, we use the concept of the bitfield and allocate five bits for member 1 and three bits for member 2. The float member takes four bytes. You may assume that the compiler will allocate four bytes and eight bits (five bits for member 1 + three bits for member 2) for the entire structure. But it is not true, the compiler allocates four bytes for floats (as per the rule), and another four bytes for the remaining members. Out of these 4 bytes (32 bits), only 8 bits are used by members 1 and 2. The remaining 24 bits will be padded.

To provide more clarity, let us consider the programs shown in Figure 12.30. For the first case (Program 12.30a), we allocate a 16-bit memory for member1 and a 16-bit memory for member2. In total, 32 bits (4 bytes) for both members. Member3 takes 4 bytes. Hence, the structure size is 8 bytes, as shown in the output. In Figure 12.30b, we allocate 16 bits for member1, 17 bits for member2, and member3 takes 4 bytes, which is obvious. The size of



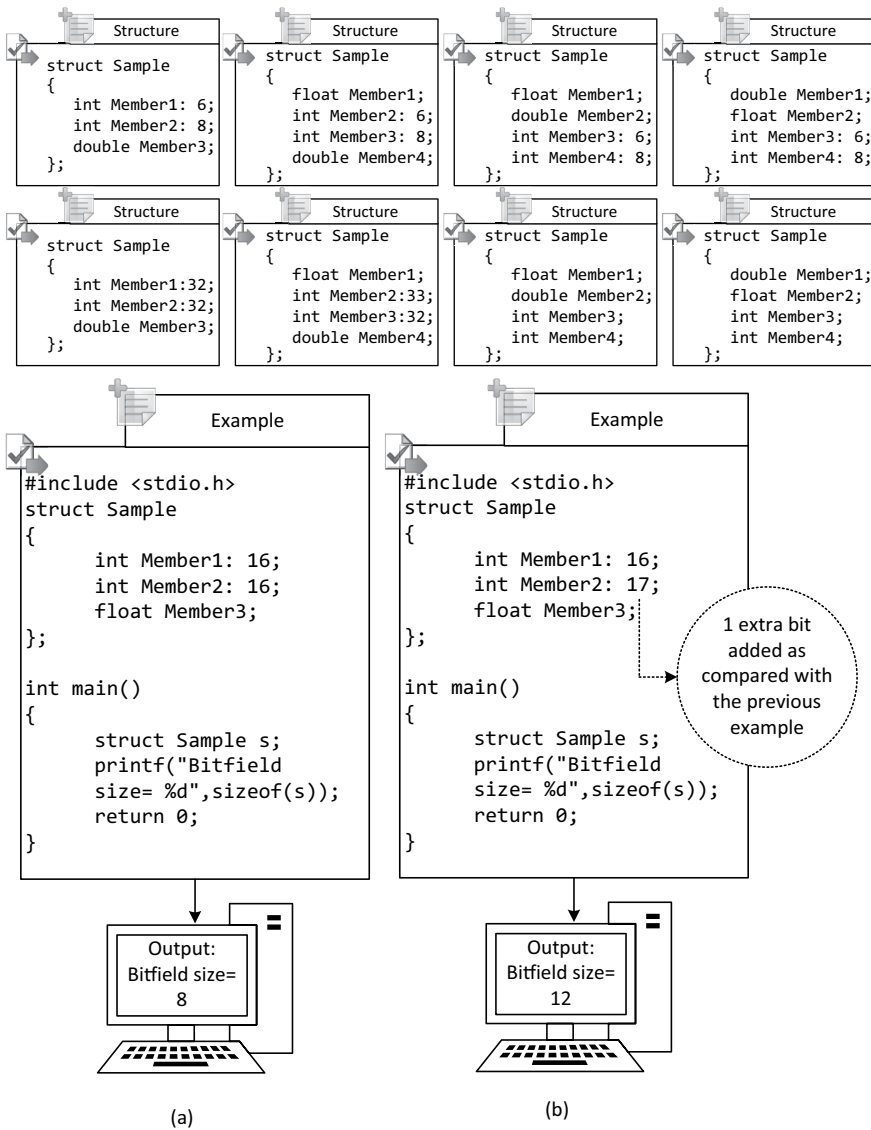
**FIGURE 12.28**  
Syntax of a bitfield declaration and an example.



**FIGURE 12.29**  
Comparing bitfield and structure members.

the structure has now become 12 bytes, as shown in the output. From the output, we can conclude that the compiler allocates another 4 bytes to accommodate 1 bit, which is an extra bit associated with member2.

**Quiz:** The reader should use the following structure declaration and find its size to know more about how the compiler allocates memory for structure members.

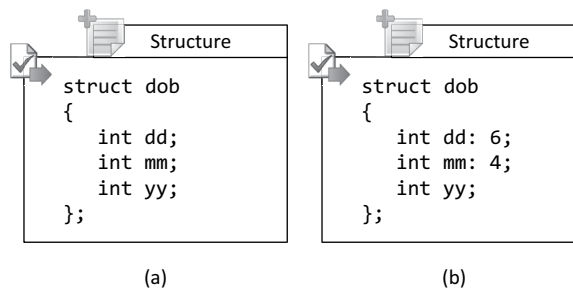


**FIGURE 12.30**  
Bitfield memory allocation.

### 12.13.2 Uses of Bitfields

We can conclude that we use bitfields to allocate memory space as much as we want, irrespective of the data type. Suppose you want to store your birth date in date, month, and year format. You may declare three integer variables for this inside a structure, as shown in Figure 12.31a.

The size of this structure is 12 bytes. We know that there is a maximum of 31 days in a month and 12 months in a year. To represent 31, we need 6 bits, and 4 bits to represent 12. That's why we can modify the above structure declaration using the concept of a bitfield, as shown in Figure 12.31b. The size of this structure will be eight bytes.



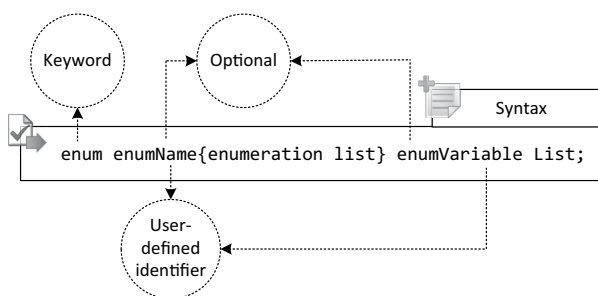
**FIGURE 12.31**  
Uses of bitfields.

## 12.14 Enumeration

In this section, we will discuss another user-defined data type known as enumeration. We use this to assign numbers to a string literal. Numbers can be assigned to departments in your college, days of a week, months of a year, or the currency of different countries. For instance, we can assign 0 to the yen, 1 to the dollar, 2 to the rupee, and so on. By doing this, we can easily process information, and the programs look more readable. To declare an enumeration, we use a new keyword known as *enum*.

Enumeration is declared like a structure. The only difference is each member of a structure is declared using a data type, but the members of an enumeration do not require any data type. The members are string literals, and the compiler assigns integers (starting from 0) to each member. The declaration syntax is shown in Figure 12.32.

Here `enumVariable list` and `enumName` are optional. We can use either to declare an enum. Program 12.21 shows how enum is declared and what value it gives when printed.



**FIGURE 12.32**  
Enumeration syntax.

### PROGRAM 12.21

```

#include<stdio.h>
enum currency{yen, dollar, rupee, pound, siling, dinar};
int main()

```

```

{
 printf("\nYen= %d", yen);
 printf("\nDollar= %d", dollar);
 printf("\nRupee= %d", rupee);
 printf("\nPound= %d", pound);
 printf("\nSiling= %d", siling);
 printf("\nDinar= %d", dinar);
}

```

*Output:*

```

Yen= 0
Dollar= 1
Rupee= 2
Pound= 3
Siling= 4
Dinar= 5

```

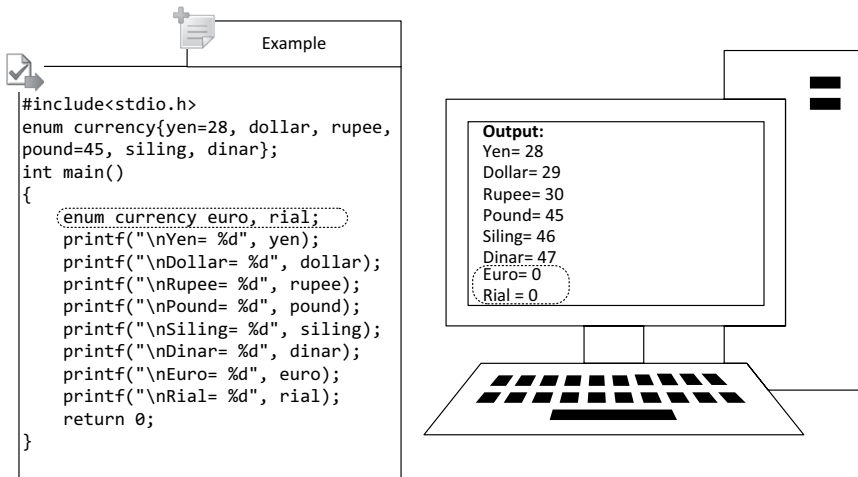
We can assign any number to the enum members. In Figure 12.33, we write two programs to show you how to assign user-defined numbers to the enum members. The output is easy and self-explanatory.



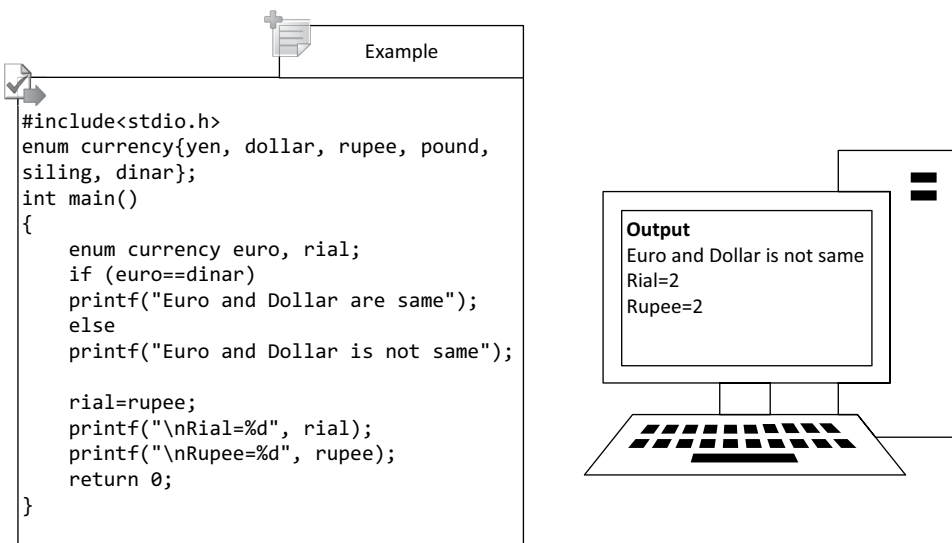
**FIGURE 12.33**  
Examples of using the enum keyword.

We can declare more members of the enum type other than the primary declaration. One such example is given in Figure 12.34. You can observe that the new member is assigned with 0 (zero).

We can also compare or assign values among members and make decisions. Observe the following program shown in Figure 12.35.



**FIGURE 12.34**  
Declaring enum members outside enum declaration.



**FIGURE 12.35**  
Compare and assign values among members of enumeration.

---

## 12.15 Review Questions

### 12.15.1 Objective Questions

1. A \_\_\_\_\_ can be defined as a collection of one or more variables of the same or different data types grouped together under a single name.
2. \_\_\_\_\_ and \_\_\_\_\_ are the two keywords used in the declaration of a structure.
3. On the declaration of a structure, no memory space is allocated for the structure members. True/false?
4. The size of a structure variable is equal to the sum of each member's length present in that structure. True/false?
5. \_\_\_\_\_ and \_\_\_\_\_ operator is used to access the members of a structure.
6. Which keyword is used to give a new name to an existing data type?
7. In a \_\_\_\_\_, each variable contains at least one pointer which can point to another variable of the same structural type.
8. We use \_\_\_\_\_ to allocate a distinct number of bits for the members of a structure irrespective of their data types.

### 12.15.2 Subjective Questions

1. Define a structure and explain how to declare one using the keywords struct and typedef.
2. How can you initialize a structure variable? Explain with an appropriate example.
3. There are two member access operators to access a structure member. Explain both with appropriate examples.
4. What do we mean by an array of structures? Explain its syntax.
5. Write short notes on enumeration and bitfields.
6. What is the difference between structures and unions. Explain with examples.
7. How can we pass a structure to a function? Explain with an appropriate example.
8. Can we pass one member of a structure variable to a function? If yes, explain how.

### 12.15.3 Programming Exercises

1. What is the output of the following program?

```
#include<stdio.h>
int main()
{
 union demo
 {
 int x;
 int y;
 };
};
```

```

 union demo a = 100;
 printf("%d %d", a.x, a.y);
}

```

2. What is the output of the following program?

```

int main()
{
 struct CBook
 {
 char *name;
 int year;
 };
 struct CBook c1 = {"Learn to Code", 2020};
 struct Cbook c2 = 11;
 printf("%s %d", c2.name, c1.year);
}

```

3. What is the output of the following program?

```

#include<stdio.h>
int main()
{
 struct employee
 {
 int empid[5];
 int salary;
 employee *s;
 }emp;
 printf("%d %d", sizeof(employee), sizeof(emp.empid));
 return 0;
}

```

4. Create a structure "Cricket" with the following fields: Player\_Name, Team\_Name, Average. Use appropriate data types. Read five players' records and display them in a formatted manner.
5. Write a program to store and print the roll no., name, age, and marks of a student using structures.
6. Write a program to add, subtract, and multiply two complex numbers using structures to a function.
7. Enter the marks of five students in Computing, Biology, and Physics (each out of 100) using a structure named Marks and having member variables roll no., name, comp\_marks, biol\_marks, and phy\_marks and then display the percentage of each student.
8. Write a program to compare two dates entered by the user. Make a structure named Date to store the day, month, and year. If the dates are equal, display "Dates are equal"; otherwise display "Dates are not equal".
9. Create a structure named Calendar having day, month, and year as its member variables. Store the current date in the structure. Now add 50 days to the current date and display the new date.



10. Create a structure containing book details like accession number, name of author, book title, and flag to know whether book is issued or not. Create a menu for a library where if we issue a book, then its number gets decreased by 1 and if we add a book, its number gets increased by 1. Write a function to implement the following operations:
- Display book information;
  - Add a new book;
  - Display all the books in the library of a particular author;
  - Display the number of books of a particular title;
  - Display the total number of books in the library;
  - Issue a book.

# 13

## *Dynamic Memory Allocation*

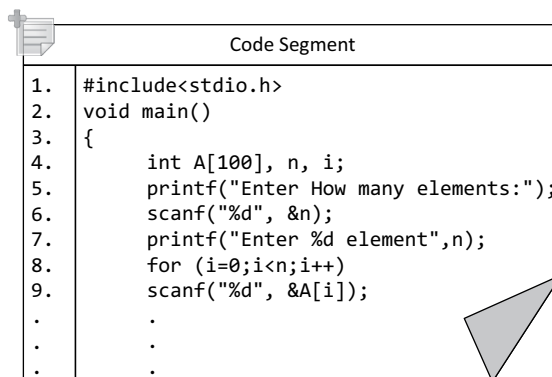
### 13.1 Introduction

We will begin this chapter by analyzing the following code segment shown in Figure 13.1. The code is written to allocate memory space for an array. Generally, we take a considerably larger value (in our code, we take 100 as the size; line number 4) for declaring the size of an array.

Here the size of the array is 100. Automatically, the memory will be allocated for 100 integers, but generally we don't use all the spaces. As you can see in line 5, we are asking about the number of elements we want to use from the allocated memory. Suppose someone enters 10; then the rest of the memory (allocated memory of 90 integers) will be of no use, and so unnecessarily we are blocking these sets of memory. Further, if we found that a 100 memory is not enough, then we need to change the code in the appropriate places. This happens because we are using the static memory allocation concept in the program code.

In the static memory allocation concept, we anticipate that  $x$  amount of memory may be required for my program, but in reality, either less or more than the  $x$  amount is used. In the former case, we are wasting the allocation by blocking some memory locations without using it, and in the latter case, we need to modify our program.

Not only in the case of an array, but in most situations, we are not able to know the amount of memory required until run time. If we can allocate memory at run time, then we can avoid the problem of memory wastage. It is possible to allocate a fixed amount of



```
Code Segment
1. #include<stdio.h>
2. void main()
3. {
4. int A[100], n, i;
5. printf("Enter How many elements:");
6. scanf("%d", &n);
7. printf("Enter %d element",n);
8. for (i=0;i<n;i++)
9. scanf("%d", &A[i]);
.
.
.
```

**FIGURE 13.1**  
A code segment.

memory as and when required. This process of allocating memory at run time is called *dynamic memory allocation*.

This chapter introduces the concept of dynamic memory allocation in C. Until now, we have written programs, all of which used static allocation. C provides different functions that help us in allocating memory dynamically. After finishing this chapter, the student will have learnt the following:

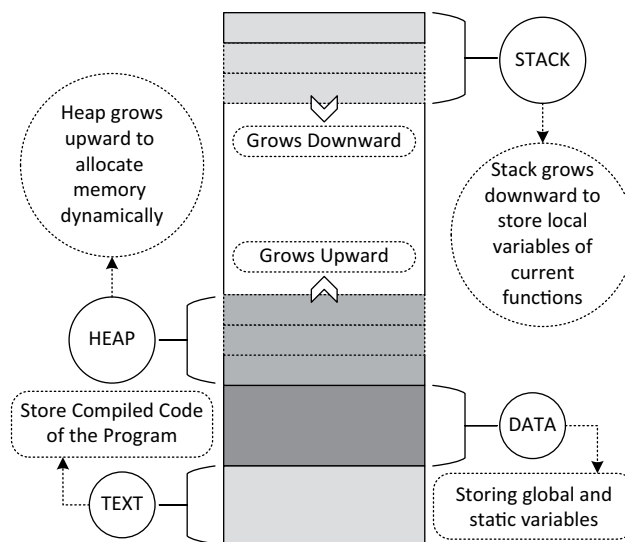
- Be able to define dynamic memory allocation and differentiate it from static allocations;
- Be able to explain different functions provided by C for dynamic memory allocation, like `malloc()`, `calloc()`, `realloc()`, and `free()`.

### 13.1.1 Process of Memory Allocation

Before we can understand dynamic memory allocation, we need to know how the memory allocation process is carried out in the C language. The memory as a whole is divided into four segments:

- Text;
- Data;
- Stack;
- Heap.

The organizations of these segments are conceptually shown in Figure 13.2.



**FIGURE 13.2**  
Organization of memory.

### 13.1.1.1 Text Segments

The compiled code of a program resides in the text segment. Text segments are also called *code segments*.

### 13.1.1.2 Data Segments

The global variables and local static variables are stored in this section. Generally, static variables and global variables can be accessed by any function.

### 13.1.1.3 Stack Segments

A stack follows the concept of LIFO (Last In First Out) to store and remove data. Inserting an element into a stack is called a PUSH operation and removing of an element is called a POP operation. When a function locally declares a new variable, it is pushed into the stack, and when the execution of the function completes, the allocated space is released.

### 13.1.1.4 Heap Segments

A heap is a free memory pool that can be used for dynamic memory allocation. The size of the heap is always larger than a stack. The C compiler provides some predefined functions which can be used to allocate memory in the heap area. Once the memory is allocated, it is the responsibility of the programmer to release that memory. The process of releasing memory is always performed after execution finishes. The C language provides several predefined functions to release the memory. If, for any reason, we are not able to release the allocated memory, then a *memory leak* occurs, which degrades the performance of the computer. Reading from a heap and writing to a heap takes slightly more time because the pointer is required to access the memory present in the heap.

---

## 13.2 Types of Memory Allocation

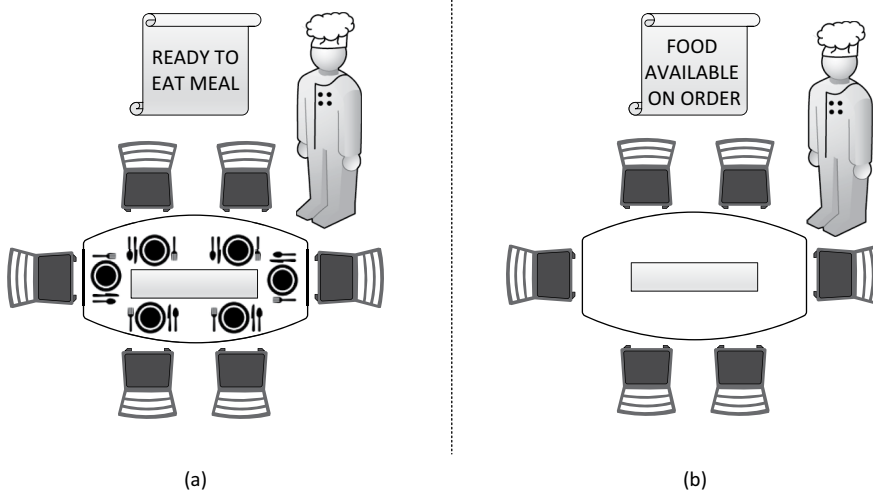
Memory allocation is categorized into two types: static memory allocation and dynamic memory allocation. Until now, in whatever programs we have written, all the allocation has been done using static allocation. Generally, static allocation is made inside a stack, and dynamic allocation is made inside a heap.

### 13.2.1 Static Memory Allocation

Static memory allocation refers to the allocation of memory at compile time. After the allocation is done, we cannot modify its size. All allocation, in this case, is made inside the stack.

### 13.2.2 Dynamic Memory Allocation

Dynamic memory allocation refers to the allocation of memory at run time. We can easily request the exact amount of memory required by the user and allocate it during execution. All allocation, in this case, is made inside a heap.



**FIGURE 13.3**  
Static and dynamic allocation analogy.

Let's use an analogy to explain the static and dynamic allocation concepts. Observe Figure 13.3, where two chefs are waiting for their customers.

- In the first case (Figure 13.3a), the chef has prepared a meal for six people and has arranged plates on the dining table with a board "READY TO EAT MEAL". This case can be considered as static, and three situations may arise. First, he gets exactly six customers, and if this happens, all the food prepared by the chef will be finished with no wastage of food. Second, he gets less than six customers, which means a wastage of food. Third, there will be more than six customers, which means he will not be able to serve them all.
- In the second case (Figure 13.3b), the chef is ready to prepare the meal when an order is placed. This case is analogous to dynamic allocation. The food will be prepared only when the chef gets an order to prepare it. Here, no food will be wasted.

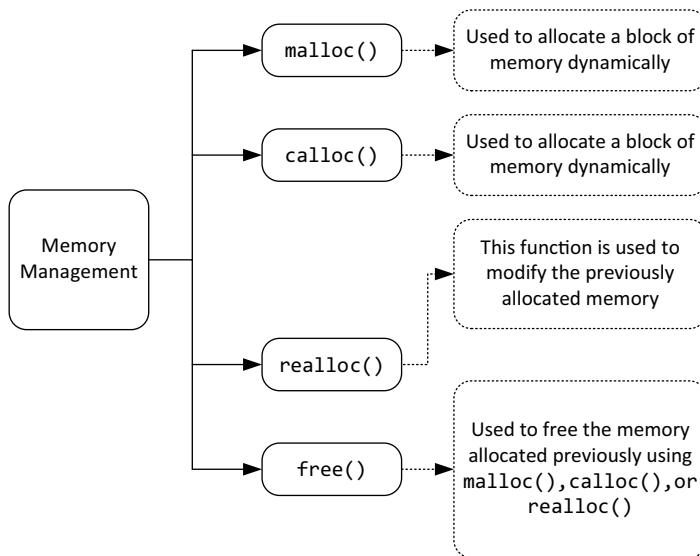
---

### 13.3 Dynamic Memory Allocation Process

There are four built-in functions available in C that help us to allocate memory dynamically:

- `malloc()`
- `calloc()`
- `realloc()`
- `free()`

These four functions are defined inside the header file `<stdlib.h>` and `<alloc.h>`. Figure 13.4 provides an overall description of these four functions, which have their own syntax for



**FIGURE 13.4**  
Memory management functions.

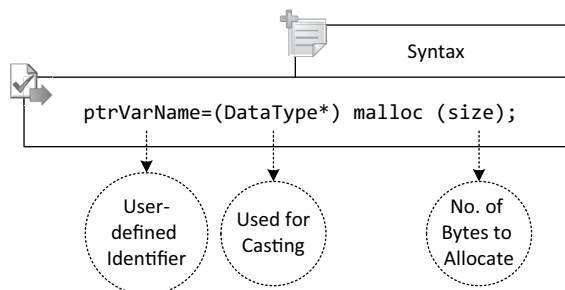
declaration. Memory allocation and accessing the data present in the heap can only be implemented by using a pointer.

### 13.3.1 The malloc () Function

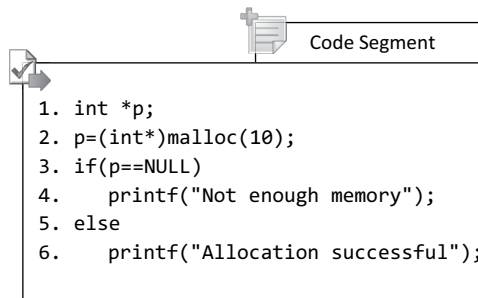
This function allocates a block of memory (specified in bytes) in the heap. This function allows a user to allocate memory as and when required. The syntax for malloc () is shown in Figure 13.5.

The syntax for malloc () allocates the required amount of memory specified by the attribute *size* and returns the first address of the allocated memory. If the allocation of memory fails due to lack of available space, then it returns a NULL pointer. The code segment for successful allocation can be written as shown in Figure 13.6.

After allocating the required amount of memory space, malloc () returns a void pointer. We know that a void pointer needs to be properly cast before it is used inside our program (refer to Section 11.5 to know more about void pointers and why casting is needed). Hence,



**FIGURE 13.5**  
Syntax for the malloc () function.



```

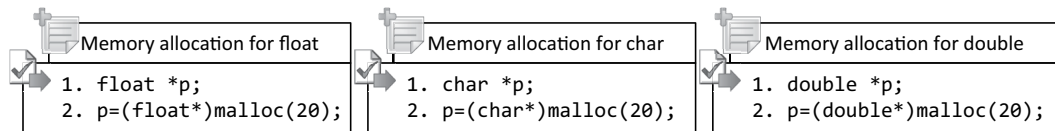
Code Segment
1. int *p;
2. p=(int*)malloc(10);
3. if(p==NULL)
4. printf("Not enough memory");
5. else
6. printf("Allocation successful");

```

**FIGURE 13.6**

Code segment showing the allocation of memory using the `malloc()` function.

`(int*)` before `malloc` is written in line 2 in Figure 13.6, which helps in casting the void pointer as an integer pointer. Note that we cast it as an integer pointer because `p` is an integer pointer. The following memory allocation statements give more clarity to the `malloc` declaration. You can observe that the casting portion of the code depends on the type of pointer to which the return address is assigned.



| Memory allocation for float                        | Memory allocation for char                       | Memory allocation for double                         |
|----------------------------------------------------|--------------------------------------------------|------------------------------------------------------|
| <pre> 1. float *p; 2. p=(float*)malloc(20); </pre> | <pre> 1. char *p; 2. p=(char*)malloc(20); </pre> | <pre> 1. double *p; 2. p=(double*)malloc(20); </pre> |

Consider Program 13.1 and its execution procedure to see how exactly memory gets allocated dynamically. Suppose you want to allocate memory for  $n$  integers. Assume that an integer takes 2 bytes; this means you need  $n \times 2$  bytes to be allocated in memory. If you don't know the size of an integer then you can write  $n \times \text{sizeof}(\text{int})$ . The `sizeof()` operator will return the size of the integer and we can then read the value of  $n$  from the user.

### PROGRAM 13.1

```

#include <stdio.h>
int main()
{
 int n, *p;
 printf("\n How many integer allocations: ");
 scanf ("%d", &n);
 p=(int*)malloc(n*sizeof(int));
 if(p!=NULL)
 printf("Allocation Successful");
 else
 printf("! Allocation Unsuccessful");
 return 0;
}

```

*Output:*

```
Run 1
How many integer allocations: 353646345234
! Allocation Unsuccessful
Run 2
How many integer allocations: 10
Allocation Successful
```

*Explanation:*

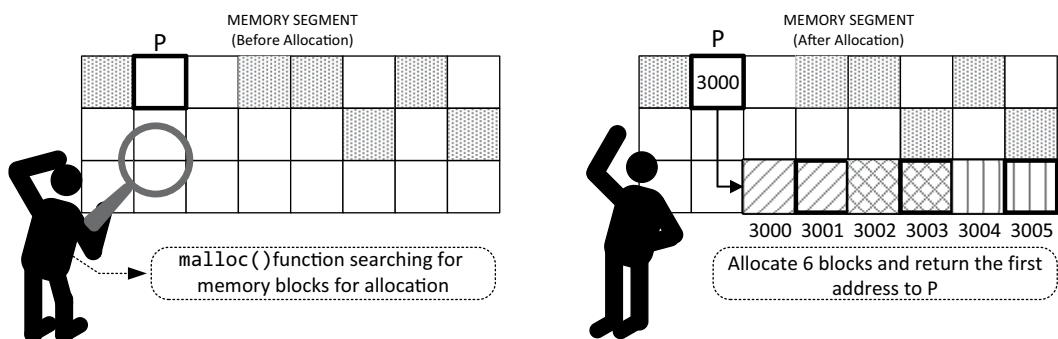
From the output, you can see that the allocation was unsuccessful in Run 1 because we specified a large number. In Run 2, we allocated for ten integers, and the allocation was successful.

Let us understand how the following line gets executed. Assume that  $n = 3$  and an integer takes 2 bytes.

```
p=(int*)malloc(3*2);
```

When the above statement gets executed, the malloc function searches for 6 bytes of free memory space to allocate. When the compiler finds the required number of free spaces in memory, it marks it and returns the address of the first block. The return address is assigned to the pointer p, and p starts pointing to the desired allocated space. The illustration of the above description is shown in Figure 13.7.

After we allocate memory space, we can store values in it. For the above example, we can store three integers in the allocated space. Let us write a complete program to allocate memory space for some integers, store some number in it, access them, and display them. Program 13.2 shows the entire code. Line 7 allocates the memory dynamically depending upon the size of n. Lines 12–14 are used to read the number from the user and assign it to the allocated space, like an array. Similarly, lines 16–18 display the content of the memory.



**FIGURE 13.7**  
Illustration of memory allocation by malloc().



### 13.3.2 The `calloc()` Function

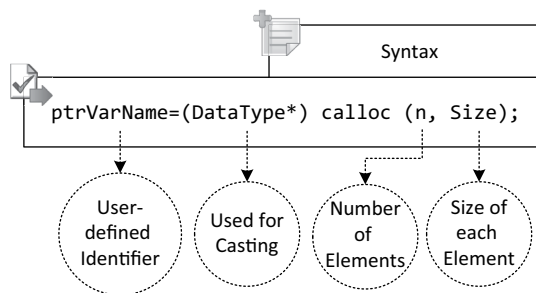
Like `malloc()`, `calloc()` also allocates a block of memory as and when required by the user. The syntax for the `calloc()` function is shown in Figure 13.8.

If you compare the syntax for `calloc()` with that for `malloc()`, you can see one difference: `malloc()` takes one argument, and `calloc()` takes two arguments. The functionality of both these functions is the same. The `calloc()` function allocates ( $n \times \text{size}$ ) number of bytes and returns the first address of the allocated block. The return address is a void pointer; casting is needed to assign the address to the appropriate pointer.

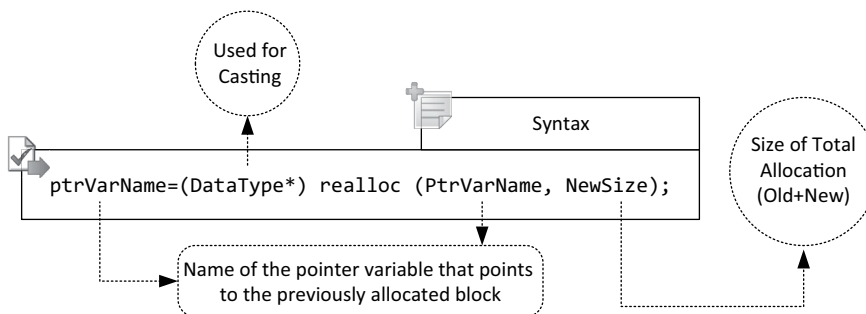
Let us rewrite Program 13.2 using the `calloc()` function. Program 13.3 shows the changes made to the program and its output.

### 13.3.3 The `realloc()` Function

This function is used to resize the memory allocated previously by `malloc()`, `calloc()`, or `realloc()`. It does not affect the content of the previously allocated memory blocks. This function tries to expand the space by allocating new blocks. But, if enough space is not available for continuous allocation, it can relocate the allocation along with the previously allocated block to a new place in memory. If the allocation is successful, the function returns the address of the newly allocated block. If it fails to allocate the new allocation (maybe due to limited size), it returns NULL. The syntax for the `realloc()` function is shown in Figure 13.9.



**FIGURE 13.8**  
Syntax for the `calloc()` function.



**FIGURE 13.9**  
Syntax for the `realloc()` function.

## PROGRAM 13.2

```

1. #include <stdio.h>
2. int main()
3. {
4. int n, i, *p;
5. printf("\n How many integer allocations: ");
6. scanf ("%d", &n);
7. p=(int*)malloc(n*sizeof(int));
8. if(p!=NULL)
9. {
10. printf("Allocation Successful");
11. /*Code for reading integers & storing them in the
12. allocated space*/
13. printf("\nEnter %d elements: ", n);
14. for(i=0;i<n;i++)
15. scanf("%d", &p[i]);
16. /*Code for displaying the numbers present in the
17. allocated space*/
18. printf("\nThe numbers are: ");
19. for(i=0;i<n;i++)
20. printf("%d ", p[i]);
21. }
22. else
23. printf("! Allocation Unsuccessful");22. return 0;

```

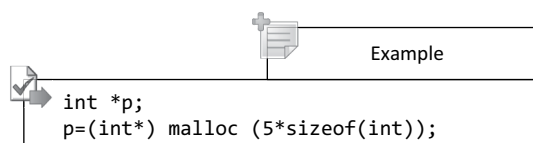
*Output:*

```

Run 1
How many integer allocations: 5
Allocation Successful
Enter 5 elements: 25 43 71 83 44
The numbers are: 25 43 71 83 44
Run 2
How many integer allocations: 3536346346463
! Allocation Unsuccessful

```

To understand how the new allocation takes place, let us begin with the following allocation with `malloc()`.



## PROGRAM 13.3

```

1. #include <stdio.h>
2. int main()
3. {
4. int n, i, *p;
5. printf("\n How many integer allocations: ");
6. scanf ("%d", &n);
7. p=(int*)calloc(n, sizeof(int)); /* Changes made on this line
 only*/
8. if(p!=NULL)
9. {
10. printf("Allocation Successful");
11. /*Code for reading integers & storing them in the
 allocated space*/
12. printf("\nEnter %d elements: ", n);
13. for(i=0;i<n;i++)
14. scanf("%d", &p[i]);
15. /*Code for displaying the numbers present in the
 allocated space*/
16. printf("\nThe numbers are: ");
17. for(i=0;i<n;i++)
18. printf("%d ", p[i]);
19. }
20. else
21. printf("! Allocation Unsuccessful");
22. return 0;
23. }

```

*Output:*

Run 1

How many integer allocations: 5

Allocation Successful

Enter 5 elements: 25 43 71 83 44

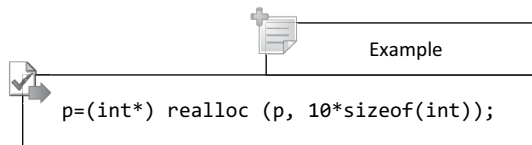
The numbers are: 25 43 71 83 44

Run 2

How many integer allocations: 3536346346463

! Allocation Unsuccessful

The `malloc()` function allocates ten bytes of memory (assuming that an integer takes two bytes) and returns the first address to `p`. The use of the `realloc()` function is required only when we want to increase the size of the above-allocated memory blocks. Suppose we want to allocate memory blocks for ten integers. The following line of code will do that using the `realloc()` function:



The above line of code will allocate 20 bytes without affecting the previous content and returns the first address to p. Consider Program 13.4.

#### PROGRAM 13.4

```
#include <stdio.h>
int main()
{
 int n, m, i, *p;
 int option;
 printf("\n How many integer allocations: ");
 scanf ("%d", &n);
 p=(int*)malloc(n*sizeof(int)); /* Changes made on this line
 only*/

 if(p!=NULL)
 {
 printf("Allocation Successful");
 /*Code for reading integers & storing them in the allocated
 space*/
 printf("\nEnter %d elements: ", n);
 for(i=0;i<n;i++)
 scanf("%d", &p[i]);
 printf("\nDo you want to enter more numbers?");
 printf("\nPress 1 for YES and 0 for No: ");
 scanf("%d", &option);
 if(option==1)//(option=='y' || option=='Y')
 {
 printf("\nHow many integer allocations: ");
 scanf ("%d", &m);
 p=(int*)realloc(p, (n+m)*sizeof(int));
 if(p!=NULL)
 {
 printf("Allocation Successful");
 /*Read numbers and store them in reallocated space*/
 printf("\nEnter %d elements: ", m);
 for(i=n;i<n+m;i++)
 scanf("%d", &p[i]);
 }
 }
 }
}
```

```

 else
 {
 printf("Reallocation Unsuccessful");
 }
 }
 /*Code for displaying the numbers present in the allocated
 space*/
 printf("\nThe numbers are: ");
 for(i=0;i<n+m;i++)
 printf("%d ", p[i]);
 }
 else
 printf("! Allocation Unsuccessful");
 return 0;
}

```

*Output:*

```

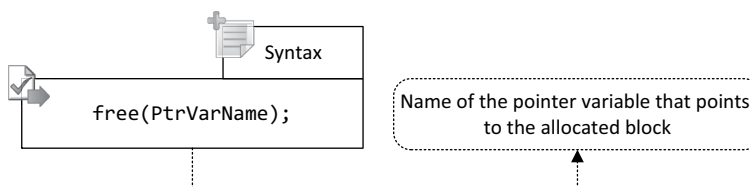
How many integer allocations: 5
Allocation Successful
Enter 5 elements: 12 34 87 35 62
Do you want to enter more numbers?
Press 1 for YES and 0 for No: 1
How many integer allocations: 5
Allocation Successful
Enter 5 elements: 11 33 44 22 66
The numbers are: 12 34 87 35 62 11 33 44 22 66

```

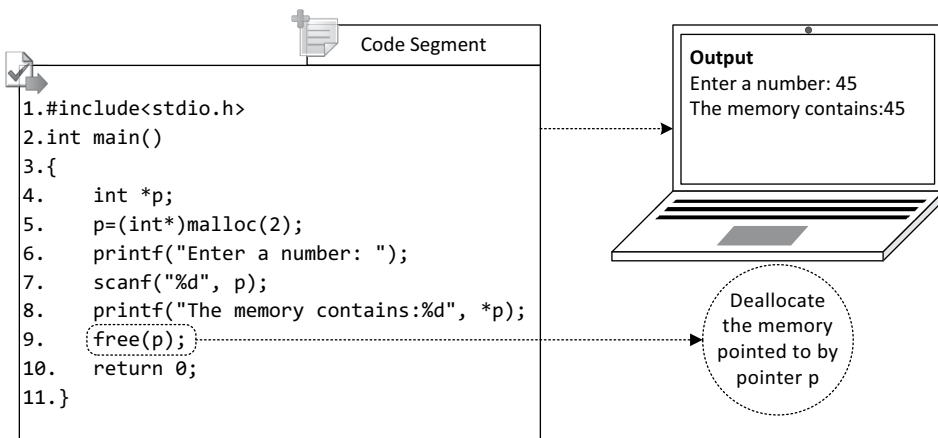
**13.3.4 The free () Function**

In the previous section, we covered the memory allocation process using three functions. In this section, we will look at how to deallocate them after the execution process is over. Deallocation of memory is necessary to optimize memory usage. The C language provides the `free()` function to deallocate the memory space allocated by `malloc()`, `calloc()`, and `realloc()`. The general structure of the `free()` function is shown in Figure 13.10.

Consider the program shown in Figure 13.11. The program allocates two bytes using the `malloc()` function and returns the address to the pointer `p` (line 5). It reads an integer from



**FIGURE 13.10**  
Syntax for `free()` function.

**FIGURE 13.11**

Example showing the uses of the `free()` function to deallocate memory.

the user and assigns it to the allocated location (lines 6 and 7). We assume the size of an integer is two bytes. Line 8 displays the content of the allocated memory, and the output is shown in the output block of Figure 13.11. After it produces the output, it is the programmer's responsibility to deallocate the memory. That's why the programmer included line 9, which uses the `free()` function to free up the memory allocated by the `malloc()` function.

## 13.4 Review Questions

1. Consider the following code.

```

int* A = malloc(4*n);
int *B = A;
free(B);

```

Does this free the original memory?

2. Write down the output of the following code.

```

int main()
{
 clrscr();
 int n=50;
 int *p;
 p=&n;
 printf("Address of n variable is %x \n",&n);
 printf("Address of p variable is %x \n",p);
 printf("Value of p variable is %d \n",*p);
 getch();
}

```

3. Write down the output of the following code.

```
int main()
{
 int s = 4;
 char *string1 = (char *)malloc(sizeof(char)*s);
 *(string1+0) = 'G';
 *(string1+1) = 'f';
 *(string1+2) = 'G';
 *(string1+3) = '\\0';
 *(string1+1) = 'n';
 getchar();
 return 0;
}
```

4. What is the need for dynamic memory allocation?
5. What is the difference between a stack and a heap?
6. Explain the different types of memory allocation possible in the C language.
7. What are the different memory management functions available in C?
8. Write down the syntax for the `malloc()` and `calloc()` functions.
9. What is the difference between the `malloc()` and `calloc()` functions? Which one is better?
10. What is memory leak and how can we avoid it?
11. Why is casting needed during dynamic memory allocation using `malloc`, `calloc`, and `realloc`?
12. What is the `free()` function and why is it used?
13. Explain the process of memory allocation using the `malloc()` function.
14. Which header file should be included in dynamic memory allocation?
15. What is the difference between static and dynamic memory allocations?
16. What functions are used to allocate memory dynamically in C programs?
17. What is the disadvantage of dynamic memory allocation in C?
18. Write a C program to create memory for `int`, `char`, and `float` variables at run time.
19. What is memory leak in C?
20. Explain the statement: `ptr = (int*) malloc(100 * sizeof(int));`
21. Write C code to inform the user when space is insufficient, allocation fails, and returns a `NULL` pointer.
22. Which method in C is used to dynamically deallocate memory?
23. Write a program to read and print an integer array. The program should input the total number of elements (limit) and the elements for the array from the user. Use dynamic memory allocation to allocate and deallocate array memory.
24. How can you determine the size of an allocated portion of memory?
25. What is the purpose of `realloc()`? Write a C program to extend the size of memory block A to double its size.

26. What is dynamic memory allocation? Write down and explain the different dynamic memory allocation functions in C.
27. Write a C program to read a 1D array and print the multiplication of all the elements along with the inputted array elements using dynamic memory allocation.
28. Write a C program to read and print patient details using structure and the `malloc()` function in order to dynamically allocate memory.
29. Write C code for creating a dynamic string.
30. Write C code to read one character at a time (using `getc(stdin)`) and grow the string (`realloc`) as you go.
31. Write a C program to read and print the details of person N using structure and the `calloc()` function to dynamically allocate memory.
32. Write a C program to find the largest number using dynamic memory allocation.





**Taylor & Francis**

Taylor & Francis Group

<http://taylorandfrancis.com>

# 14

---

## *File Handling*

---

### 14.1 Introduction

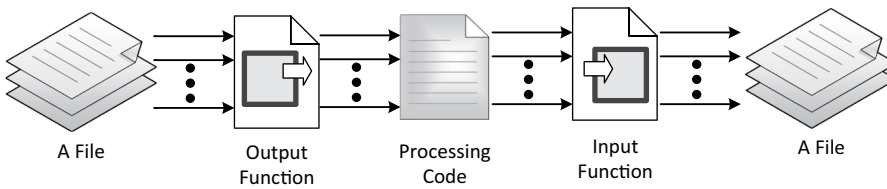
I believe most of the readers of this book know of at least one kind of word-processing software. Every mobile phone today has a word processor installed in it. Consider also a notepad where we type (write) something and store it in our local drive. In the future, we can then access its content anytime. We add (write) new content, remove (delete) the existing content, copy the content from one to another, or we can read the content from it. A notepad is a piece of software that gives us the facility to create a file and allow us to perform the several kinds of operations mentioned above. C programming also provides us with some predefined functions that will enable us to create a file or read an existing file and perform several operations in it. In Chapter 7, Section 7.1 we classified the I/O function into three groups: (i) console I/O (ii), file I/O, and (iii) port I/O. The file I/O functions provided by C enable us to perform several operations on files. Generally, files reside on our hard disk drive. In this chapter, we will learn how to access these files, read the existing content from it, write new content in it, and so on, by writing C code. The overall process flow of the file handling process is shown in Figure 14.1.

Files reside in our local drive and are available on demand. The output functions extract the content of the file and send it for processing. After the processing is completed, the updated content is either written to the file or displayed on the screen using the input function. Here input functions are used for inputting (writing) the content to a file. The output functions are used for extracting (reading) the content of the file.

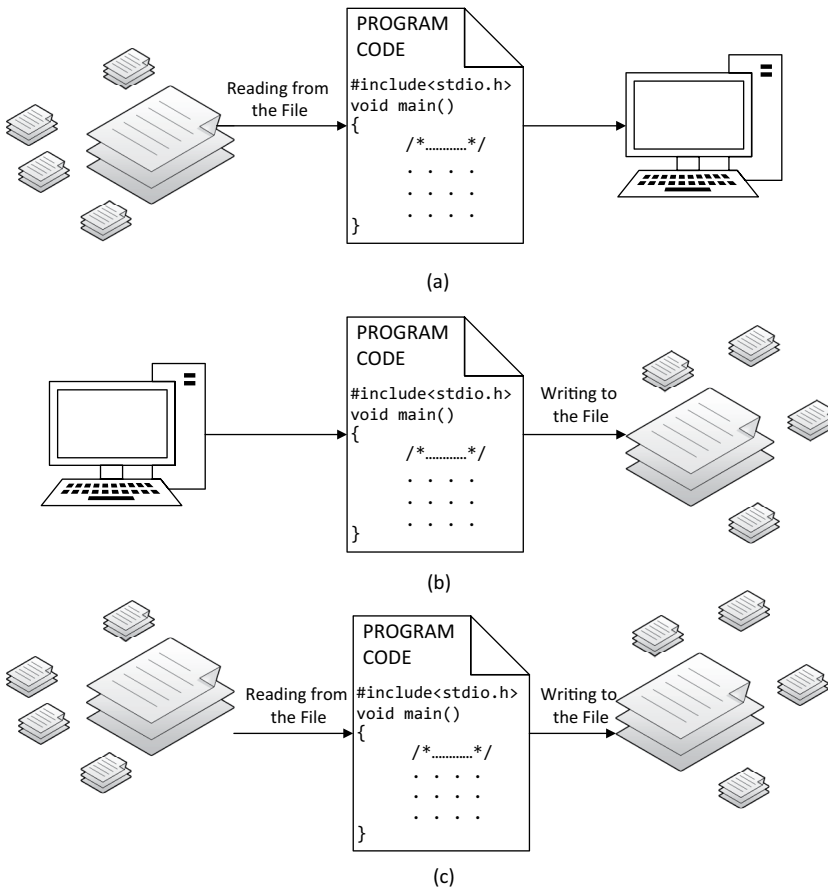
This entire chapter is about writing code that performs several operations on a file. We categorize these operations into three groups as shown in Figure 14.2. The first group describes the output functions that help us in reading (extracting) the content from a file and displaying it on a screen using a console output function like `printf()`. The second group explains the functions available for writing new content to a file by reading it from the console input. The third group describes the program code for how to perform read/write operations on a file using I/O functions, as explained by the two groups above.

After finishing this chapter, the student will be able to:

- Define and differentiate between file I/O and console I/O;
- Open a file in different modes, such as read mode, write mode, and append mode;
- Use several file input and output functions along with their syntax and the way they are used inside our programs;
- Copy the content of one file to another file;
- Know about file streams and types of files.



**FIGURE 14.1**  
Process flow of file handling.



**FIGURE 14.2**  
Overall operations on a file.

Before a discussion on file handling, let us understand the difference between console I/O and file I/O. This is just an extension of the content of Chapter 7.

### 14.1.1 Difference between Console I/O and File I/O

- A console (keyboard and monitor) always exists but a file may or may not exist.
- Console inputs are taken from a keyboard and displayed on a screen. But input and output may be done on the same file.

As we know, “console” means the input and output devices connected to our computer. When we write code to read content through an input device and display it on the output device, we call it console I/O. When we do the same with files, we call it file I/O.

---

## 14.2 Basics of File I/O

This section introduces the file system and the steps we follow to process files. We begin with what a file is and gradually show you how to process files using the I/O functions available to us.

### 14.2.1 What is a File?

In C, a file refers to a disk file that stores some information in textual form and which is sometimes referred to as a data file. We can create a data file, write some information to it, store it in an appropriate place, and in the future access it to read/write or update the information present in it.

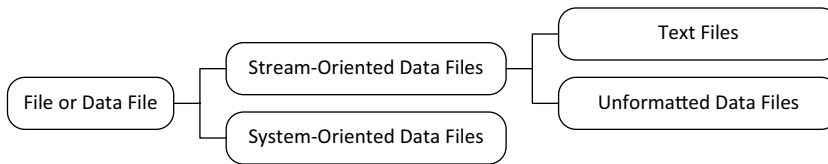
In C, a file refers to a disk file that stores some information in textual form and which is sometimes referred to as a data file.

A data file is of two types: (1) stream-oriented data files and (2) system-oriented data files. *System-oriented data files* are system files or low-level files and which are difficult to manage. In this chapter, we will concentrate on stream-oriented files. *Stream-oriented data files* are again divided into two types: (1) text files and (2) unformatted data files. *Text files* store the information either as characters or numbers. The C language provides several library functions to read/write the information, character by character, from these files. The other option is to read/write string-wise. *Unformatted data files* store the information in the form of continuous blocks. Specialized data structures and functions are required to store and retrieve information from this type of file. Data structures, like arrays and arrays of structures, will be used to store data inside this type (see Figure 14.3).

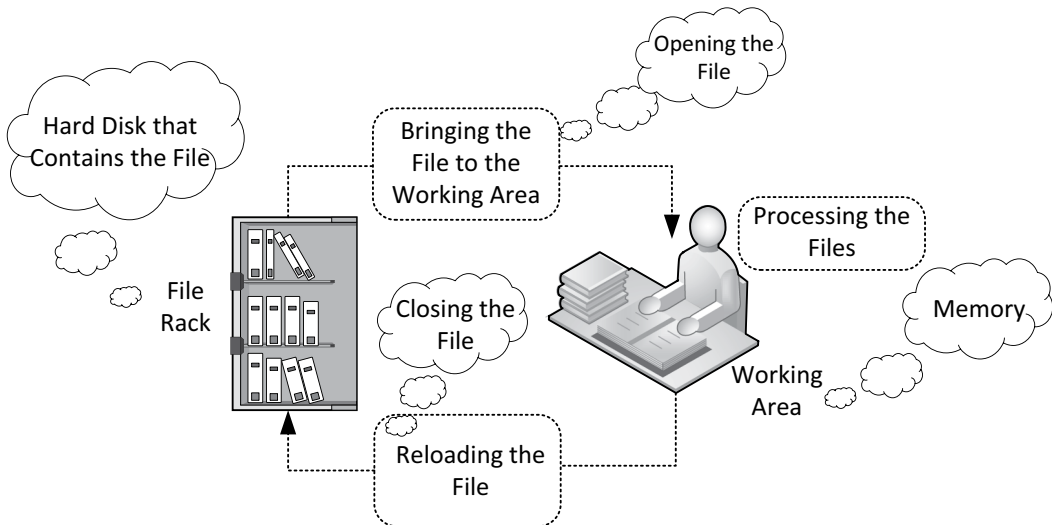
### 14.2.2 File Handling Process Flow

Before performing any operation on a file, the file must be loaded into the memory first. Loading the file from the hard disk to the memory is known as opening the file. There are several ways to open a file. After opening, we perform various operations on it as per our requirements; finally we close the file. Closing means reloading the file onto the disk where it came from. So, the process flow comprises three tasks:

1. Opening a file;
2. Perform operations like read/write;
3. Close the file.



**FIGURE 14.3**  
Classification of file types.



**FIGURE 14.4**  
Analogy illustrating the file handling process flow.

Figure 14.4 shows an analogy that illustrates the file handling process flow. A file rack is shown in the figure that contains all the files of a department. One officer works upon this file. Every day, he brings the file from the rack to his working area, processes it, and at the end of the day, he put it back on the rack. To understand the process flow, we can assume that the file rack is the hard disk that contains our files. Bringing the file to the working area is the same as opening the file. The working area is the memory where we process the file. Finally, putting it back to the file rack is the same as closing the file.

In the following section, we will discuss every step in detail. To execute the process flow, the C language provides us with a set of library functions. The definition of these functions is available in the header file `<stdio.h>`. Table 14.1 describes all the functions.

## 14.3 Opening a File

Before we read (or write) information from (to) a file on a disk, we must open the file. To open the file, we have a function known as `fopen()`, which returns the file pointer to the opened file. In fact, `fopen()` performs three important tasks when you open the file:

- First, it searches on the disk for the requested file;

**TABLE 14.1**

List of File Processing Functions

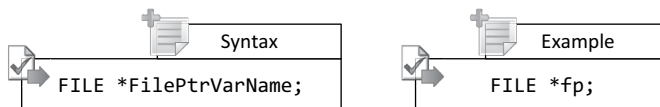
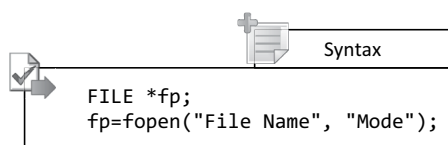
| Serial No. | Function Name          | Description                                                             |
|------------|------------------------|-------------------------------------------------------------------------|
| 1          | <code>fopen()</code>   | Create a new file or open an existing file                              |
| 2          | <code>fclose()</code>  | Close a file                                                            |
| 3          | <code>fprintf()</code> | Write a set of formatted data values to a file                          |
| 4          | <code>fscanf()</code>  | Read a set of formatted data values from a file                         |
| 5          | <code>fputc()</code>   | Write a character to a file                                             |
| 6          | <code>fgetc()</code>   | Read a character from a file                                            |
| 7          | <code>fputs()</code>   | Write a string to a file                                                |
| 8          | <code>fgets()</code>   | Read a string from a file                                               |
| 9          | <code>putw()</code>    | Write an integer to a file                                              |
| 10         | <code>getw()</code>    | Read an integer from a file                                             |
| 11         | <code>fseek()</code>   | Set the position to desired point in the file                           |
| 12         | <code>ftell()</code>   | Determine current position in a file (in terms of bytes from the start) |

- Then it loads the file from the disk into a place in memory called the buffer;
- It sets up a pointer that points to the first character of the buffer.

We need to establish a buffer area where the information will be stored temporarily. From this area, the content will be transferred to the data file. The buffer area will be created by declaring a pointer variable of `FILE` type. `FILE` (note the capital letter) is a predefined structure that helps us to create a buffer. The syntax for and example of creating a buffer are shown in Figure 14.5.

After creating the buffer, the next step is to open the file using `fopen()`. This will be done using the syntax given in Figure 14.6.

Here the filename specifies the name of the file that exists on the disk, or is to be created. The file will be opened if a file exists with that name. If not, a new file with the same name will be created. But, the creation of a file depends on the "mode" specified in your code.

**FIGURE 14.5**  
Creating a buffer.**FIGURE 14.6**  
Syntax for `fopen()`.

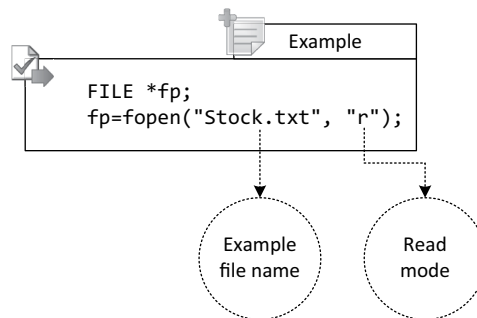
**TABLE 14.2**  
Various Modes of Opening a File and Their Meaning

| Mode | Meaning                                                                                                                                                                                                        |
|------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| r    | Open file in read mode <ul style="list-style-type: none"> <li>• If file exists, the pointer is positioned at beginning</li> <li>• If the file doesn't exist, error returned</li> </ul>                         |
| w    | Open file in write mode <ul style="list-style-type: none"> <li>• If file exists, it is erased</li> <li>• If the file doesn't exist, it is created</li> </ul>                                                   |
| a    | Open file in append mode <ul style="list-style-type: none"> <li>• If file exists, the pointer is positioned at end</li> <li>• If the file doesn't exist, it is created</li> </ul>                              |
| r+   | Open file in update (both read and write) mode <ul style="list-style-type: none"> <li>• If file exists, the pointer is positioned at beginning</li> <li>• If the file doesn't exist, error returned</li> </ul> |
| w+   | Open file in update (both read and write) mode <ul style="list-style-type: none"> <li>• If file exists, it is erased</li> <li>• If the file doesn't exist, it is created</li> </ul>                            |
| a+   | Open file in update (both read and write) mode <ul style="list-style-type: none"> <li>• If file exists, the pointer positioned at end</li> <li>• If the file doesn't exist, it is created</li> </ul>           |

Mode points to another string that specifies the way (read mode, write mode, etc.) to open the file. The `fopen()` function returns a pointer of type `FILE`. If an error occurs during the procedure to open, the `fopen()` function returns a `NULL` pointer.

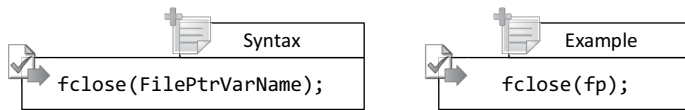
Table 14.2 shows all the modes of opening a file and their description

An example of file open can be viewed as follows:



## 14.4 Closing a File

We need to close a file after we finish all the operations on it. By doing this, we will deallocate the resources allocated to that file. The required content will be updated on the file,



**FIGURE 14.7**  
Syntax and example of `fclose()`.

and it will be ready for reopening in another mode. The buffer is flushed out, and the pointer to that location will be broken. Closing the file is also necessary for unauthorized access. To close the file, we use the `fclose()` function.

The syntax for and an example of writing the `fclose()` function is shown in Figure 14.7.

Sometimes when we open or close a file, it shows errors, for many reasons. One of the most common errors occurs when the external file name does not match a name on the disk.

- Always check to make sure that a stream has opened successfully. If it succeeds, then we have a valid address in the file pointer. But if it fails for any reason, the stream pointer variable contains NULL.
- Similarly, we can test the return value from `fclose()` to make sure it has closed successfully. The `fclose()` function returns an integer that is zero if the close succeeds and EOF if there is an error. EOF stands for end of file.

To check the open or closing error, we need an if statement. The following program segment shows the general structure of writing any file programs (Figure 14.8).

From the next section onwards, we will write code for processing a file, which means reading from or writing to a file. During the code writing, we will introduce several file functions and explain their syntaxes.

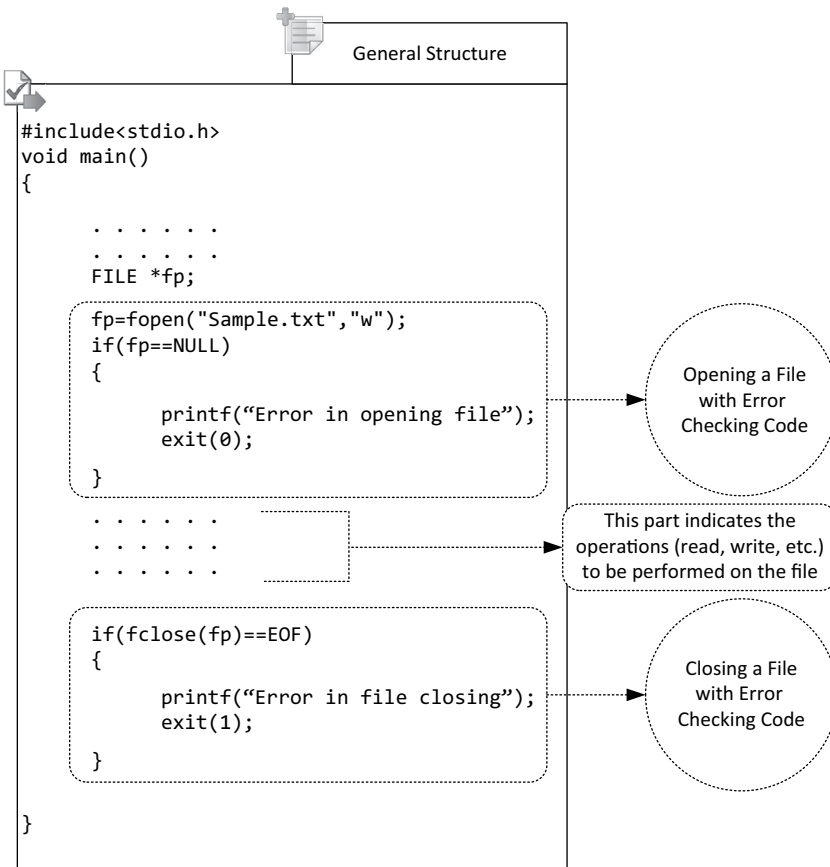
## 14.5 File Functions with Examples

This section will describe different file functions available in C to perform reading or writing operations. Every function will be explained by using several appropriate examples. Let us begin with the `fprintf()` and `fscanf()` functions.

### 14.5.1 The `fprintf()` and `fscanf()` Functions

The first set of functions we discuss here is `fprintf()` and `fscanf()`. The names are analogous to the console I/O functions `printf()` and `scanf()` but with an extra `f`. The syntax of both the functions is very similar to the console I/O functions, but takes an extra argument: the name of the file pointer. The `fscanf()` function extracts the content of the file, and the `fprintf()` function writes the given information to the file. The syntax of these two functions is given below.





**FIGURE 14.8** General structure of a file program with error checking code.

```

Syntax
fprintf(FilePtr, "Format String", VarNames);

```

(a) fprintf() syntax

```

Syntax
fscanf(FilePtr, "Format String", &VarNames);

```

(b) fscanf() syntax

**14.5.1.1 Writing and Reading an Integer Using fprintf() and fscanf()**

Write a program to create file. Write an integer to it. Read the integer and display it on the screen.

**PROGRAM 14.1**

```
1. #include<stdio.h>
2. #include<stdlib.h>
3. void main()
4. {
5. int x;
6. FILE *fp; /* Declaration of File Pointer*/
7. fp=fopen("Sample.txt","w");
8. if(fp==NULL)
9. {
10. printf("Error in opening file");
11. exit(0);
12. }
13.
14. printf("Enter a number ");
15. scanf("%d",&x);
16. fprintf(fp,"%d",x); /* Writing the value of x to file*/
17. if(fclose(fp)==EOF)
18. {
19. printf("Error in file closing");
20. exit(1);
21. }
22.
23. fp=fopen("Sample.txt","r");
24. fscanf(fp,"%d",&x); /*Reading the content of file and assign
it to x*/
25. printf("File contains=%d",x);
26. if(fclose(fp)==EOF)
27. {
28. printf("Error in file closing");
29. exit(1);
30. }
31. }
```

*Output:*

```
Enter a number 25
File contains=25
```

*Explanation:*

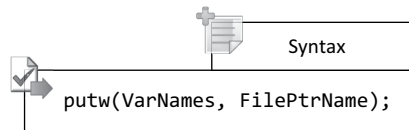
Lines 7–12: Open a file in write mode.

Lines 14–15: Read an integer from the user using console and store it in x. Here x is an integer variable.

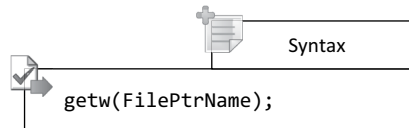
Line 16: Write the value of `x` to the file using `fprintf()` function.  
 Lines 17–21: Close the file.  
 Line 23: Open the same file in read mode  
 Line 24: Read the content of the file using `fscanf()` function and assign it to the variable `x`.  
 Line 25: Print the value of `x` on the screen which is the content of the file.  
 Lines 26–30: Close the file.

### 14.5.2 The `putw()` and `getw()` Functions

Here we discuss another two functions: `putw()` and `getw()`. We use the `putw()` function to write a series of integers to a file, and the `getw()` function retrieves the integers from the file. The general syntax of these two functions is given below.



(a) `putw()` syntax



(a) `getw()` syntax

#### 14.5.2.1 Writing and Reading More than One Integer Using the `putw()` and `getw()` Functions

Write a program to create a file, store some integers, and display the content.

#### PROGRAM 14.2

```

1. #include<stdio.h>
2. void main()
3. {
4. int x,i,n;
5. FILE *fp;
6. fp=fopen("Sample.txt","w");
7. printf("How many numbers do you want? ");
8. scanf("%d", &n);
9. for(i=0;i<n;i++)
10. {
11. printf("Enter numbers:");

```

```
12. scanf ("%d", &x) ;
13. putw(x, fp) ;
14. }
15. fclose(fp) ;
16.
17. fp=fopen("Sample.txt", "r") ;
18. while((x=getw(fp)) !=EOF)
19. {
20. printf("\nFile contains=%d", x) ;
21. }
22. fclose(fp) ;
23. }
```

*Output:*

```
How many numbers do you want? 3
Enter numbers:25
Enter numbers:49
Enter numbers:43
File contains=25
File contains=49
File contains=43
```

*Explanation:*

Line 6: Open the file in write mode.

Lines 7–8: Ask the user to enter how many integers are to be stored in the file.

Lines 9–14: Read the numbers from the user through the console, store it in a variable *x*. “*x*” is an integer here. Write the value of *x* to the file using the `putw()` function.

Line 15: Close the file.

Line 17: Open the file in read mode.

Line 18–21: Read the numbers present in the file until EOF (end of the file) using `getw()` function and display the numbers on the screen using `printf()` function.

Line 22: Close the file.

In the next example, we will read the numbers present in a file and check for odd or even numbers. We copy all the odd numbers to a file named “odd.txt”. Copy the even numbers to a file named “even.txt”. Finally, we display the content of both files. In this example, we need to open three files simultaneously. The first file contains all the numbers, and we should open it in read mode. The other two files must be opened in write mode because we read the numbers from the first file and check it for even or odd, and finally, write it to the appropriate file. Finally, we should close all the files.

#### **14.5.2.2 Reading Numbers from a File and Checking Them for Even or Odd**

Write a program to read an integer file, copy all even numbers to a file named “even”, and copy all odd numbers to another file named “odd”. Display the content of all the files.

**PROGRAM 14.3**

```
1. #include<stdio.h>
2. void main()
3. {
4. int x,i,n;
5. FILE *fp,*fe,*fo;
6. fp=fopen("Sample.txt","w");
7. printf("How many numbers do you want? ");
8. scanf("%d", &n);
9. for(i=0;i<n;i++)
10. {
11. printf("Enter numbers:");
12. scanf("%d",&x);
13. putw(x,fp);
14. }
15. fclose(fp);
16.
17. fp=fopen("Sample.txt","r");
18. fe=fopen("even.txt","w");
19. fo=fopen("odd.txt","w");
20.
21. while((x=getw(fp))!=EOF)
22. {
23. if(x%2==0)
24. putw(x,fe);
25. else
26. putw(x,fo);
27. }
28. fclose(fe);
29. fclose(fo);
30.
31. fe=fopen("even.txt","r");
32. fo=fopen("odd.txt","r");
33. printf("\nThe odd file contains:");
34. while((x=getw(fo))!=EOF)
35. printf("%d\t",x);
36.
37. printf("\nThe even file contains:");
38. while((x=getw(fe))!=EOF)
```

```

39. printf("%d\t",x) ;
40. fclose(fp) ;
41. fclose(fe) ;
42. fclose(fo) ;
43. }

```

*Output:*

```

How many numbers do you want? 10
Enter numbers:12
Enter numbers:34
Enter numbers:56
Enter numbers:78
Enter numbers:43
Enter numbers:23
Enter numbers:45
Enter numbers:88
Enter numbers:21
Enter numbers:22
The odd file contains:

```

|    |    |    |     |
|----|----|----|-----|
| 43 | 23 | 45 | 215 |
|----|----|----|-----|

The even file contains:

|    |    |    |    |    |    |
|----|----|----|----|----|----|
| 12 | 34 | 56 | 78 | 88 | 22 |
|----|----|----|----|----|----|

*Explanation:*

Line 5: Declare three FILE pointers because we need to open three files simultaneously as discussed above.

Lines 6–15: Open a file Sample.txt, store some integers in it by reading the numbers from the console as in Program 14.2, and finally close the file in line 15.

Lines 17–19: Open three files: Sample.txt opened in read mode, but even.txt and odd.txt opened in write mode.

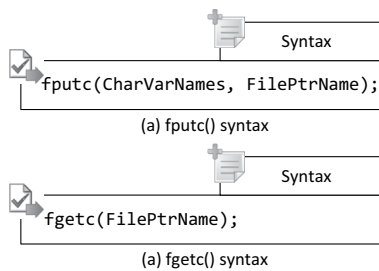
Lines 21–27: Read the number from Sample.txt using the function `getw()` until EOF, check for even or odd, copy the even numbers to the file even.txt, and copy the odd numbers to the file odd.txt using `putw()` function.

Lines 28–29: Close both files even.txt and odd.txt.

Lines 31–39: Open the files in read mode. Read the content of odd.txt and even.txt and display it on the screen.

### 14.5.3 The `fputc()` and `fgetc()` Functions

Until now, we have seen how to read and write numbers in a file. In this section, we will introduce you to another two functions for reading and writing characters in a file: `fgetc()` and `fputc()`. `fgetc()` reads one character from a file, and `fputc()` writes one character into a file. The syntax of these two functions is shown below.



### 14.5.3.1 Writing and Reading a Character Using *fputc()* and *fgetc()*

Write a program to create a file, input a character, and display the character.

#### PROGRAM 14.4

```

1.#include<stdio.h>
2.void main()
3.{
4. char ch;
5. FILE *fp;
6. fp=fopen("Charfile.txt", "w");
7. printf("Enter a character: ");
8. ch=getchar();
9. fputc(ch, fp);
10. fclose(fp);
11.
12. fp=fopen("charfile.txt", "r");
13. ch=fgetc(fp);
14. printf("The file contain=%c", ch);
15. fclose(fp);
16.}

```

#### Output:

```

Enter a character: s
The file contain=s

```

#### Explanation:

- Line 5: Declare a file pointer.
- Line 6: Open a file named Charfile.txt in read mode.
- Lines 7–8: Read a character through console using `getchar()` function and store it in a character variable `ch`.
- Line 9: Write the character to the file using `fputc()` function.
- Line 10: Close the file.
- Line 12: Read the same file in read mode.
- Line 13: Read the character from the file using `fgetc()` function, and assign it to the variable `ch`.
- Line 14: Print the value of `ch` onto the screen using `printf()` function.
- Line 15: Close the file.

### 14.5.3.2 Writing and Reading Multiple Characters Using `fputc()` and `fgetc()`

The next program will show you how to write and read multiple characters in a file using the `fputc()` and `fgetc()` functions. As these two functions can only read one character at a time, so we need a looping construct to read multiple characters.

Write a program to create a file, input some characters, and display the contents of the file on the screen.

#### PROGRAM 14.5

```
1.#include<stdio.h>
2.void main()
3.{
4. char ch;
5. FILE *fp;
6. fp=fopen("Charfile.txt","w");
7. printf("Enter characters and press ctrl+z to stop");
8. while(1) /* 1 represents condition is always true*/
9. {
10. ch=getchar();
11. if(ch==EOF)
12. break;
13. fputc(ch,fp);
14. }
15. fclose(fp);
16.
17. fp=fopen("Charfile.txt","r");
18. printf("\nYour file contains:\n");
19. while (1) /* 1 represents condition is always true*/
20. {
21. ch = fgetc (fp) ;
22. if (ch == EOF) /* HasIs ch reacheds EOF?*/
23. break ;
24. printf ("%c", ch) ;
25. }
26. fclose(fp);
27.}
```

#### Output:

```
Enter characters and press ctrl+z to stop
C Programming Learn to Code
File Handling
Nonstop Writing
Continue typing
Let us see the output^z
Your file contains:
```



C Programming Learn to Code  
 File Handling  
 Nonstop Writing  
 Continue typing  
 Let us see the output

*Explanation:*

Lines 7–14: Read the character through the console and write it to file (line 13). The loop will stop only when the user presses ctrl+z.

Lines 18–25: Read the characters from the file using `fgetc()` (line 21) until EOF and write to the screen using `printf()` function (line 24).

### 14.5.3.3 Count Number of Characters, Lines, Tabs, and Blank Spaces Present in a File

Using the `getc()` function, we can read the content of a file character by character, and if we do so, then we can compare each character for blank spaces, next line, tabs, and so on. Program 14.6 shows the complete code and is self-explanatory.

Write a program to create a file, input some lines of characters, and count the number of characters, number of lines, number of tabs, and blank spaces present in the file.

#### PROGRAM 14.6

```

1. #include<stdio.h>
2. void main()
3. {
4. char ch;
5. int noc=0,nol=0,not=0,nob=0;
6. FILE *fp;
7. fp=fopen("Charfile.txt","w");
8. printf("Enter a characters and press ctrl+z to stop");
9. while(1)
10. {
11. ch=getchar();
12. if(ch==EOF)
13. break;
14. fputc(ch,fp);
15. }
16. fclose(fp);
17.
18. fp=fopen("Charfile.txt","r");
19. while (1)
20. {
21. ch = fgetc (fp) ;
22. if (ch == EOF)
23. break ;

```

```

24. noc++ ;
25. if (ch == ' ')
26. nob++ ;
27. if (ch == '\n')
28. nol++ ;
29. if (ch == '\t')
30. not++ ;
31. }
32. fclose (fp) ;
33. printf ("\nNumber of characters = %d", noc) ;
34. printf ("\nNumber of blanks = %d", nob) ;
35. printf ("\nNumber of tabs = %d", not) ;
36. printf ("\nNumber of lines = %d", nol) ;
37. }

```

**Output:**

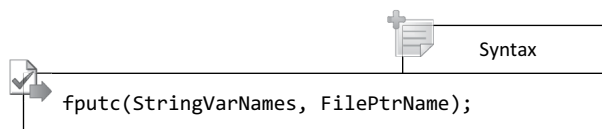
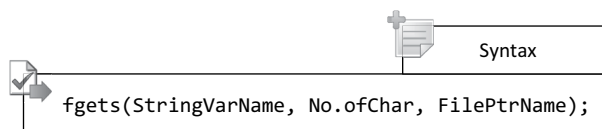
```

Enter a character and press ctrl+z to stop
C Programming Language
LEarn to Code
Coding is fun it is ok
Learning id 12345
Learn^z
Number of characters = 84
Number of blanks = 8
Number of tabs = 3
Number of lines = 5

```

**14.5.4 The `fputs()` and `fgets()` Functions**

In the previous section, we learnt how to read/write the numbers and characters in a file. In this section, we will see how to read/write the strings in a file. C provides two built-in functions for this purpose: `fputs()` and `fgets()`. Here also the names are analogous to the console I/O functions `puts()` and `gets()` with an extra `f`. The syntax of both functions is very similar to the console I/O function, but takes an extra argument: the name of the file pointer. The `fgets()` function extracts the content of the file, and the `fputs()` function writes the given information to the file. The syntax of these two functions is given below.

(a) `fputs()` syntax(a) `fgets()` syntax

**14.5.4.1 Writing and Reading a String Using `fputs()` and `fgets()`**

Write a program to write a string to a file, then read the contents of the file and display it on the screen.

**PROGRAM 14.7 (A)**

```
1.#include<stdio.h>
2.void main()
3.{
4. char str[30];
5. FILE *fp;
6. fp=fopen("Sample.txt","w");
7. printf("Enter a string: ");
8. gets(str);
9. fputs(str,fp);
10. fclose(fp);
11.
12. fp=fopen("Sample.txt","r");
13. printf("The file contains\n");
14. fgets(str,30,fp);
15. puts(str);
16. fclose(fp);
17.}
```

OR

**PROGRAM 14.7 (B)**

```
1.#include<stdio.h>
2.void main()
3.{
4. char str[30];
5. FILE *fp;
6. fp=fopen("Sample.txt","w");
7. fputs("Learn to Code Series C Programming",fp);
8. fclose(fp);
9.
10. fp=fopen("Sample.txt","r");
11. printf("The file contains\n");
12. fgets(str,12,fp);
13. puts(str);
14. fclose(fp);
15.}
```

*Output Program 14.7(a):*

Enter a string: C Programming Learn to Code

The file contains

C Programming Learn to Code

*Output Program 14.7(b):*

The file contains

Learn to Co

*Explanation Program 14.7(a):*

Line 8: Read a string from the console using `gets()` function.

Line 9: Write the string stored in `str` to the file using `fputs()` function.

Line 14: Read 30 characters from the file using `fgets()` function and store it in `str`.

Line 15: Display the content of `str` using `puts()` function.

Program 14.7(b) is self-explanatory.

We have seen different types of programs for reading/writing numbers, characters, and strings in a file. In the following section, we will write some programs using file operations, which are self-explanatory. We encourage students to write this code and run it to see the output.

---

## 14.6 Other Programming Examples

Write a program to create a file "Source.txt", input some character to it, copy the content to another file named "Destination.txt", and print the content of "Destination .txt".

### PROGRAM 14.8

```
#include<stdio.h>
void main()
{
 char ch;
 FILE *fs,*fd;
 fs=fopen("Source.txt","w");
 printf("Enter a characters and press ctrl+z to stop");
 while(1)
 {
 ch=getchar();
 if(ch==EOF)
 break;
 fputc(ch,fs);
 }
 fclose(fs);
 fs=fopen("Source.txt","r");
 fd=fopen("Destination.txt","w");
```

```

while (1)
{
 ch = fgetc (fs) ;
 if (ch == EOF)
 break ;
 fputc(ch,fd);
}
fclose(fs);
fclose(fd);
fd=fopen("Destination.txt","r");
printf("\nYour new file contains:\n");
while (1)
{
 ch = fgetc (fd) ;
 if (ch == EOF)
 break ;
 printf("%c",ch);
}
fclose(fd);
}

```

Write a program to input a name, roll number, and mark of a student into a file then display the content of the file.

### PROGRAM 14.9

```

#include<stdio.h>
void main()
{
 FILE *fp;
 int roll;
 char name[30];
 float marks;
 fp=fopen("Student.txt","w");
 printf("Enter the nName, rRoll number, and mark of the
student");
 scanf("%s%d%f",name, &roll, &marks);
 fprintf(fp,"%s %d %f",name,roll,marks);
 fclose(fp);
 fp=fopen("Student.txt","r");
 printf("\nthe file contain\n");
}

```

```
fscanf(fp,"%s%d%f",name, &roll, &marks);
printf("%s %d %f",name,roll,marks);
fclose(fp);
}
```

Write a program to pass a file to another function and display the content of the file using a function.

### PROGRAM 14.10

```
#include<stdio.h>
void filefun(FILE *cp)
{
 char ch;
 cp=fopen("Charfile.txt","r");
 printf("\nYour file contains\n");
 while(1)
 {
 ch=fgetc(cp);
 if(ch==EOF)
 break;
 printf("%c",ch);
 }
 fclose(cp);
}
void main()
{
 char ch;
 FILE *fp;
 fp=fopen("Charfile.txt","w");
 printf("Enter a characters and press ctrl+z to stop");
 while(1)
 {
 ch=getchar();
 if(ch==EOF)
 break;
 fputc(ch,fp);
 }
 fclose(fp);
 filefun(fp);
}
```

---

## 14.7 Review Questions

1. What will be the output of this C program?

```
#include<stdio.h>
int main()
{
 int EOF = 0;
 printf("%d", EOF);
 return 0;
}
```

2. What will be the output of this C program?

```
#include<stdio.h>
int main()
{
 printf("%d", EOF);
 return 0;
}
```

3. What will be the output of this C program?

```
#include<stdio.h>
int main()
{
 EOF++;
 printf("%d", EOF);
 return 0;
}
```

4. What will be the output of this C program?

```
int main()
{
 unsigned char chr;
 FILE *fp;
 fp = fopen("data1.txt", "r");
 while ((chr = fgetc(fp)) != EOF)
 {
 printf("%c", ch);
 }
 printf(" C Code");
 fclose(fp);
 return 0;
}
```

5. What will be the output of this C program?

```
#include<stdio.h>
int main()
```

```
{
 char ch;
 FILE *fptr1, *fptr2;
 fp = fopen("data1.txt", "w");
 fp = fopen("data2.txt", "w");
 printf("File Handling Practice");
 fclose(*fptr1, *fptr2);
 return 0;
}
```

6. Define a file and its uses in C?
7. Write a program in C to read the names and marks of N number of grocery items and store them in a file.
8. Write a program in C to read an existing file.
9. Write a program in C to write multiple lines in a text file.
10. Write a program in C to create and store information in a text file.
11. Write a program in C to read a file and store the lines into an array.
12. Write C code to keep records and perform analysis for a class of 20 students. The information on each student contains a roll number, name, sex, test scores (two tests per semester), mid-term score, final score, and total score. The program will prompt the user to choose an operation on the records from a menu as shown below:
  - (a) Add student records;
  - (b) Display all student records;
  - (c) Display an average of a selected student's scores;
  - (d) Display the highest and lowest scores;
  - (e) Search students' records by roll number.
13. Write a program in C to find the number of lines in a text file.
14. Write a program in C to delete a specific line from a file.
15. Write a program in C to replace a specific line with other text in a file.
16. Write a program in C to count the number of words and characters in a file.
17. Write a program in C to append multiple lines at the end of a text file.
18. Write C code to read the name and marks of N number of students from the user and store them in a file. If the file exists, add the information to the file or create it.
19. Write a C program to write all the members of an array of structures to a file using `fwrite()`.
20. Read an array from a file and display it on the screen.





**Taylor & Francis**

Taylor & Francis Group

<http://taylorandfrancis.com>

# 15

## *The Preprocessor*

---

### 15.1 Introduction

As the name suggests, this is something that needs to be processed before the actual processing is carried out. In the C language, preprocessing is nothing other than executing some special statements before actual compilation. These special statements are included inside a directive called a preprocessor directive. I want to show you where exactly the preprocessor directives lie if the general structure of a C program is taken into consideration (see Figure 15.1). Before the actual compilation process, the preprocessor does its work. There are several commands available to control the activity of a preprocessor. Preprocessors generally remove the comment lines from the program and replace the macro-definitions wherever necessary. After writing the code, the preprocessor does the preprocessing, and the file name with a .i extension will be generated before the actual compilation process.

The C language provides several preprocessor directives. In this chapter, we will discuss the frequently used preprocessor directives. After completing this chapter, the student will:

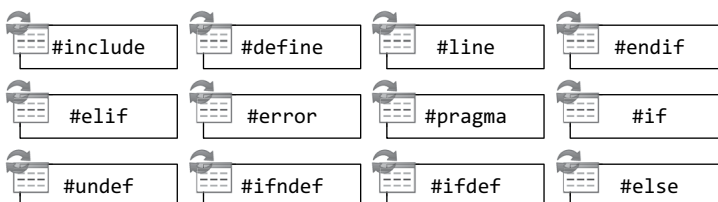
- Be able to know what a preprocessor is and why it is required;
- Be able to declare, define, and learn to code using preprocessor directives;
- Learn about the macro and the way it is used inside a program.

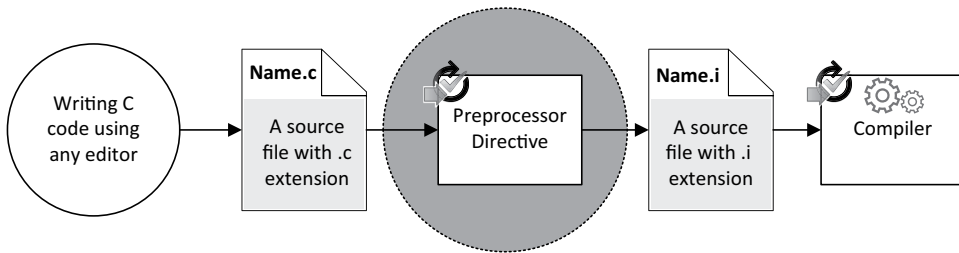
---

### 15.2 Preprocessor Directives

Everything that starts with a # symbol is known as a preprocessor directive, such as #include. Several other preprocessor directives are also present in C, and the objective of this chapter is to introduce those.

Here is a list of preprocessor directives:





**FIGURE 15.1**  
The position of a preprocessor directive.

The preprocessor statements do not end with a semicolon; rather, they extend over a whole line. That means you cannot write two preprocessors in a single line. The following statement shows an error:



Like other variables and functions, the preprocessor directive must appear before it is used inside the program. Generally, preprocessor directives are written at the beginning of the program, but it is not mandatory. Most of the preprocessor directives mentioned in the list are not used frequently. We will discuss the frequently used directives only. We categorize the above directives into four groups:

1. *Macro*: This is used to assign a symbolic name to a constant. It can also be used for defining macro-definitions. We use a `#define` preprocessor to write a macro.
2. *Include file*: We use a `#include` preprocessor to do this task, which includes the content of a file following the preprocessor directive. We have already used this in all our programs when using the statement `#include<stdio.h>`.
3. *Conditional preprocessors*: Several conditional preprocessor directives are available, such as `#if`, `#else`, `#endif`, `#elif`, `#ifndef`, and `#ifdef`. They are used for assigning conditions, whether to execute a line of code or not.
4. *Other directives*: This includes preprocessors like `#line`, `#pragma`, and `#error`. The overall definition of these directives is shown in Table 15.1.

---

### 15.3 Macro-substitutions

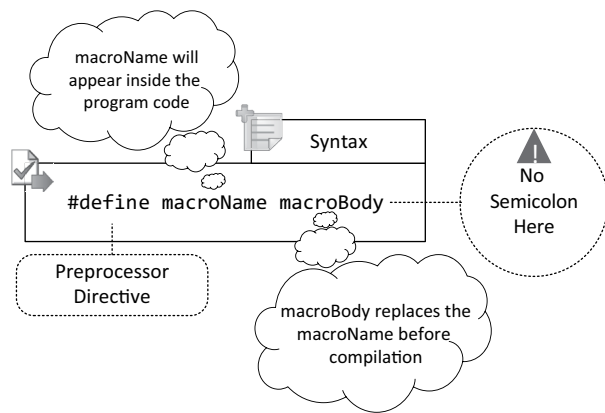
As per the discussion in Section 15.2, we use the `#define` preprocessor directive to define a macro. First, let us understand what a macro is? A macro, as the name suggests, is one line of code used for substituting a macro-name with a macro-body. In other words, we can say it is a process of assigning a symbolic name to a program constant.

**TABLE 15.1**  
List of Preprocessor Directives and Their Actions

| Serial No. | Name     | Action                                                                   |
|------------|----------|--------------------------------------------------------------------------|
| 1          | #define  | Defines a macro-substitution                                             |
| 2          | #undef   | Undefines a macro-substitution                                           |
| 3          | #include | Includes the source code of a file                                       |
| 4          | #if      | Used to test a constant expression                                       |
| 5          | #else    | Alternative source text when #if fails                                   |
| 6          | #endif   | Ends conditional statement                                               |
| 7          | #elif    | Alternative source text when #if fails                                   |
| 8          | #ifndef  | Checks for the existence of a macro-definition                           |
| 9          | #ifdef   | If a macro is defined, then includes the source text                     |
| 10         | #line    | Numbers the source text                                                  |
| 11         | #pragma  | An implementation-oriented directive which provides various instructions |
| 12         | #error   | Produces an error during debugging                                       |

A macro, as the name suggests, is one line of code used for substituting a macro-name with a macro-body.

Assume that you are using a constant value at several places in a big program. Later, due to a change in the problem statement, your solution demands a change to the constant value. Now, it isn't very easy to change that value that appears in several places. We need time and effort to update the change. This kind of problem is quite easy to solve using a macro. Let us take a program to explain it in more detail. Before proceeding to the program, let us learn how to write a macro. The syntax for writing a macro takes the following form (Figure 15.2).



**FIGURE 15.2**  
Syntax for writing a macro.

Consider the program shown in Figure 15.3. We declare a macro, `#define size 5`, where `size` represents the macroName, and `5` represents the macroBody. The program is quite simple; it declares an array, stores some numbers in it by reading the numbers from the console, and finally display those numbers. If you observe the program, the variable `size` is declared at four locations in the program. Due to the concept of macro-substitution discussed above, all these four locations will be replaced by the value `5`. Here you can see another important fact: that we have declared an array with a variable size (`int A[size];`), which is generally not allowed. As per the discussion in Chapter 9, we must provide an integer to represent the array size, though here we are using a variable, and the compiler does not complain because the variable `size` will be replaced with `5` before compilation. That is the beauty of macro-substitution.

### 15.3.1 Writing Macros with Arguments

It is possible to write a macro-substitution by providing arguments. The syntax is slightly different from the previous one. We need to provide the arguments within a bracket followed by the macroName. Wherever the macroName with an argument list appears inside the program, it is replaced with the macroBody. Figure 15.4 shows the syntax of declaring a macro-substitution with arguments.

Consider the following program shown in Figure 15.5. Here, we show the simplest macro to find the square of a number. Here `Square(a)` represents the macroName, and `a*a` represents the macroBody. As per the rule of macro-substitution, macroBody will be replaced with macroName throughout the program. In the program, macroName appears once and is replaced with `x*x`. The corresponding output is shown.

Let us consider some more programs using macros. The first program swaps two numbers (Figure 15.6) and the second program finds the bigger of two numbers (Figure 15.7).

### 15.3.2 Removing a Macro

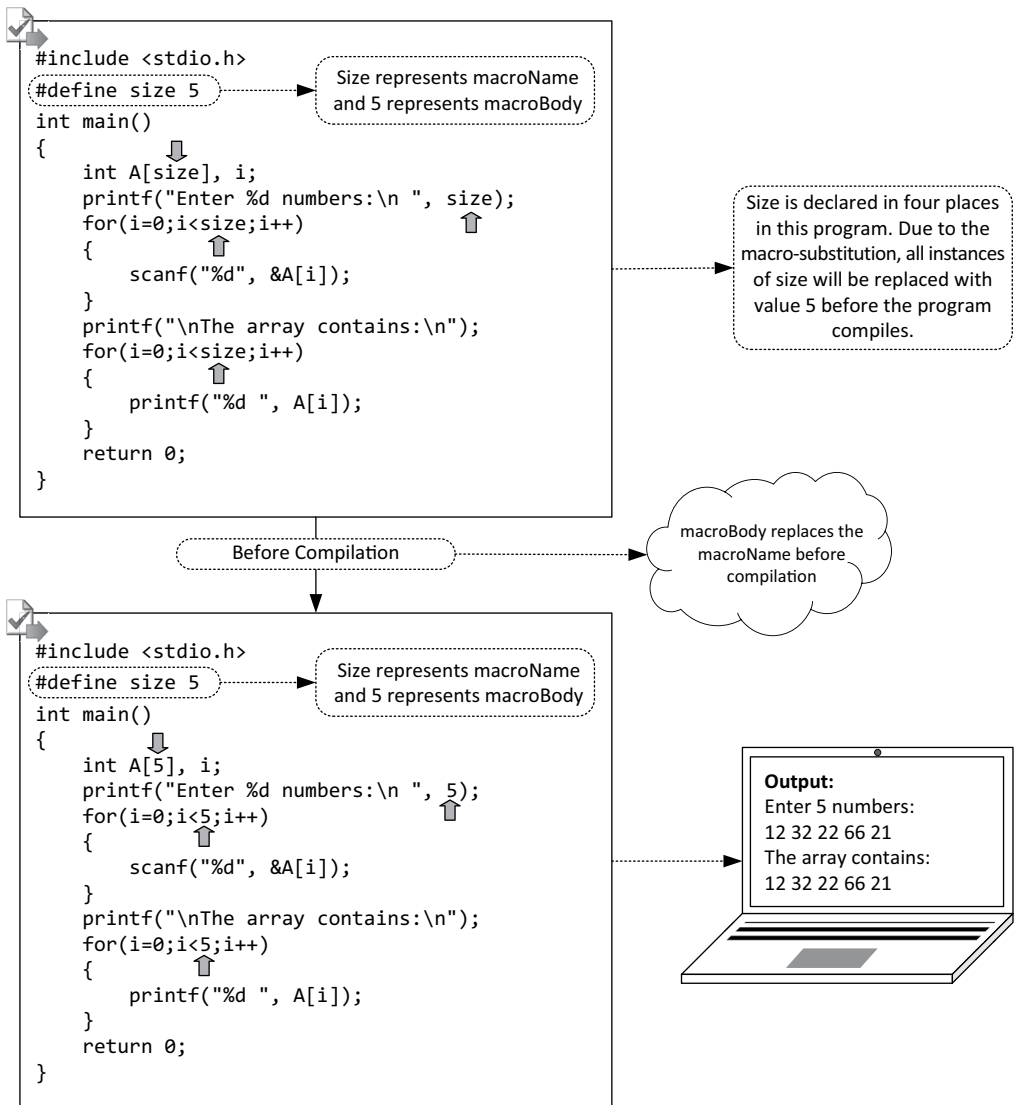
To remove a macro, we use the `#undef` preprocessor. The process is simple; we need to specify the macro name following the `#undef` preprocessor. The general syntax and an example are shown in Figure 15.8.

---

## 15.4 The #include Preprocessor

If you want to include a file that contains several functions and macros in your program, then you can do it with `#include` preprocessor directives. There are two ways to specify the file name that you want to include in your program. The general form of these two kinds is shown in Figure 15.9. The file name can either be specified within angle brackets or within double quotes, as shown in Figure 15.9a and 15.9b, respectively. After specifying the file name, the preprocessor inserts the code present inside the file into the current program.

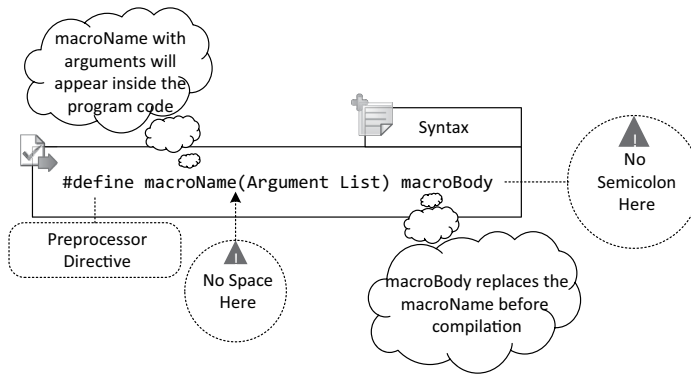
We have already used this preprocessor directive concept in all our programs by writing `#include<stdio.h>` as the first line of every program, where `stdio.h` is the file name; it contains the definition of several library functions used in our programs.



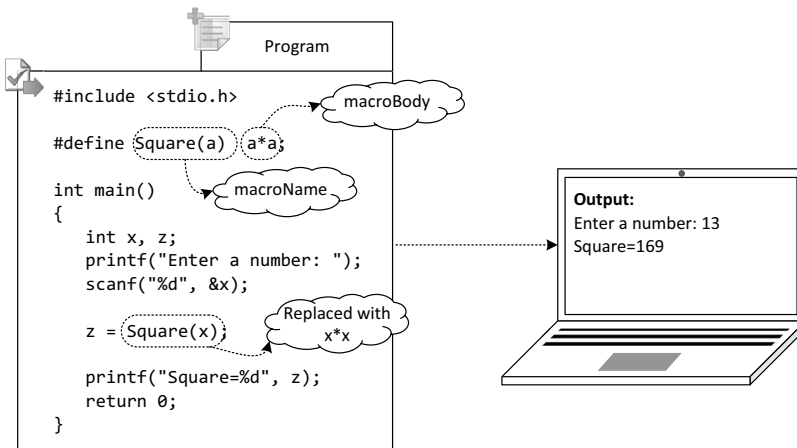
**FIGURE 15.3**  
Program showing the concept of macro-substitution.

Following this, we can create our own header file with several user-defined functions. To use the header file in our program, we can use the #include preprocessor. We must save the newly created header file (user-define header file) in the current directory where the program will be stored.

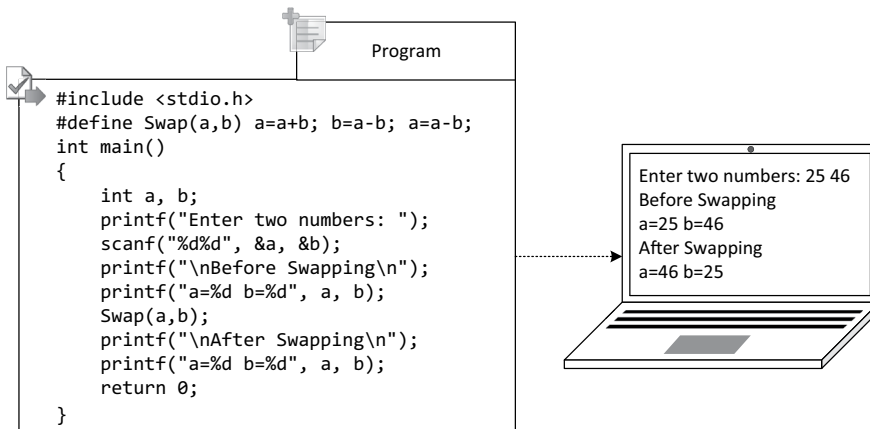
The angle bracket tells the preprocessor to search for the header file somewhere other than in the current directory. The double quotes tells the preprocessor to search in the current directory. In general, the header files are stored inside a subdirectory called include.



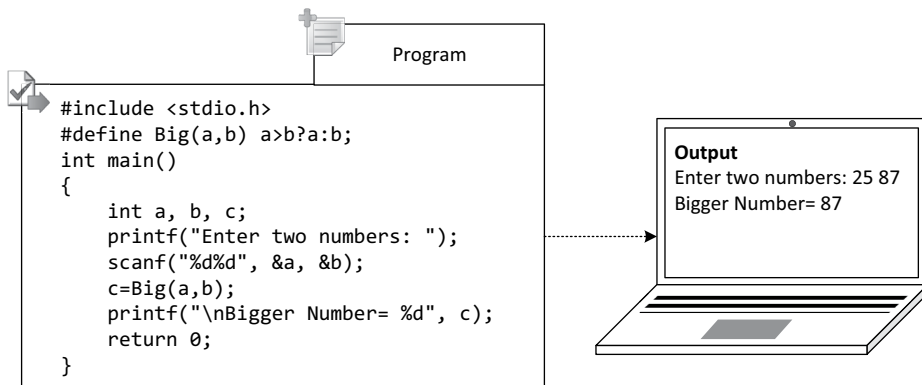
**FIGURE 15.4**  
Syntax of a macro-substitution with arguments.



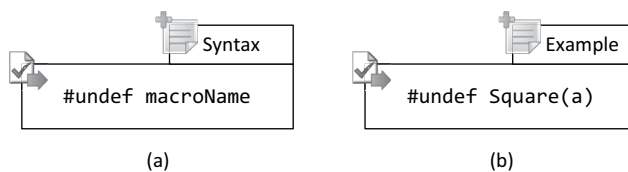
**FIGURE 15.5**  
Program showing a macro with arguments.



**FIGURE 15.6**  
Macro for swapping two numbers.



**FIGURE 15.7**  
Macro to find the bigger of two numbers.



**FIGURE 15.8**  
Syntax and example of the undef preprocessor.

## 15.5 Conditional Preprocessors

We use conditional preprocessors to compile a certain portion of the entire program which is referred to as conditional compilation. This is useful when we want to test some portion of the program. We can tell the compiler to skip some parts when certain conditions are met.

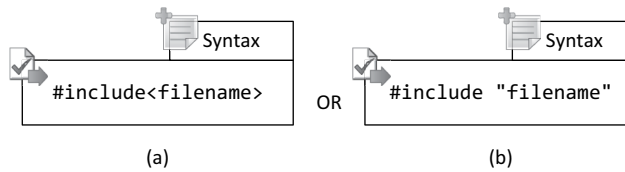
### 15.5.1 The #ifdef and #endif Preprocessor Directives

These two preprocessor directives help us to write some program statements, and the execution of those program statements depends on the presence of certain macro-definitions. The general syntax is shown in Figure 15.10.

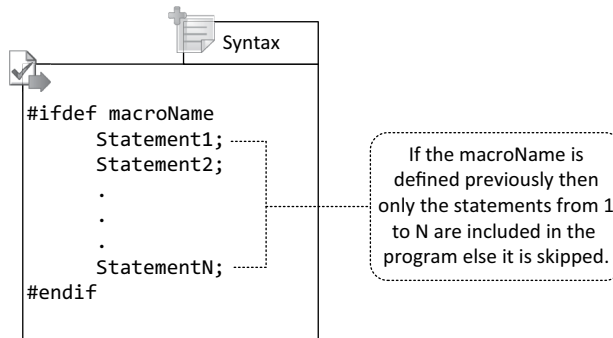
Consider the program shown in Figure 15.11. We focus on line 6 (Figure 15.11a), and whether to include it or not. That's why we put that statement inside #ifdef and #endif. The macro-name Sample has already been defined above the main() function. Hence, line 6 gets executed, and the output is shown in the figure.

Now, consider Figure 15.11b, where we define the macro Sample in a comment line. Hence, the program does not show any output.





**FIGURE 15.9**  
Syntax of the #include preprocessor.



**FIGURE 15.10**  
Syntax of #ifdef and #endif.

### 15.5.2 The #ifndef and #endif Directives

#ifndef means "if not defined" and it works in the opposite way to #ifdef. In this case, if the macro-name is not defined, then the statement is included in the program, otherwise it is not included. The syntax is exactly the same; only replace #ifdef with #ifndef. The syntax and one example is shown in Figure 15.12 and is self-explanatory.

### 15.5.3 The #if and #endif Directives

The statements included between these two directives will be included in the program only if an expression is evaluated as true. The general syntax of using these directives is shown in Figure 15.13.

If we want to choose some alternative statement when #if fails, then we can use the #else directive. The modified general form is shown in Figure 15.14.

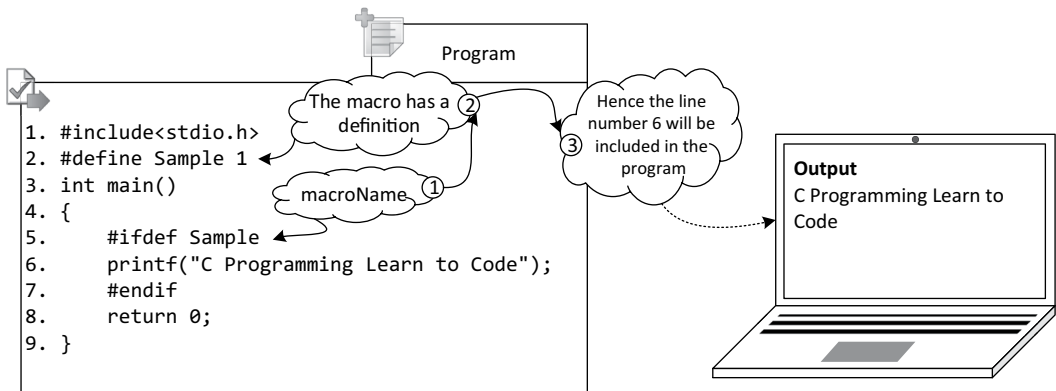
An alternative to this form exists that includes the #elif directive. The general form is shown in Figure 15.15.

---

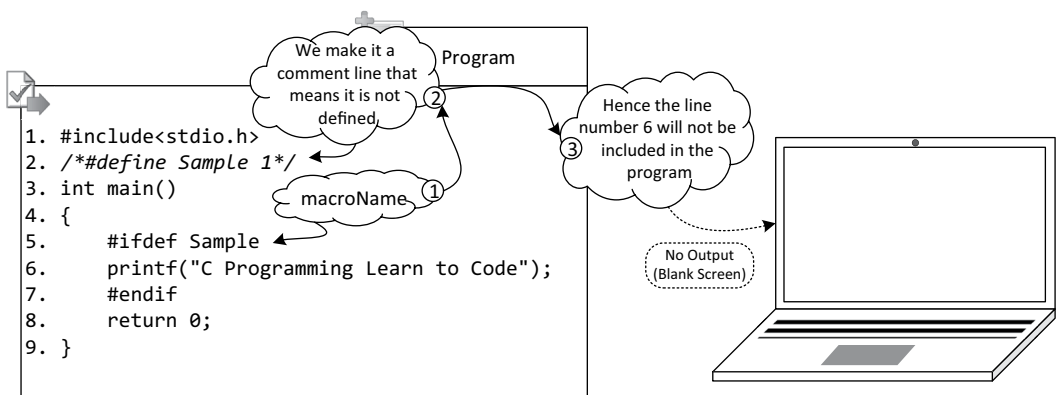
## 15.6 Other Preprocessor Directives

The other preprocessor directives include the following:

1. #line
2. #error
3. #pragma



(a)

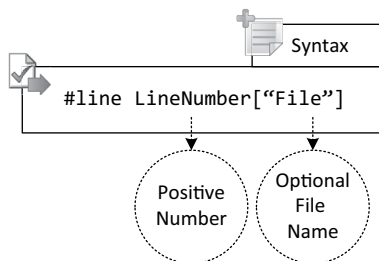


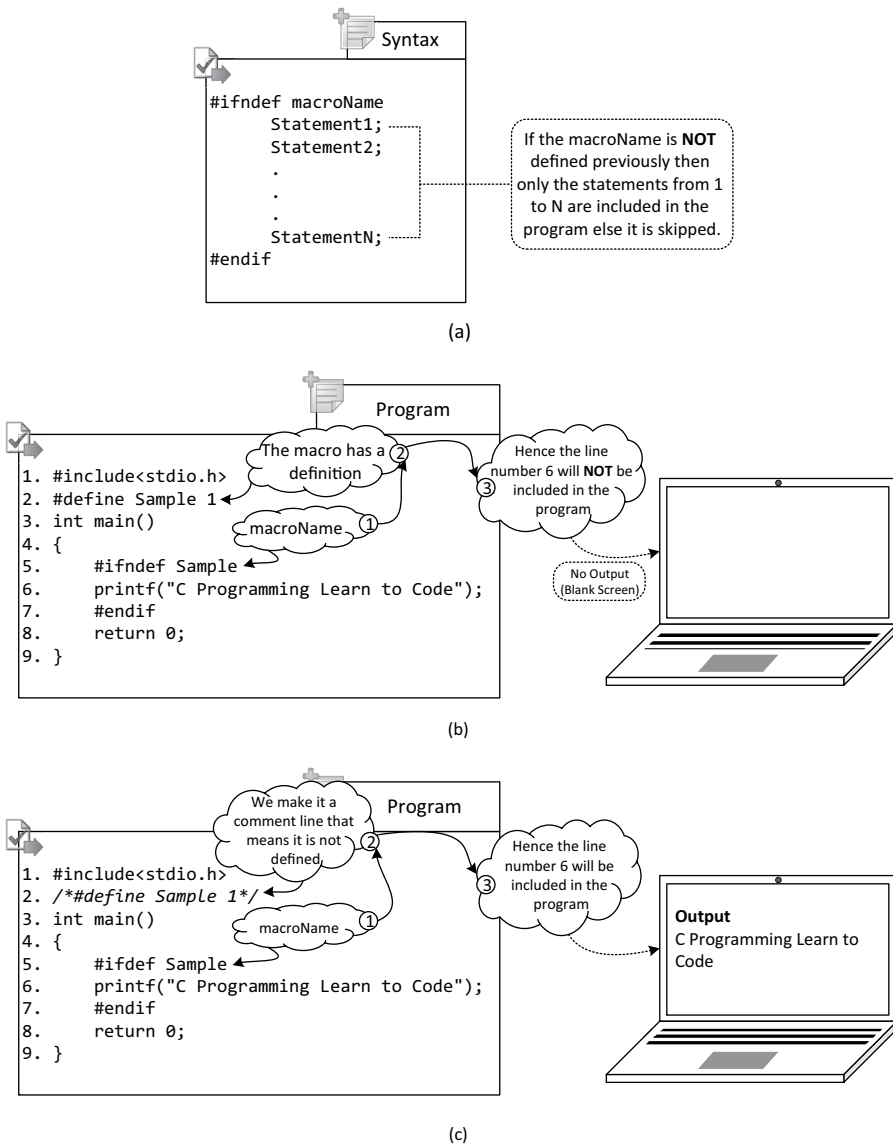
(b)

**FIGURE 15.11** Programming Example showing the use of #ifdef and #endif.

### 15.6.1 #line Directives

The #line directive is used to change or update the content of the predefined macro `_LINE_`. `_LINE_` contains the line number of the currently compiled line. The general form of this directive is as shown below:



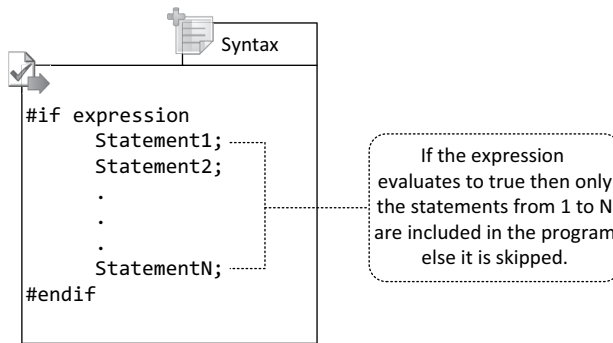


**FIGURE 15.12** Syntax and example of #ifndef directive.

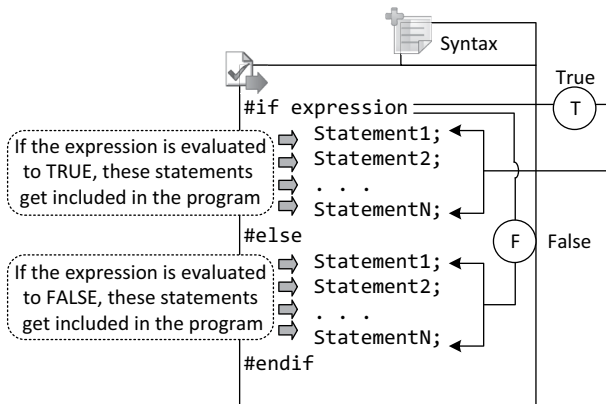
### 15.6.2 #error Directives

This forces the compiler to stop compilation and is used for debugging. The general syntax of declaring the #error directive is shown below:

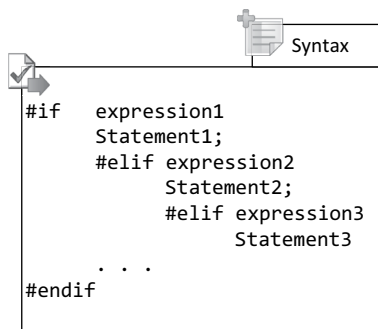




**FIGURE 15.13**  
Syntax of the #if directive.



**FIGURE 15.14**  
Syntax of #if including the #else directive.



**FIGURE 15.15**  
Syntax of #if including the #elif directive.

When the compiler encounters the #error statement, it stops compilation and displays an error of the following form:

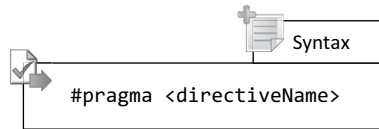
```

Fatal: File Name Line Error Directive: ErrorMessage
The ErrorMessage is the message specified in the syntax.

```

### 15.6.3 #pragma Directives

As mentioned earlier, #pragma is an implementation-dependent directive. Its usage varies from compiler to compiler. Here we are talking about a Turbo C compiler. We use #pragma to pass special messages to the Turbo C compiler. It takes the following general form:



We can pass a predefined directive supported by Turbo C as a message. If we pass anything other than the supported directive name, the compiler does not show any error, but it ignores it. There are two types of header files supported: inline and warn. The details of how they are used are left to the reader to discover.

## 15.7 Review Questions

1. What is the output of the following programs.

- (a) 

```
#include<stdio.h>
#define set(a) a+a
int main()
{
int b;
b=set(4)*set(5);
printf("%d", b);
return 0;
}
```
- (b) 

```
#include<stdio.h>
#define set(a) a+a*a
int main()
{
int b;
b=set(4)*set(5);
printf("%d", b);
return 0;
}
```
- (c) 

```
#include<stdio.h>
#define set(a) a++
int main()
{
int b=6;
b=set(b)+set(b);
```

```
 printf("%d", b);
 return 0;
}
(d) #include<stdio.h>
 int main()
 {
 #if 1
 printf("C Programming");
 #else
 printf("Learn to Code");
 #endif
 return 0;
 }
(e) #include<stdio.h>
 int main()
 {
 #if 0
 printf("C Programming");
 #else
 printf("Learn to Code");
 #endif
 return 0;
 }
(f) #include<stdio.h>
 #define a 0
 int main()
 {
 #if a+a
 printf("C Programming");
 #else
 printf("Learn to Code");
 #endif
 return 0;
 }
(g) #include<stdio.h>
 #define sum(a,b) a=a+b; b=a+b; b=a+b;
 int main()
 {
 int c=5, d=6;
 sum(c,d);
 printf("%d\n%d", c, d);
 return 0;
 }
```

```
(h) #include<stdio.h>
#define sum(a,b) a++; b++;
int main()
{
int c=5, d=6;
sum(c,d);
printf("%d\n%d", c, d);
return 0;
}

(i) #include<stdio.h>
#define sum(a,b) a++ + b--;
int main()
{
int c=5, d=6;
sum(c,d);
printf("%d\n%d", c, d);
return 0;
}
```

2. What is the difference between the compiler and the preprocessor and how are they related to each other?
3. What are preprocessor directives and why do we require a preprocessor directive?
4. How can we remove a macro-name from the program?
5. Describe the process of writing macro-substitutions with an appropriate example.
6. What is the difference between a macro and a function?
7. What are the advantages and disadvantages of using a macro in our program code?
8. What is the difference between `#ifdef` and `#ifndef` preprocessor directives. Explain with examples.
9. Describe the syntaxes of the `#if`, `#elif`, `#else`, and `#endif` preprocessor directives.
10. What is the use of the `#include` preprocessor directive and how can we use it in our program code?
11. Can we skip some lines of code and leave them unexecuted? If yes, then how?
12. Write a macro-substitution to find the biggest among three numbers.
13. Write macro-substitutions for addition, subtraction, multiplication, and division. Read two numbers and an operator (+, -, \*, or /) from the user. Perform the operation depending upon the operator provided by the user.
14. Write a macro-substitution to find the cube of a number.
15. Write the syntax of the `#line`, `#error`, and `#pragma` preprocessor directives and explain why they are used.

# 16

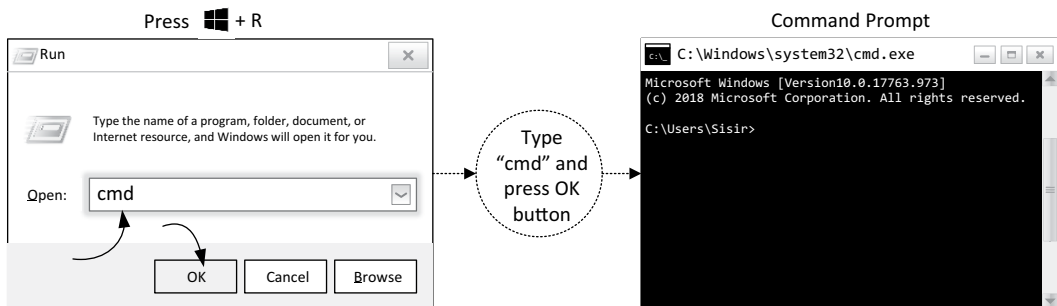
## Command Line Arguments

### 16.1 Introduction

As the name suggests, the command line argument means passing arguments (inputs) through the command line. Should we wonder what a command line is? It is the command prompt window we are talking about. The Windows user can find the command prompt window by pressing the **Win+R** button. A popup window will appear; type "cmd" and press the Enter key. You will see a black window on your screen which is called the command prompt. Figure 16.1 shows how to open a command prompt on the Windows operating system.

Now, why are we talking about this? Because we want to show you how to run a program using command prompts. In general, we are using a GUI (Graphical User Interface) editor to write, compile, and execute our C code. There are several GUI editors available for it. For instance, CodeBlocks, Turbo C++ Editor, Dev C++ Editor, Visual C++ Editor, and many more. What do these editors do? They provide us with an IDE (Integrated Development Environment), where we type our code; specialized buttons are given to compile and run our code with a mouse click only. We all use it because the process is easy to follow, and it fulfills our objectives. It also has another benefit: rather than thinking about how to run our code, we can concentrate more on program logic. But, as you are a programmer, you should know how exactly the editors compile and run your code. So, let us begin.

We need two different things to execute a C program code: a compiler and an editor. We know that a compiler is software that compiles our program and converts it into a form that is easy to understand by the machine. The machine executes the code and returns the output. But, where to write the code? We need an editor like notepad. Unfortunately, the



**FIGURE 16.1**  
How to open the command prompt in Windows.



notepad does not provide the facility to run your program. That's why the IDE developers produced several IDEs that combine the features of the editor and the compiler. We have mentioned several IDEs of this kind above. Let us discuss how the execution of a program is carried out using such IDEs.

### 16.1.1 The Code::Block IDE

Figure 16.2 shows a screenshot of the code block 17.12 IDE. You can see there are specialized buttons given for compile and run, and it also has an area for writing code, an error display area, and so on. The steps of writing and execution of the code are simple. First, we write the code in the code writing area and press the compile button using the mouse. The errors (if any) will be shown in the error showing area. If the program does not show any error, then the compilation process is successful. Now you can click on the run button to run your code. After you run your code, a separate output window (shown in Figure 16.2) will appear where you can provide the input (if your program demands it) and view the output.

Three files get created during the entire process of execution. The first file contains our code with a .c extension that we save after writing the code. The second file will be created after we compile the code, which is saved with a .obj (object file) extension. The third file is the .exe file that we can directly execute. You can view the files by going to the location where you saved your .c file.

It is quite easy to use an IDE, but when you install it you need to include two things simultaneously: the editor and the compiler. In general, the package you download from the internet contains both. Sometimes we can install them separately and link them after.

This chapter does not demand any discussion of IDEs, but for the sake of completeness, we mention it here. Our main concern is how to execute our program through the command prompt. To do this we don't require any specialized editor; a notepad will do. We only need a compiler that compiles the code. To pass the argument using the command

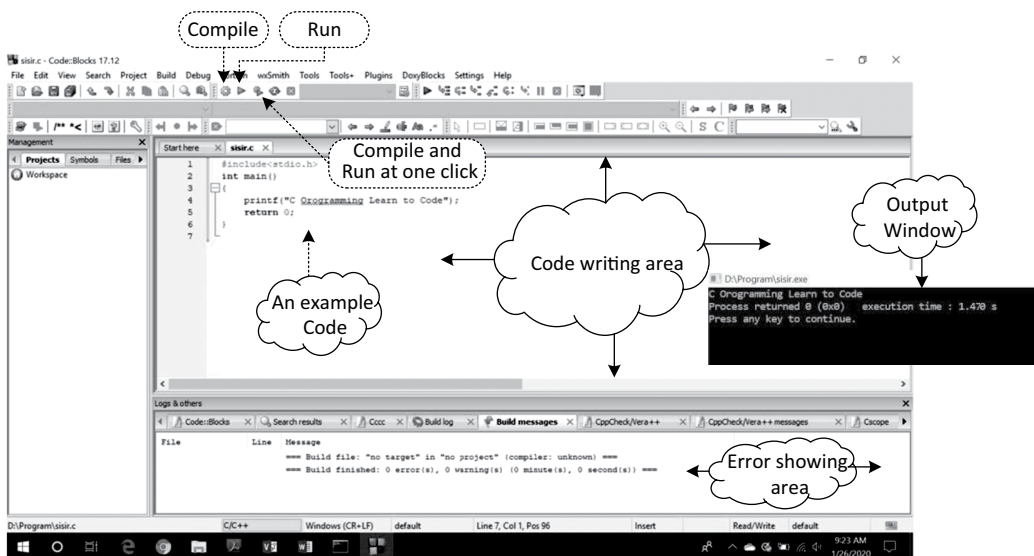


FIGURE 16.2  
Code block IDE.

line, it is mandatory to understand this process of program execution. After finishing this chapter, the student will know the following:

- How to execute programs without using any specialized GUI editors.
- What a command line argument is, and how to use them.
- How to pass parameters through the command line.
- Learn about `argc` and `argv`.

---

## 16.2 Execute a Program Using a Command Prompt

We will proceed step by step. First, we install a compiler, write a program using any editor, for instance, notepad, and finally execute the program using a command prompt. There are several compilers available to compile C code. Listed below are the most popular ones:

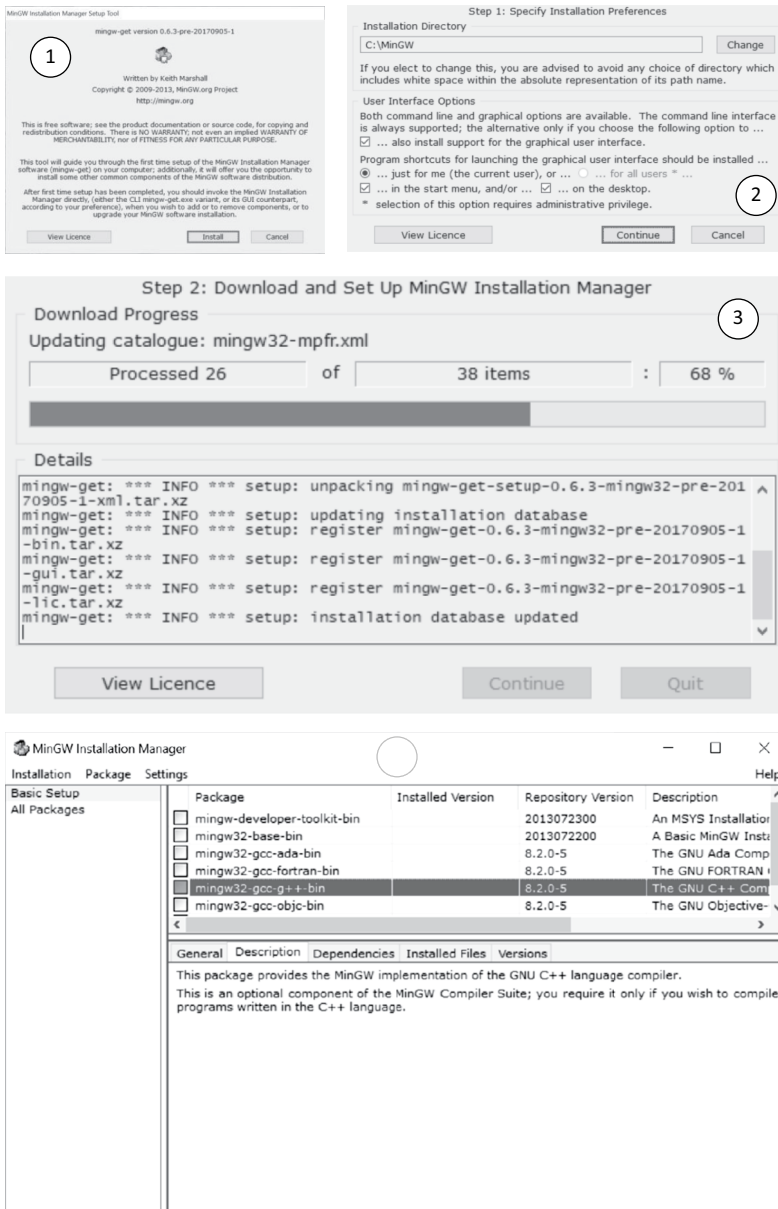
1. Turbo C Compiler: The most widely used compiler runs on the Windows operating system.
2. The gcc compiler: This is also one of the most popular compilers and runs on Linux/Unix systems. The Windows version of this compiler is also available with the name minGW and Cygwin.

For our purpose, we will show you the steps to install the minGW compiler on the Windows operating system.

### 16.2.1 Installing the minGW Compiler

The first step is to download the minGW setup file from the internet. The screenshots of each step of the installation procedure is shown in Figure 16.3. Follow the below steps to install it on your computer.

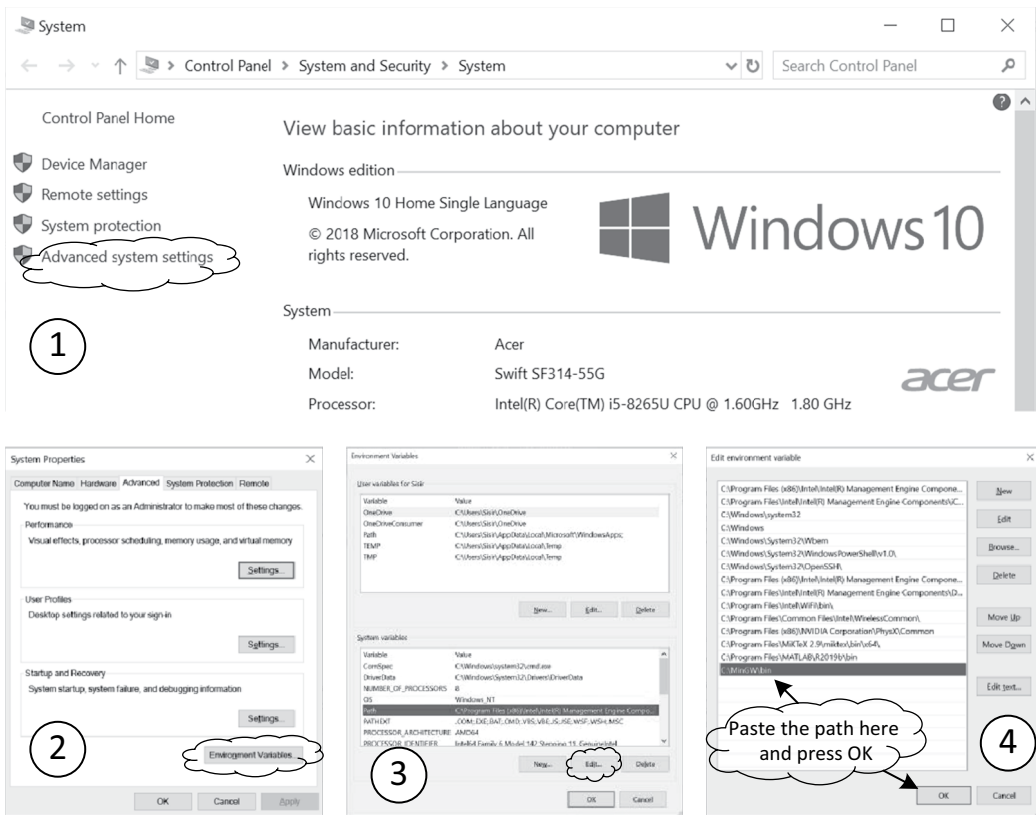
1. After downloading, double click on the mingw setup file and the ① first window will appear.
2. Click on the install button of ① the first window.
3. Choose your installation directory and the other options available to you in the next window marked as ②. Click on continue.
4. The ③ next window will appear that downloads the required files from the internet. An active internet connection is required during this process.
5. After the download completes, click on continue.
6. The next ④ window will appear: the minGW installation manager. On that window, mark the installation by choosing the appropriate compiler shown in the list.
7. For our case, we choose minGW-gcc-g++-bin. Click on the installation menu shown in the window ④ and click on the update catalog.
8. Finally, the installer will install the required package to your selected directory.



**FIGURE 16.3**  
minGW installation steps.

- After finishing the installation, click Alt+f4 to quit the installation manager. Now your system is installed with the mnGW compiler.

After the installation process is over, we need to set the Environment Variable, by specifying the path of the minGW installation folder. Setting the Environment variable is quite simple. Follow the steps given below to set it. Refer to Figure 16.4 for the following description.



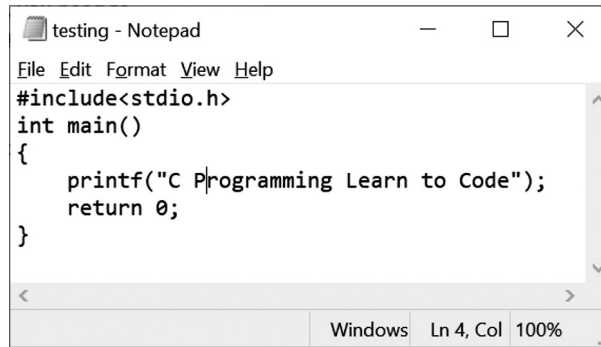
**FIGURE 16.4**  
Setting environment variables.

1. Right-click on *This PC* and click on *Properties*; a popup window will appear as shown in Figure 16.4 ①.
2. Click on *Advanced system settings* ⇒ A new popup window will appear named as ② *System properties* window ⇒ Click on *Environment Variable*.
3. The ③ *Environment Variable* popup will appear. In the *System Variables* section, select *path* and click on the edit button.
4. Another pop-up window will appear named as *Edit environment variable*. Paste the path “C:\MinGW\bin” and click on the OK button.

### 16.2.2 Compiling and Executing a Program

The process of executing a program is as follows:

1. Write a program using any editor. For our case, we use notepad. Save the program at any location of your choice. For example, I wrote a program with the name `testing.c` and saved it in a folder named `CProgram` inside the `D` drive. My complete file path will look like: `D:\Program\testing.c`. Let us write a program to print a line of text as shown below.



```

testing - Notepad
File Edit Format View Help
#include<stdio.h>
int main()
{
 printf("C Programming Learn to Code");
 return 0;
}
Windows Ln 4, Col 100%

```

- Open the command prompt, as shown in Figure 16.1. Go to the directory where your .c file is saved. Now the question is: how to go to that directive? We need to know some simple commands to do that.

Example:

Let the complete path of your program be D:\CProgram\testing.c

When you open the command prompt, you may see the following line (it will be different for different users):

```
C:\Users\Sisir>
```

But our program is present in D drive, and currently, we are in C drive. To change to D drive, just type D: and press enter (↵), and you will reach the D drive.

```
C:\Users\Sisir>D: ↵
```

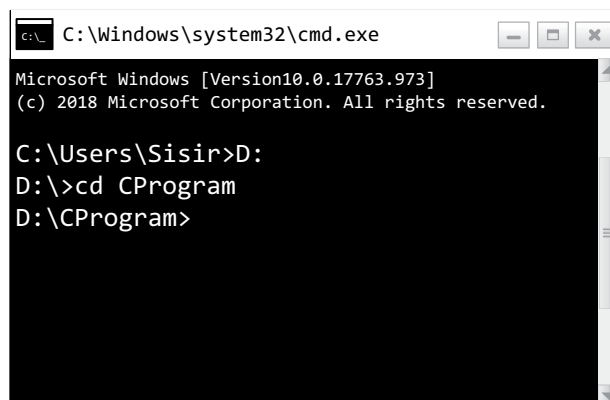
```
D:\>
```

Now you need the cd command to change through directories. cd stands for change directory. To reach your current folder (where your program is stored), you need to type cd foldername. In our example, our program is stored in CProgram folder. So, we will type the following and press enter (↵).

```
D:\>cd CProgram ↵
```

```
D:\CProgram>
```

That's all! Now you are in your directory where your testing.c file is stored. See the following command window for the complete line of command.



```

C:\Windows\system32\cmd.exe
Microsoft Windows [Version 10.0.17763.973]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Sisir>D:
D:\>cd CProgram
D:\CProgram>

```

- The next step is to compile the code. To do that we need to type the following command:

```
g++ filename.c ↓
```

where filename.c indicates the C program file that you want to compile.

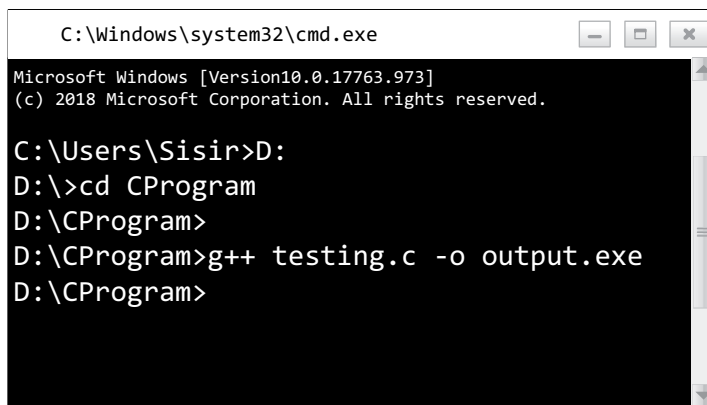
Example:

```
D:\CProgram>g++ testing.c ↓
```

After we compile it, the output file is generated with a default name a.exe. But, if you want to specify a name for your output file, then you can use the -o command. The complete example to write the command is shown below:

```
D:\CProgram>g++ testing.c -o output.exe ↓
```

- If your program does not contain any errors, then the .exe file will be generated; otherwise, the above command will show you the errors present in your program.



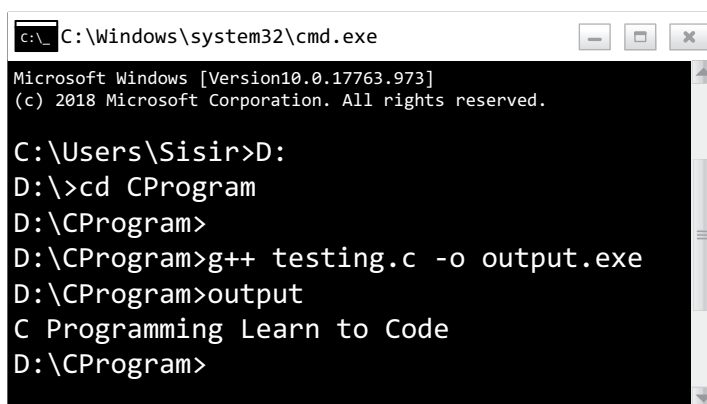
```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version10.0.17763.973]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Sisir>D:
D:\>cd CProgram
D:\CProgram>
D:\CProgram>g++ testing.c -o output.exe
D:\CProgram>
```

- Finally, to execute the file, we type the output file name and press enter as the following. See the command prompt window shown below.

```
D:\CProgram>output ↓
```

```
C Programming Learn to Code
```



```
c:\> C:\Windows\system32\cmd.exe
Microsoft Windows [Version10.0.17763.973]
(c) 2018 Microsoft Corporation. All rights reserved.

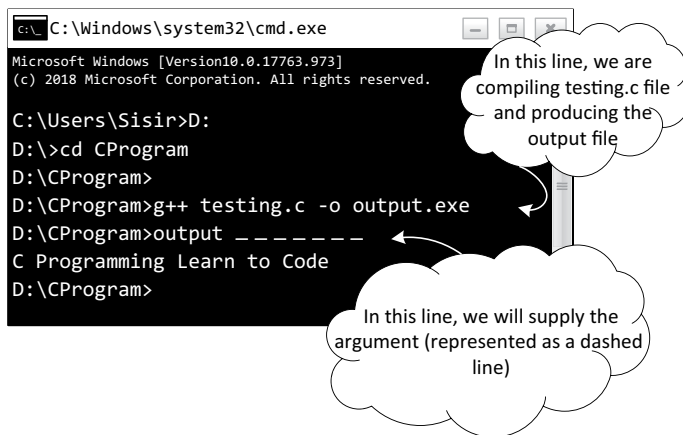
C:\Users\Sisir>D:
D:\>cd CProgram
D:\CProgram>
D:\CProgram>g++ testing.c -o output.exe
D:\CProgram>output
C Programming Learn to Code
D:\CProgram>
```

## 16.3 Fundamentals of the Command Line Argument

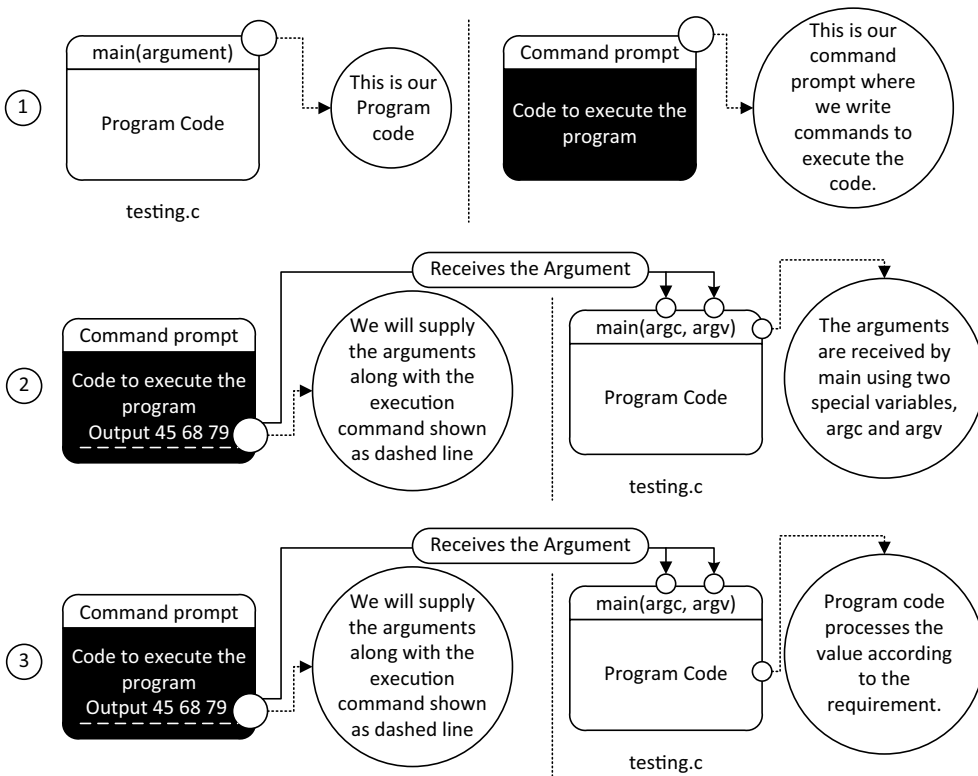
In Section 16.1.1, we saw the process of executing a program using an IDE: Code Block. As we need to click on different buttons (such as compile or run) to execute our program, we can say the process is quite user-friendly and straightforward. But using this technique of executing the program does not allow us to provide arguments through the command line. But the second method that we discussed in Section 16.2 allows us to give arguments through the command prompt. Why? Because we use commands to execute the program and hence while writing the command on a command prompt, we can supply the argument too.

Now the question is, where exactly is the argument given? Figure 16.5 shows the place where the arguments are supplied. As can be seen, when we want to execute the output file (output.exe), we can give the argument after typing the output file name (shown as the dashed line in the figure). The next thing we need to know is what should we supply as an argument, and after providing it, who will receive it, and finally, what will it do with it?

The argument is actually supplied to the `main()` function of our program. Until now, whatever program we have discussed, we have never supplied any argument inside the `main()` function. But, the `main()` function can take arguments. When we supply the argument at the command prompt, the `main()` function receives it and starts the processing. In the next section, we will discuss how the main function receives the argument. Figure 16.6 shows a step-by-step description of how to supply command line arguments to the `main()` function.



**FIGURE 16.5**  
Supplying arguments.



**FIGURE 16.6**  
Command line argument supply process.

## 16.4 Using Command Line Arguments

Assume that we were supplying some parameters through the command prompt, as discussed in the previous section. To receive these parameters, the main function must have some arguments, like other functions. According to the concept of the command line argument, the main function can take two arguments: (1) *argc* and (2) *argv*.

*argc*: This counts the total number of parameters passed through the command prompt. Each parameter is treated as a string.

*argv*: This is a pointer to an array of strings, and it points to the parameters passed in the command prompt.

Example:

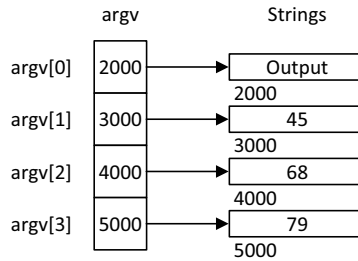
In the command prompt, let's say you write:

Output 45 68 79 ↓

where, Output is the Output.exe file that helps us in executing the program.

After receiving this parameter, the value of *argc* is set to 4, because there are four parameters passed in the command prompt, i.e., Output, 45, 68, and 79. All these parameters are treated as strings.

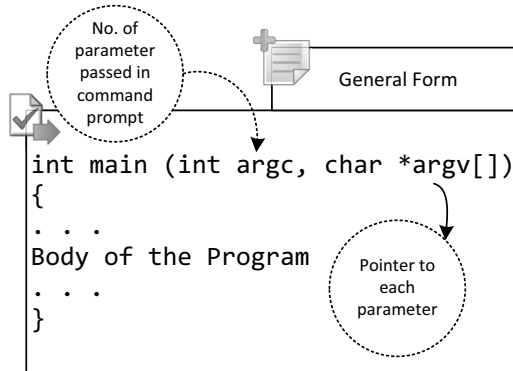




**FIGURE 16.7**  
Showing argv as an array of pointers to strings.

As mentioned above argv is an array of pointers to strings, and points to the parameters, as shown in Figure 16.7.

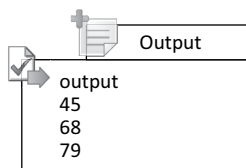
Now, we will not use blank parentheses after declaring the main() function. The main() function takes the following form.

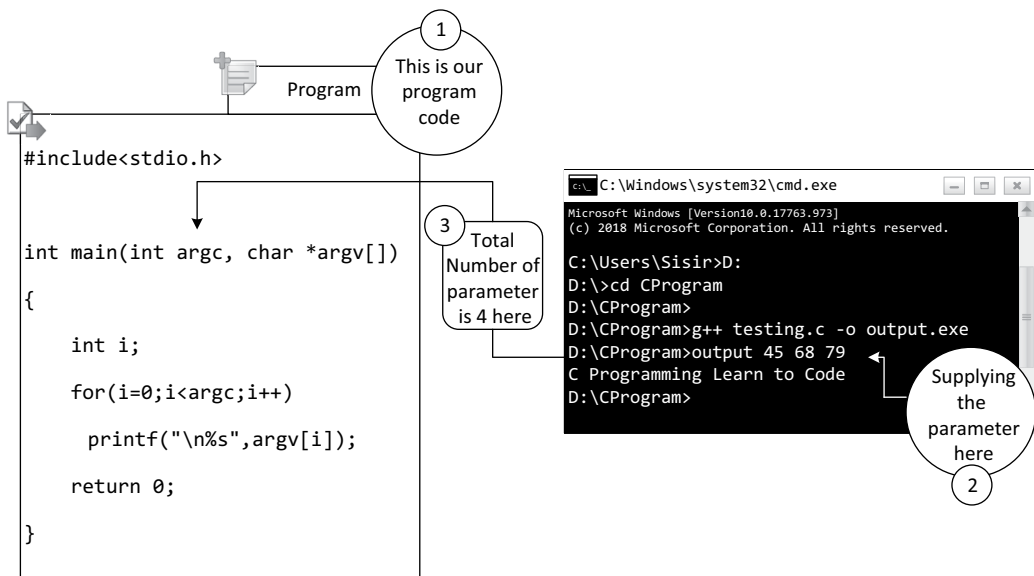


Let us write the body of the main() function that receives the value supplied in the command prompt and print those values.

Figure 16.8 shows a program that reads the argument supplied through the command prompt and prints the values. The argc argument receives the number of parameters supplied through the command prompt. The argument argv will point to all these parameters as shown in Figure 16.7.

Let us understand the for loop in our program. The for loop will execute 4 times here because argc has a value 4. Hence, the printf() statement will execute four times which prints the parameter to which argv points. The output of the above program will be:





**FIGURE 16.8**  
Execution of a simple command line argument program.

## 16.5 Review Questions

1. What will be the output of the program if it is executed from the command line?

```
cmd> Myprog Learn to Code
```

- a. 

```
int main(int argc, char *argv[])
{
 printf("%d %s", argc, argv[1]);
 return 0;
}
```
- b. 

```
int main(int argc, char **argv)
{
 printf("%c\n", **++argv);
 return 0;
}
```
- c. 

```
int main(int argc, char **argv)
{
 printf("%d\n", argc);
 return(0);
}
```
- d. 

```
int main (int argc, char*argv[])
{
 printf("%c", **++argv[1]);
}
```

2. If the following program (Helloprog) is run from the command line as CProg 11 12 13, what would be the output?

```
a. Int main (int argc, char*argv[])
{
 int i;
 i = argv[1] + argv [2] + argv [3];
 printf("%d", i);
}
```

```
b. void main (int argc, char*argv[]){int i,j=0;for (i=0; i < argc;
i++)j = j + atoi (argv[i]);printf ("%d",j);}
```

```
c. #include<stdio.h>
#include<stdlib.h>
int main(int argc, char *argv[])
{
 int num1 = atoi(argv[1]);
 printf("Number of arguments: %d\n", argc);
 printf("The entered number is %d", num1);
}
```

3. What is the command line used for? What does argv mean in a command line argument?
4. Write the correct form to declare main with command line arguments.
5. What is the maximum number of arguments that can be given in a command line in C?
6. What data type is argc?
7. What value does argc take when the program gets invoked?
8. What data type is argv?
9. What data type is argv[0]?
10. What gets stored in argv[0]?
11. Write C code with the input value 25 to check whether the value is even or odd.
12. Write C code to add two numbers by using a command line argument.
13. Write C Code to print all the arguments passed during program execution using the command line.
14. Write C Code to find the largest integers among three using a command line argument.
15. Write C code to compute the first  $n$  Fibonacci numbers using command line arguments.
16. Write C code to reverse a string entered from command line arguments.
17. Write C code to check whether a string is a palindrome using command line arguments.
18. Write C code for swapping numbers using command line arguments.

---

## *Appendix A: ASCII Character Table*

---

For a programmer, it is quite important to remember some ASCII character code as it helps in solving several programming problems. Here I present the ASCII character table for your reference.

| ASCII Value | Character | ASCII Value | Character | ASCII Value | Character | ASCII Value | Character |
|-------------|-----------|-------------|-----------|-------------|-----------|-------------|-----------|
| 0           | NUL       | 32          | (blank)   | 64          | @         | 96          | .         |
| 1           | SOH       | 33          | !         | 65          | A         | 97          | a         |
| 2           | STX       | 34          | "         | 66          | B         | 98          | b         |
| 3           | ETX       | 35          | #         | 67          | C         | 99          | c         |
| 4           | EOT       | 36          | \$        | 68          | D         | 100         | d         |
| 5           | ENQ       | 37          | %         | 69          | E         | 101         | e         |
| 6           | ACK       | 38          | &         | 70          | F         | 102         | f         |
| 7           | BEL       | 39          | `         | 71          | G         | 103         | g         |
| 8           | BS        | 40          | (         | 72          | H         | 104         | h         |
| 9           | HT        | 41          | )         | 73          | I         | 104         | i         |
| 10          | LF        | 42          | *         | 74          | J         | 106         | j         |
| 11          | VT        | 43          | +         | 75          | K         | 107         | k         |
| 12          | EF        | 44          | ,         | 76          | L         | 108         | l         |
| 13          | CR        | 45          | -         | 77          | M         | 109         | m         |
| 14          | SO        | 46          | .         | 78          | N         | 110         | n         |
| 15          | SI        | 47          | /         | 79          | O         | 111         | o         |
| 16          | DLE       | 48          | 0         | 80          | P         | 112         | p         |
| 17          | DC1       | 49          | 1         | 81          | Q         | 113         | q         |
| 18          | DC2       | 50          | 2         | 82          | R         | 114         | r         |
| 19          | DC3       | 51          | 3         | 83          | S         | 115         | s         |
| 20          | DC4       | 52          | 4         | 84          | T         | 116         | t         |
| 21          | NAK       | 53          | 5         | 84          | U         | 117         | u         |
| 22          | SYN       | 54          | 6         | 86          | V         | 118         | v         |
| 23          | ETB       | 55          | 7         | 87          | W         | 119         | w         |
| 24          | CAN       | 56          | 8         | 88          | X         | 120         | x         |
| 25          | EM        | 57          | 9         | 89          | Y         | 121         | y         |
| 26          | SUB       | 58          | :         | 90          | Z         | 122         | z         |

| ASCII Value | Character | ASCII Value | Character | ASCII Value | Character | ASCII Value | Character |
|-------------|-----------|-------------|-----------|-------------|-----------|-------------|-----------|
| 27          | ESC       | 59          | ;         | 91          | [         | 123         | {         |
| 28          | FS        | 60          | <         | 92          | \         | 124         |           |
| 29          | GS        | 61          | =         | 93          | ]         | 125         | }         |
| 30          | RS        | 62          | >         | 94          | ^         | 126         | ~         |
| 31          | US        | 63          | ?         | 95          | _         | 127         | DEL       |

---

## Appendix B: Integer Representation

---

### B.1 Introduction

This appendix introduces you to how to represent an integer (positive and negative) in a computer. After reading this appendix the student will be able to understand negative number representation as well as the addition and subtraction of numbers in two's complement representation.

If we want to store nonnegative integers only, then the representation is straightforward. Say I want to store +5 to +7 in an eight-bit word. We can simply convert it into its equivalent binary as shown below:

```
5 → 00000101
6 → 00000110
7 → 00000111
```

But, when it comes to negative number representation, we cannot convert it directly. We need some special technique to represent a negative number in a computer.

---

### B.2 Representation Type

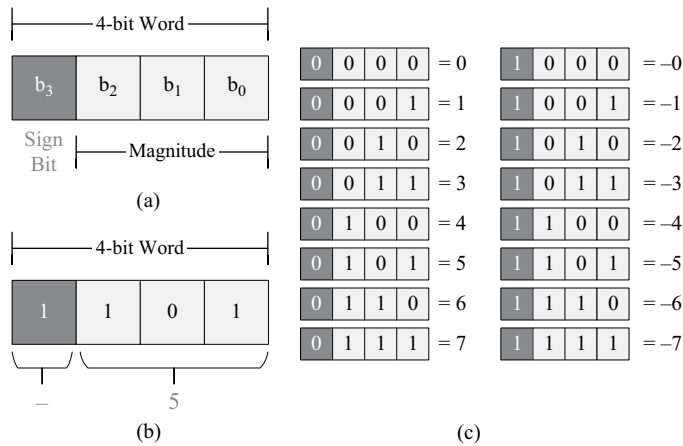
There are three ways to represent an integer:

1. Sign-magnitude representation;
2. One's complement representation;
3. Two's complement representation.

#### B.2.1 Sign-magnitude Representation

With this technique, the most significant bit (leftmost bit) acts as a sign bit. When the leftmost bit (sign bit) is "0", the number is positive, and for "1", the number is negative. The remaining bits of the word act as the magnitude. Figure B.1 shows the representation style.

Figure B.1a shows a four-bit word to represent a number. The leftmost bit  $b_3$  is reserved for the sign bit and the remaining bits i.e., from  $b_0$  to  $b_2$ , is used to represent the number. With three bits ( $b_0$  to  $b_2$ ) we can represent  $2^3 = 8$  different numbers from 000 (0 in decimal) through 111 (7 in decimal). Figure B.1b shows how  $-5$  is represented with this format.



**FIGURE B.1**  
Sign-magnitude representation.

Figure B.1c shows the complete range of numbers that can be represented with four bits using sign-magnitude representation. The complete range is  $-7$  to  $+7$  with two representations of 0's. In general, if an  $n$ -bit word is given, then the range will be from  $-(2^{n-1} - 1)$  to  $+(2^{n-1} - 1)$ .

### Q&A

1. With 16 bits, find the range of numbers that can be represented using sign-magnitude representation.

#### Solution

With  $n$  bits the range will be:  $-(2^{n-1} - 1)$  to  $+(2^{n-1} - 1)$ .

With 16 bits the range will be:  $-(2^{16-1} - 1)$  to  $+(2^{16-1} - 1) = -32767$  to  $+32767$ .

2. Represent the  $+75$  in sign-magnitude representation using 16 bits.

#### Solution

The binary equivalent of 75 is 1001011.

With 16 bits, the representation will look like 000000001001011 (adding nine 0's on the left side).

As it is a positive number, the 16th bit will be 0.

Hence with 16 bits the result will be 000000001001011.

3. Represent  $-39$  in sign-magnitude representation using 16 bits.

#### Solution

With 15 bits, the representation will look like 00000000100111 (adding nine 0's on the left side).

As it is a negative number, the 16th bit will be 1.

Hence with 16 bits the result will be 100000000100111.

### B.2.1.1 Demerits of Sign-magnitude Representation

In sign-magnitude form, 0 has two representations:

$$\begin{aligned}
 +0 &= 0000 \\
 -0 &= 1000
 \end{aligned}$$

which is not acceptable. Due to this, sign-magnitude representation is not used in a computer.

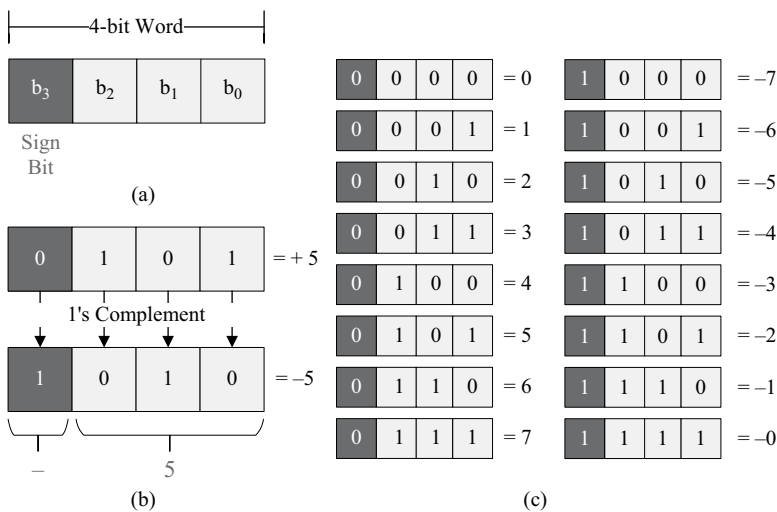
### B.2.2 One's Complement Representation

With this technique also, the most significant bit (leftmost bit) acts as a sign bit. When the leftmost bit (sign bit) is "0", the number is positive, and for "1", the number is negative. The remaining bits of the word is used to represent the actual number. Figure B.2 shows the representation style.

Figure B.2a shows a four-bit word to represent a number. The leftmost bit  $b_3$  is reserved for sign bit and the remaining bits i.e., from  $b_0$  to  $b_2$  are used to represent the number.

- To represent a positive number, we convert it to its equivalent binary number.
- To represent a negative number, we first convert the number to its binary form and then apply one's complement (flipping the bits from 1 to 0 and 0 to 1).

Figure B.2b shows the representation of  $-5$ . First, we find the binary equivalent of 5, which is 0101 in four bits. Then we flip the bits (1 to 0 and 0 to 1) to get the resultant bit for  $-5$ . Similarly, Figure B.2c shows the complete range of numbers that can be represented with four bits using one's complement representation. The complete range is  $-7$  to  $+7$  with two representations of 0's. In general, if an  $n$ -bit word is given, then the range will vary from  $-(2^{n-1} - 1)$  to  $+(2^{n-1} - 1)$ .



**FIGURE B.2** One's complement representation.



**Q&A**

1. With 16 bits, find the range of numbers that can be represented using one’s complement representation.

**Solution**

With n bits the range will be:  $-(2^{n-1} - 1)$  to  $+(2^{n-1} - 1)$ .

With 16 bits the range will be:  $-(2^{16-1} - 1)$  to  $+(2^{16-1} - 1) = -32767$  to  $+32767$ .

2. Represent +75 in one’s complement representation using 16 bits.

**Solution**

The binary equivalent of 75 is 1001011.

With 16 bits, the representation will look like 000000001001011 (adding nine 0’s on the left side).

As it is a positive number, the 16th bit will be 0.

Hence with 16 bits the result will be 000000001001011.

3. Represent -39 in one’s complement representation using 16 bits.

**Solution**

The binary equivalent of 39 is 100111.

With 16 bits, the representation will look like 00000000100111 (adding ten 0’s on the left side).

As it is a negative number, we will find its one’s complement by flipping the bits. Refer to Figure B.3.

Hence with 16 bits the result will be 111111111011000.

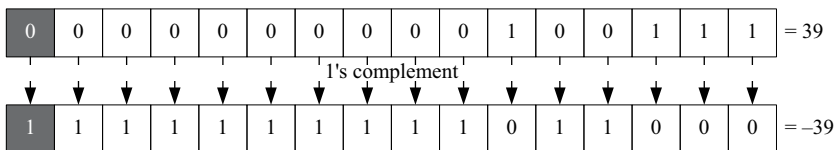
**B.2.2.1 Demerits of One’s Complement Representation**

In one’s complement form, 0 has two representations:

$$+0 = 0000$$

$$-0 = 1000$$

which is not acceptable. Due to this, one’s complement representation is also not used in a computer.



**FIGURE B.3**  
One’s complement of 39.

### B.2.3 Two's Complement Representation

With this technique the most significant bit (leftmost bit) acts as a sign bit. When the leftmost bit (sign bit) is "0", the number is positive, and for "1", the number is negative. The remaining bit of the word is used to represent the actual number. Figure B.4 shows the representation style.

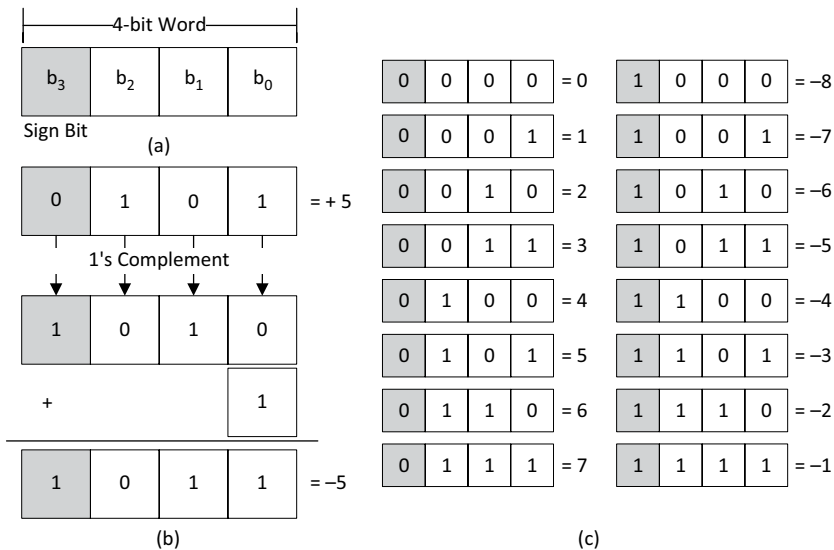
Figure B.4a shows a four-bit word to represent a number. The leftmost bit ( $b_3$ ) is reserved for the sign bit and the remaining bits i.e., from  $b_0$  to  $b_2$  are used to represent the number.

- To represent a positive number, we simply convert it to its equivalent binary number.
- To represent a negative number, we first convert the number to its binary form and then apply two's complement.
- To find two's complement; we add 1 to the one's complement of that number.

Figure B.4b shows the representation of  $-5$ . First, we find the binary equivalent of 5 which is 0101 in four bits. Then we flip the bits (1 to 0 and 0 to 1) to get the one's complement, i.e., 1010. Finally, we add 1 to the one's complement to get the resultant two's complement which is 1011. Hence,  $-5$  is represented as 1011 in two's complement representation. Figure B.4c shows the complete range of numbers that can be represented with four bits using two's complement representation. The complete range is  $-8$  to  $+7$  with a single representation of 0's. In general, if an  $n$ -bit word is given, the range will vary from  $-(2^{n-1})$  to  $+(2^{n-1} - 1)$ .

#### Q&A

1. With 16 bits, find the range of numbers that can be represented using two's complement representation.



**FIGURE B.4** Two's complement representation.

**Solution**

With  $n$  bits the range will be:  $-(2^{n-1})$  to  $+(2^{n-1} - 1)$ .

With 16 bits the range will be:  $-(2^{16-1})$  to  $+(2^{16-1} - 1) = -32768$  to  $+ 32767$ .

2. Represent +75 in two's complement representation using 16 bits.

**Solution**

The binary equivalent of 75 is 1001011.

With 16 bits, the representation will look like 000000001001011 (adding nine 0's on the left side).

As it is a positive number, the 16th bit will be 0.

Hence with 16 bits the result will be 000000001001011.

3. Represent -39 in one's complement representation using 16 bits.

**Solution:**

The binary equivalent of 39 is 100111.

With 16 bits, the representation will look like 00000000100111 (adding ten 0's on the left side).

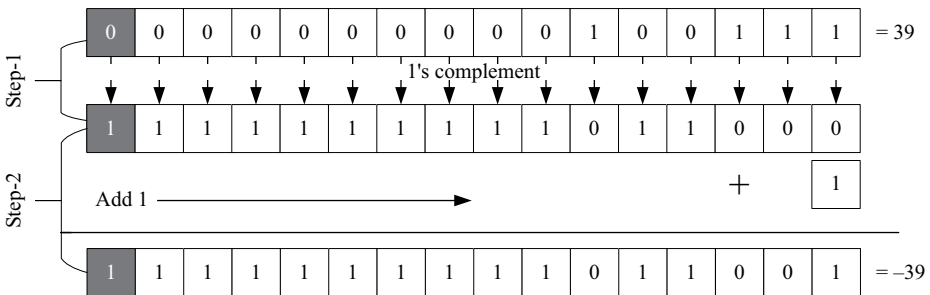
As it is a negative number, we need to find its two's complement. To find two's complement refer to the following steps:

Step 1: Find the ones complement of 00000000100111.

Step 2: Add 1 to the result obtained from step 1.

Refer to Figure B.5 which shows the above steps graphically.

Hence with 16 bits the result will be 111111111011001.



**FIGURE B.5**  
Two's complement of 39.

---

# Index

---

## A

\a alert escape sequence, 92  
actual argument, 187–188, 276–282  
+= addition assignment, 99  
addition of two numbers, 47, 90  
& address of operator, 89, 134, 260–265, 277–280, 293–294  
advantages of C Language, 61  
algorithm, 39–45  
    advantages, 44  
    basics, 43  
    characteristics, 42  
    examples, 44–45  
ALU, 5–6  
ampersand, 89  
AND operator, 99–101  
ANSI, 62–63, 65, 70  
argc, 411–412  
arguments, 182, 187–188, 191–194  
argv, 411–413  
-arithmetic minus operator, 96  
\* arithmetic multiplication operator, 96  
arithmetic operator, 96  
    division operator, 96–97  
    remainder operator, 96–97  
+ arithmetic plus operator, 96  
array, 117, 126, 213  
    accessing elements, 217  
    bound checking, 225  
    character array, 126, 214, 245  
    declaration, 215  
    initialization, 216  
    memory representation, 217  
    passing to function, 285  
    subscript, 217  
    three dimensional, 242–244  
    two dimensional, 225  
array of pointers, 295, 412  
array of structures, 316  
-> arrow operator, 112, 312, 323, 336  
assembly language, 54  
= assignment operator, 43, 85, 98–99, 117  
auto, 78, 204–206  
automatic variable, 204–206

## B

base, 17–18  
\b backspace escape sequence, 92  
BCPL, 62–63  
Bell Lab, 63  
binary operator, 95, 97, 109  
binary to decimal conversion, 20, 23–26  
binary to hexadecimal conversion, 34–35  
binary to octal conversion, 30–31  
bit, 9, 20–22  
bit-fields, 338–339, 342–343  
& bit-wise AND, 106  
~ bit-wise complement operator, 107  
bitwise operator, 96, 105–111  
    bit-wise AND, 106  
    bit-wise OR, 106  
    bit-wise XOR, 106  
    complement, 107  
    left shift, 109  
    right shift, 109  
| bit-wise OR, 106  
^ bit-wise XOR, 106  
B language, 63  
block diagram of a computer system, 4–5  
bound checking, 225  
branching, 141, 143, 173  
break, 78, 155–158, 173  
byte, 9, 80–82

## C

c99, 63, 65, 69–70, 78, 338  
called function, 186, 188, 276–277  
calling function, 186, 188, 193, 276–277  
calloc(), 350, 352–353, 356  
case, 78, 155, 157  
character constant, 83, 86–87, 155  
character sets, 77  
command line arguments, 403, 410–411  
    argc, 411–412  
    argv, 411–413  
comma operator, 112, 117, 169  
comment, 43, 65, 67  
    multi line comment, 65

- single line comment, 65, 67
- compiler, 56–57, 64, 67, 69–73, 78
- compound statements, 143–144, 146–147, 154
- computer, 1, 4–7
  - basic organization, 4
  - central processing unit (CPU), 6
  - control unit (CU), 6
  - input device, 5
  - output device, 7
  - storage device, 6
- ?: conditional operator, 95, 103–105
- conio.h, 66, 124
- console I/O, 123–124, 181, 365–366
- const, 78, 269–270
- constants, 86–89
- continue, 78, 173–174
- control statements, 43–44, 142–143
  - loop, 44, 142, 160–170
  - selection, 44, 142–158
  - sequence, 142
- control unit, 5–7
- conversion, 23–35
- CPU, 4–8

## D

- data segment, 351
- data types, 80–83
  - character, 83
  - double, 78, 80–81, 83, 89
  - float, 78, 80–83
  - integer, 80–82
  - long long, 82, 89
- decimal numbers, 17–20
- decision-making, 46, 141
- declaration, 84
  - array, 215, 226, 242, 245
  - bitfield, 340
  - enumeration, 343
  - function prototype, 187
  - pointer, 261, 272, 295, 298
  - structure, 307–308
  - union, 333
  - of variable, 84
- decrement operator, 95, 100–103, 117
- default, 78–79, 155
- Dennis Ritchie, 62–63
- %d format specifier for int, 89
- /= divide assignment, 99
- / division operator, 96
- dot operator, 112, 312, 319, 336
- double, 78, 80–81, 83, 89

- do-while loop, 164–167
- dynamic memory allocation, 349–361
  - calloc, 352, 356
  - free, 352, 360
  - malloc, 352–353
  - realloc, 352, 356

## E

- editor, 64–65, 69–72, 403–404
- else, 44, 78, 146–153
- enum, 78, 343–345
- enumeration, 80, 343–345
- EOF, 371
- == equality operator, 98
- escape sequences, 91–92
- executable file, 65, 71–72
- executing a C program, 64
  - compiling, 64
  - executable file, 65
  - linking, 65
  - object file, 65
- expressions, 86, 113
- extern, 78, 204–205, 207
- external variable, 207

## F

- factorial, 160, 170, 195–197
- \f form feed, 92
- fibonacci series, 203–204
- file handling functions, 369
  - fgetc(), 377–379
  - fgets(), 381–382
  - fprintf(), 371–373
  - fputc(), 377–379
  - fputs(), 381–382
  - fscanf(), 371–373
  - getw(), 374
  - putw(), 374
- FILE pointer, 369
- float, 78, 80–83
- flowchart, 45–53
  - advantages, 45
  - symbols, 46
- for loop, 167–169
- formal argument, 187–188
- format specifier, 88–89, 134
- formatted functions, 128–136
  - printf(), 128
  - scanf(), 134
- free, 352, 360

function declaration, 180, 295  
 function prototype, 187  
 functions, 179  
   arguments, 188  
   called function, 186  
   calling function, 186  
   library functions, 182  
   parameters, 188  
   prototype, 187  
   user-defined function, 182–183

**G**

gcc, 70–72, 82, 405  
 getch(), 124, 126–127  
 getchar(), 124  
 getche(), 124, 126–127  
 gets(), 124–125  
 getw(), 374  
 gigabyte, 9  
 global variable, 66, 207  
 goto, 78, 159  
 > greater than, 98  
 >= greater than equal to, 98  
 GUI, 403

**H**

hardware, 12  
 hash symbol (#), 389  
 header file, 65  
 heap, 350–351  
 hexadecimal number system, 21  
 hexadecimal to binary conversion, 33  
 hexadecimal to decimal conversion, 33  
 high-level language, 53, 55

**I**

IDE, 69, 403  
 identifier, 79  
 if, 143  
 if-else, 146  
 if-else-if ladder, 151  
 implicit type conversion, 115  
 include, 65, 392  
 increment operator, 100  
 index (subscript), 217  
 \* indirection operator, 262, 270  
 infinite loop, 162  
 input device, 5  
 input/output functions

  getch(), 124, 126–127  
   getchar(), 124  
   getche(), 124, 126–127  
   printf(), 128  
   scanf(), 134  
 int, 78, 81–82  
 integer constant, 86  
 interpreter, 56–57  
 ISO, 63  
 iterations, 51–52, 161, 166, 170

**K**

keywords, 78  
 kilobyte, 9

**L**

<< left-shift operator, 106, 109  
 < less than, 98  
 <= less than equal to, 98  
 linked list, 324  
 linker, 64–65  
 local variables, 350  
 && logical AND, 99–100  
 ! logical NOT, 99–100  
 logical operator, 95, 99  
   logical AND, 100–101  
   logical NOT, 100  
   logical OR, 100  
 || logical OR, 99–100  
 long, 78, 82, 89  
 long double, 83, 89, 340  
 long int, 82, 89, 340  
 long long, 82, 89  
 looping, 44, 142, 160–170  
 low-level language, 53  
 lvalue, 98–99, 271

**M**

machine-level language, 54  
 macro, 268, 390–395  
   arguments, 392  
   #define, 391, 393  
   functions, 394, 395  
   substitution, 391  
 magnitude, 417  
 main, 66  
 malloc(), 352–353  
 mantissa, 87  
 math.h, 316

megabyte, 9  
 memory, 1–2, 4, 6–7  
   primary, 7–8  
   secondary, 7, 10  
 minGW, 70, 72, 82, 405  
 modular programming, 181, 185  
 modules operator, 96, 117  
 multidimensional array, 214  
 /\*...\*/ multiline comment, 65  
 \*= multiply assignment, 99

## N

nested conditional operator, 105  
 nested for loop, 231  
 nested if-else, 150  
 new line character (\n), 91–92  
 \n new line, 92  
 != not equal to, 98  
 NOT operator, 100  
 NULL, 245–246, 249, 251, 268  
 null pointer, 268  
 number systems, 17  
   binary, 20  
   decimal, 18  
   hexadecimal, 21  
   octal, 22

## O

%o, 89  
 object code, 55–56  
 octal number system, 22  
 octal to binary conversion, 29  
 octal to decimal conversion, 29  
 one-dimensional array, 214–225  
   characteristics, 218  
   declaration, 215  
   examples, 221  
   initialization, 216  
 one's complement, 106–107, 417, 419  
 open a file, 367–368, 370  
 operand, 54, 95  
 operating system, 12  
 operator, 95–113  
   arithmetic, 96  
   assignment, 98  
   bit-wise, 105  
   conditional, 103  
   increment and decrement, 100  
   logical, 99  
   precedence, 117  
   relational, 97

  special, 112  
   ternary, 103  
 ++ operator, 100–101  
 -- operator, 100–101  
 OR operator, 100  
 output unit, 7

## P

palindrome, 164–165, 171, 251  
 parameters, 182–183, 188  
 pass by reference, 277, 294  
 pass by value, 276, 280  
 passing argument to a function, 188, 191–194  
 passing array to function, 285  
 pointer, 259  
   arithmetic, 272  
   constant pointer, 268  
   declaration, 261  
   definition, 261  
   null pointer, 268  
   pass by reference, 277  
   pointer and array, 282  
   void pointer, 266  
 pointer and array, 282  
 pointer to function, 297  
 pointer to pointer, 265  
 postfix, 101–103  
 pre-defined function, 182  
 prefix operator, 101  
 # preprocessor, 65, 68, 389  
   directive, 389  
 primary memory, 7–8  
 prime number, 188–190  
 printf(), 128  
 problem-solving, 39–40  
 \' produce a single quote, 92  
 \" produce a double quote, 92  
 \0 produce a null character, 92  
 \? produce a question mark, 92  
 \\ produce a single backslash, 92  
 program, 3, 53  
 programming, 53  
   language, 53  
 putchar(), 124–125  
 puts(), 125  
 putw(), 374

## R

RAM, 9–10  
 \r carriage return, 92  
 reading from file, 372, 374–375

real constant, 87  
 realloc(), 352, 356  
 recursion, 195  
 registers, 7–8  
 register variables, 206  
 relational operator, 97  
 %= remainder assignment, 99  
 % remainder operator, 51, 96  
 return type, 83, 183, 187–188, 191–194  
 >> right-shift operator, 106, 109  
 ROM, 9–10  
 rvalue, 98–99

## S

scanf(), 134  
 scientific notation (exponential form), 87  
 search an element in an array, 222  
 secondary memory, 7, 10  
 self-referential structure, 324  
 semicolon, 66, 69, 88  
 signed magnitude, 417  
 // single line comment, 65  
 sizeof(), 78, 112–113, 117, 310, 337  
 software, 12  
 special character, 77  
 stack segment, 351  
 static memory allocation, 351  
 static variable, 207, 351  
 storage class, 204
 

- automatic, 205
- extern, 207
- register, 206
- static, 206

 storage devices, 2, 10
 

- HDD, 10
- SSD, 11

 string, 245–255
 

- compare, 254
- concatenation, 253
- copy, 253
- declaration, 245
- display, 247
- functions, 252
- initialization, 245
- length, 249
- palindrome, 251
- reading, 246–247
- reverse, 250
- strcat(), 253
- strcmp(), 253
- strlen(), 249
- strlwr(), 253

strncmp(), 253  
 strncpy(), 253  
 strrev(), 250, 263  
 structure, 305  
 subscript, 217  
 subscripted variable, 217, 224  
 -=subtract assignment, 99  
 switch, 155  
 system software, 13

## T

terabyte, 9  
 ternary operator, 103  
 text file, 367  
 text segments, 351  
 \t horizontal tab, 92  
 tokens, 77  
 two-dimensional array, 225–242
 

- declaration, 226
- initialization, 228
- matrix addition, 238
- matrix multiplication, 240
- memory representation, 226
- passing 2D array to function, 291

 two's complement, 107, 421  
 type casting, 115–116
 

- explicit, 116
- implicit, 115

 typedef, 308, 321–322  
 type modifiers, 82–83, 89
 

- long, 82, 89
- short, 82, 89
- signed, 82, 89
- unsigned, 82, 89

## U

unary operator, 95, 100, 107  
 underscore, 79  
 unformatted functions, 124–128
 

- getch(), 124, 126–127
- getchar(), 124
- getche(), 124, 126–127
- gets(), 124–125
- putch(), 124, 127
- putchar(), 124
- puts(), 124–125

 union, 78, 80, 333  
 unsigned keyword, 78, 82–83, 89  
 user-defined function, 67, 182



**V**

value at address operator, 262, 270–272, 283  
variables, 79, 84  
void keyword, 68, 78, 80–81, 83  
void pointer, 266  
volatile memory, 8, 10  
\v vertical tab, 92

**W**

while loop, 161  
writing to a file, 371

**X**

%X, 89  
xor operator (^), 106, 117