Data Structures Using C

Second Edition

Reema Thareja

Assistant Professor
Department of Computer Science
Shyama Prasad Mukherji College for Women
University of Delhi

OXFORD
UNIVERSITY PRESS

OXFORD
UNIVERSITY PRESS

Oxford University Press is a department of the University of Oxford.
It furthers the University's objective of excellence in research, scholarship,
and education by publishing worldwide. Oxford is a registered trade mark of
Oxford University Press in the UK and in certain other countries.

Published in India by
Oxford University Press
YMCA Library Building, 1 Jai Singh Road, New Delhi 110001, India

First Edition published in 2011
Second Edition published in 2014

ISBN-13: 978-0-19-809930-7
ISBN-10: 0-19-809930-4

Typeset in Times New Roman
by Pee-Gee Graphics, New Delhi
Printed in India by Radha Press, New Delhi 110031

I dedicate this book to my family
and
my uncle Mr B.L. Theraja

Preface to the Second Edition

A data structure is the logical or mathematical arrangement of data in memory. It considers not only the physical layout of the data items in the memory but also the relationships between these data items and the operations that can be performed on these items. The choice of appropriate data structures and algorithms forms the fundamental step in the design of an efficient program. Thus, a thorough understanding of data structure concepts is essential for students who wish to work in the design and implementation of software systems. C, a general-purpose programming language, having gained popularity in both academia and industry serves as an excellent choice for learning data structures.

This second edition of *Data Structures Using C* has been developed to provide a comprehensive and consistent coverage of both the abstract concepts of data structures as well as the implementation of these concepts using C language. The book utilizes a systematic approach wherein the design of each of the data structures is followed by algorithms of different operations that can be performed on them, and the analysis of these algorithms in terms of their running times.

New to the Second Edition

Based on the suggestions from students and faculty members, this edition has been updated and revised to increase the clarity of presentation where required. Some of the prominent changes are as follows:

- New sections on omega and theta notations, multi-linked lists, forests, conversion of general trees into binary trees, 2-3 trees, binary heap implementation of priority queues, interpolation search, jump search, tree sort, bucket hashing, cylinder surface indexing
- Additional C programs on header linked lists, parentheses checking, evaluation of prefix expressions, priority queues, multiple queues, tree sort, file handling , address calculation sort
- New appendices on dynamic memory allocation, garbage collection, backtracking, Johnson's problem
- Stacks and queues and multi-way search trees are now covered in separate chapters with a more comprehensive explanation of concepts and applications

Extended Material

Chapter 1—This chapter has been completely restructured and reorganized so that it now provides a brief recapitulation of C constructs and syntax. Functions and pointers which were included as independent chapters in the first edition have now been jointly included in this chapter.

Chapter 2—New sections on primitive and non-primitive data structures, different approaches to designing algorithms, omega, theta, and little notations have been included. A number of new examples have also been added which show how to find the complexity of different functions.

Chapter 5—This chapter now includes brief sections on unions, a data type similar to structures.

Chapter 6—This chapter has been expanded to include topics on multi-linked lists, multi-linked list implementation of sparse matrices, and a C program on header linked lists.

Chapter 7—New C programs on parenthesis checking and evaluation of prefix expressions have been added. Recursion, which is one of the most common applications of stacks, has been moved to this chapter.

Chapter 8—New C programs on priority queues and multiple queues have been included.

Chapter 9—This chapter now includes sections on general trees, forests, conversion of general trees into binary trees, and constructing a binary tree from traversal results.

Chapter 10—An algorithm for in-order traversal of a threaded binary tree has been added.

Chapter 11—A table summarizing the differences between B and B+ trees and a section on 2-3 trees have been included.

Chapter 12—A brief section on how binary heaps can be used to implement priority queues has been added.

Chapter 13—This chapter now includes a section which shows the adjacency multi-list representation of graphs.

Chapter 14—As a result of organization, the sections on linear and binary search have been moved from Chapter 3 to this chapter. New search techniques such as interpolation search, jump search, and Fibonacci search have also been included. The chapter also extends the concept of sorting by including sections on practical considerations for internal sorting, sorting on multiple keys, and tree sort.

Chapter 15—New sections on bucket hashing and rehashing have been included.

Chapter 16—This chapter now includes a section on cylinder surface indexing which is one of the widely used indexing structures for files stored in hard disks.

Content and Coverage

This book is organized into 16 chapters.

Chapter 1, *Introduction to C* provides a review of basic C constructs which helps readers to familiarize themselves with basic C syntax and concepts that will be used to write programs in this book.

Chapter 2, Introduction to Data Strctures and Algorithms introduces data structures and algorithms which serve as building blocks for creating efficient programs. The chapter explains how to calculate the time complexity which is a key concept for evaluating the performance of algorithms.

From *Chapter 3* onwards, every chapter discusses individual data structures in detail.

Chapter 3, Arrays provides a detailed explanation of arrays that includes one-dimensional, two-dimensional, and multi-dimensional arrays. The operations that can be performed on such arrays are also explained.

Chapter 4, Strings discusses the concept of strings which are also known as character arrays. The chapter not only focuses on reading and writing strings but also explains various operations that can be used to manipulate the character arrays.

Chapter 5, Structures and Unions deals with structures and unions. A structure is a collection of related data items of different types which is used for implementing other data structures such as linked lists, trees, graphs, etc. We will also read about unions which is also a collection of variables of different data types, except that in case of unions, we can only store information in one field at any one time.

Chapter 6, Linked Lists discusses different types of linked lists such as singly linked lists, doubly linked lists, circular linked lists, doubly circular linked lists, header linked lists, and multi-linked lists. Linked list is a preferred data structure when it is required to allocate memory dynamically.

Chapter 7, Stacks focuses on the concept of last-in, first-out (LIFO) data structure called stacks. The chapter also shows the practical implementation of these data structures using arrays as well as linked lists. It also shows how stacks can be used for the evaluation of arithmetic expressions.

Chapter 8, Queues deals with the concept of first-in, first-out (FIFO) data structure called queues. The chapter also provides the real-world applications of queues.

Chapter 9, Trees focuses on binary trees, their traversal schemes and representation in memory. The chapter also discusses expression trees, tournament trees, and Huffman trees, all of which are variants of simple binary trees.

Chapter 10,Efficient Binary Trees broadens the discussion on trees taken up in *Chapter 9* by going one step ahead and discussing efficient binary trees. The chapter discusses binary search trees, threaded binary trees, AVL trees, red-black trees, and splay trees.

Chapter 11, *Multi-way Search Trees* explores trees which can have more than one key value in a single node, such as M-way search trees, B trees, B+ trees, tries, and 2-3 trees.

Chapter 12, Heaps discusses three types of heaps—binary heaps, binomial heaps, and Fibonacci heaps. The chapter not only explains the operations on these data structures but also makes a comparison, thereby highlighting the key features of each structure.

Chapter 13, Graphs contains a detailed explanation of non-linear data structure called graphs. It discusses the memory representation, traversal schemes, and applications of graphs in the real world.

Chapter 14, Searching and Sorting covers two of the most common operations in computer science, i.e. searching and sorting a list of values. It gives the technique, complexity, and program for different searching and sorting techniques.

Chapter 15, Hashing and Collision deals with different methods of hashing and techniques to resolve collisions.

Chapter 16, the last chapter of the book, *Files and Their Organization*, discusses the concept related to file organization. It explains the different ways in which files can be organized on the hard disk and the indexing techniques that can be used for fast retrieval of data.

The book also provides a set of seven appendices.

Appendix A introduces the concept of dynamic memory allocation in C programs.

Appendix B provides a brief discussion of garbage collection technique which is used for automatic memory management.

Appendix C explains backtracking which is a recursive algorithm that uses stacks.

Appendix D discusses Johnson's algorithm which is used in applications where an optimal order of execution of different activities has to be determined.

Appendix E includes two C programs which show how to read and write binary files.

Appendix F includes a C program which shows how to sort a list of numbers using address calculation sort.

Appendix G provides chapter-wise answers to all the objective questions.

Reema Thareja

Preface to the First Edition

A data structure is defined as a group of data elements used for organizing and storing data. In order to be effective, data has to be organized in a manner that adds to the efficiency of an algorithm, and data structures such as stacks, queues, linked lists, heaps, and trees provide different capabilities to organize data.

While developing a program or an application, many developers find themselves more interested in the type of algorithm used rather than the type of data structure implemented. However, the choice of data structure used for a particular algorithm is always of the utmost importance. Each data structure has its own unique properties and is constructed to suit various kinds of applications. Some of them are highly specialized to carry out specific tasks. For example, B-trees with their unique ability to organize indexes are well-suited for the implementation of databases. Similarly, stack, a linear data structure which provides 'last-in-first-out' access, is used to store and track the sequence of web pages while we browse the Internet. Specific data structures are essential components of many efficient algorithms, and make possible the management of large amounts of data, such as large databases and Internet indexing services. C, as we all know, is the most popular programming language and is widespread among all the computer architectures. Therefore, it is not only logical but also fundamentally essential to start the introduction and implementation of various data structures through C. The course *data structures* is typically taught in the second or third semester of most engineering colleges and across most engineering disciplines in India. The aim of this course is to help students master the design and applications of various data structures and use them in writing effective programs.

About the Book

This book is aimed at serving as a textbook for undergraduate engineering students of computer science and postgraduate level courses of computer applications. The objective of this book is to introduce the concepts of data structures and apply these concepts in problem solving. The book provides a thorough and comprehensive coverage of the fundamentals of data structures and the principles of algorithm analysis. The main focus has been to explain the principles required to select or design the data structure that will best solve the problem.

A structured approach is followed to explain the process of problem solving. A theoretical description of the problem is followed by the underlying technique. These are then ably supported by an example followed by an algorithm, and finally the corresponding program in C language.

The salient features of the book include:

- Explanation of the concepts using diagrams
- Numerous solved examples within the chapters
- Glossary of important terms at the end of each chapter
- Comprehensive exercises at the end of each chapter
- Practical implementation of the algorithms using tested C programs
- Objective type questions to enhance the analytical ability of the students

- Annexures to provide supplementary information to help generate further interest in the subject

The book is also useful as a reference and resource to young researchers working on efficient data storage and related applications, who will find it to be a helpful guide to the newly established techniques of a rapidly growing research field.

Acknowledgements

The writing of this textbook was a mammoth task for which a lot of help was required from many people. Fortunately, I have had the fine support of my family, friends, and fellow members of the teaching staff at the Institute of Information Technology and Management (IITM). My special thanks would always go to my father Mr Janak Raj Thareja and mother Mrs Usha Thareja, my brother Pallav and sisters Kimi and Rashi who were a source of abiding inspiration and divine blessings for me. I am especially thankful to my son Goransh who has been very patient and cooperative in letting me realize my dreams. My sincere thanks go to my uncle Mr B.L. Thareja for his inspiration and guidance in writing this book.

I would also like to thank my students and colleagues at IITM who had always been there to extend help while designing and testing the algorithms. Finally, I would like to thank the editorial team at Oxford University Press for their help and support.

Comments and suggestions for the improvement of the book are welcome. Please send them to me at reemathareja@gmail.com

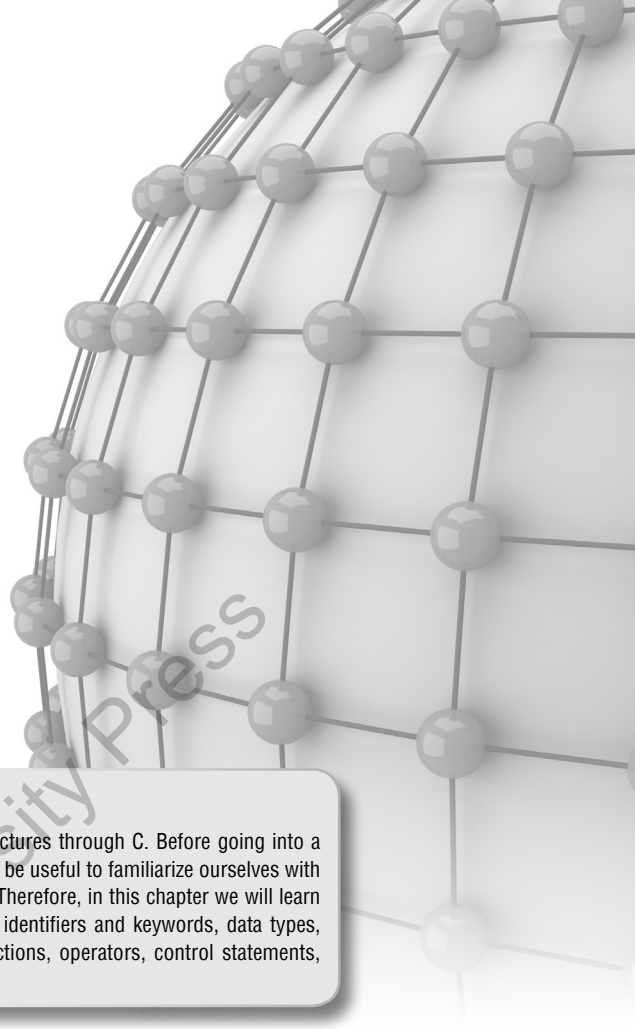
Reema Thareja

Brief Contents

Detailed Contents

CHAPTER 1

Introduction to C



LEARNING OBJECTIVE

This book deals with the study of data structures through C. Before going into a detailed analysis of data structures, it would be useful to familiarize ourselves with the basic knowledge of programming in C. Therefore, in this chapter we will learn about the various constructs of C such as identifiers and keywords, data types, constants, variables, input and output functions, operators, control statements, functions, and pointers.

1.1 INTRODUCTION

The programming language 'C' was developed in the early 1970s by Dennis Ritchie at Bell Laboratories. Although C was initially developed for writing system software, today it has become such a popular language that a variety of software programs are written using this language. The greatest advantage of using C for programming is that it can be easily used on different types of computers. Many other programming languages such as C++ and Java are also based on C which means that you will be able to learn them easily in the future. Today, C is widely used with the UNIX operating system.

Structure of a C Program

A C program contains one or more functions, where a function is defined as a group of statements that perform a well-defined task. Figure 1.1 shows the structure of a C program. The statements in a function are written in a logical sequence to perform a specific task. The `main()` function is the most important function and is a part of every C program. Rather, the execution of a C program begins with this function.

From the structure given in Fig. 1.1, we can conclude that a C program can have any number of functions depending on the tasks that have to be performed, and each function can have any number

```
main()
{
          Statement 1;
          Statement 2;
          ............
          ............
          Statement N;
}
Function1()
{
          Statement 1;
          Statement 2;
          ............
          ............
          Statement N;
}
Function2()
{
          Statement 1;
          Statement 2;
          ............
          ............
          Statement N;
}
..................
..................
FunctionN()
{
          Statement 1;
          Statement 2;
          ............
          ............
          Statement N;
}
```

Figure 1.1 Structure of a C program

of statements arranged according to specific meaningful sequence. Note that programmers can choose any name for functions. It is not mandatory to write Function1, Function2, etc., with an exception that every program must contain one function that has its name as `main()`.

1.2 IDENTIFIERS AND KEYWORDS

Every word in a C program is either an identifier or a keyword.

Identifiers

Identifiers are basically names given to program elements such as variables, arrays, and functions. They are formed by using a sequence of letters (both uppercase and lowercase), numerals, and underscores.
Following are the rules for forming identifier names:

- Identifiers cannot include any special characters or punctuation marks (like #, $, ^, ?, ., etc.) except the underscore "_".
- There cannot be two successive underscores.
- Keywords cannot be used as identifiers.
- The case of alphabetic characters that form the identifier name is significant. For example, 'FIRST' is different from 'first' and 'First'.
- Identifiers must begin with a letter or an underscore. However, use of underscore as the first character must be avoided because several complier-defined identifiers in the standard C library have underscore as their first character. So, inadvertently duplicated names may cause definition conflicts.
- Identifiers can be of any reasonable length. They should not contain more than 31 characters. (They can actually be longer than 31, but the compiler looks at only the first 31 characters of the name.)

Keywords

Like every computer language, C has a set of reserved words often known as keywords that cannot be used as an identifier. All keywords are basically a sequence of characters that have a fixed meaning. By convention, all keywords must be written in lower case letters. Table 1.1 contains the list of keywords in C.

Table 1.1 Keywords in C language

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

1.3 BASIC DATA TYPES

Data type determines the set of values that a data item can take and the operations that can be performed on the item. C language provides four basic data types. Table 1.2 lists the data types, their size, range, and usage for a C programmer.

The `char` data type is of one byte and is used to store single characters. Note that C does not provide any data type for storing text. This is because text is made up of individual characters.You

might have been surprised to see that the range of `char` is given as –128 to 127. `char` is supposed to store characters not numbers, so why this range? The answer is that in the memory, characters are stored in their ASCII codes. For example, the character 'A' has the ASCII code of 65. In memory we will not store 'A' but 65 (in binary number format).

Table 1.2 Basic data types in C

Data Type	Size in Bytes	Range	Use
char	1	-128 to 127	To store characters
int	2	-32768 to 32767	To store integer numbers
float	4	3.4E-38 to 3.4E+38	To store floating point numbers
double	8	1.7E-308 to 1.7E+308	To store big floating point numbers

In addition, C also supports four modifiers—two sign specifiers (`signed` and `unsigned`) and two size specifiers (`short` and `long`). Table 1.3 shows the variants of basic data types.

Table 1.3 Basic data types and their variants

Data Type	Size in Bytes	Range
char	1	-128 to 127
unsigned char	1	0 to 255
signed char	1	-128 to 127
int	2	-32768 to 32767
unsigned int	2	0 to 65535
signed int	2	-32768 to 32767
short int	2	-32768 to 32767
unsigned short int	2	0 to 65535
signed short int	2	-32768 to 32767
long int	4	-2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
signed long int	4	-2147483648 to 2147483647
float	4	3.4E-38 to 3.4E+38
double	8	1.7E-308 to 1.7E+308
long double	10	3.4E-4932 to 1.1E+4932

Note When the basic data type is omitted from a declaration, then automatically type `int` is assumed. For example,

```
long var;    //int is implied
```

While the smaller data types take less memory, the larger data types incur a performance penalty. Although the data type we use for our variables does not have a big impact on the speed or memory usage of the application, we should always try to use `int` unless there is a need to use any other data type.

1.4 VARIABLES AND CONSTANTS

A variable is defined as a meaningful name given to a data storage location in the computer memory. When using a variable, we actually refer to the address of the memory where the data is stored. C language supports two basic kinds of variables.

Numeric Variables

Numeric variables can be used to store either integer values or floating point values. Modifiers like `short`, `long`, `signed`, and `unsigned` can also be used with numeric variables. The difference between signed and unsigned numeric variables is that signed variables can be either negative or positive but unsigned variables can only be positive. Therefore, by using an unsigned variable we can increase the maximum positive range. When we omit the `signed/unsigned` modifier, C language automatically makes it a signed variable. To declare an unsigned variable, the `unsigned` modifier must be explicitly added during the declaration of the variable.

Character Variables

Character variables are just single characters enclosed within single quotes. These characters could be any character from the ASCII character set—letters (‘a’, ‘A’), numerals (‘2’), or special characters (‘&’).

Declaring Variables

To declare a variable, specify the data type of the variable followed by its name. The data type indicates the kind of values that the variable can store. Variable names should always be meaningful and must reflect the purpose of their usage in the program. In C, variable declaration always ends with a semi-colon. For example,

```
int emp_num;
float salary;
char grade;
double balance_amount;
unsigned short int acc_no;
```

In C, variables can be declared at any place in the program but two things must be kept in mind. First, variables should be declared before using them. Second, variables should be declared closest to their first point of use so that the source code is easier to maintain.

Initializing Variables

While declaring the variables, we can also initialize them with some value. For example,

```
int emp_num = 7;
float salary = 9800.99
char grade = 'A';
double balance_amount = 100000000;
```

Constants

Constants are identifiers whose values do not change. While values of variables can be changed at any time, values of constants can never be changed. Constants are used to define fixed values like *pi* or the charge on an electron so that their value does not get changed in the program even by mistake.

Declaring Constants

To declare a constant, precede the normal variable declaration with `const` keyword and assign it a value.

```
const float pi = 3.14;
```

1.5 WRITING THE FIRST C PROGRAM

To write a C program, we first need to write the code. For that, open a text editor. If you are a Windows user, you may use Notepad and if you prefer working on UNIX/Linux, you can use `emac` or `vi`. Once the text editor is opened on your screen, type the following statements:

```
#include <stdio.h>
int main()
{
      printf("\n Welcome to the world of C ");// prints the message on the screen
      return 0;// returns a value 0 to the operating system
}
```

After writing the code, select the directory of your choice and save the file as `first.c`.

`#include <stdio.h>` This is the first statement in our code that includes a file called `stdio.h`. This file has some in-built functions. By simply including this file in our code, we can use these functions directly. `stdio` basically stands for Standard Input/Output, which means it has functions for input and output of data like reading values from the keyboard and printing the results on the screen.

`int main()` Every C program `contains` a `main()` function which is the starting point of the program. `int` is the return value of the `main` function. After all the statements in the program have been executed, the last statement of the program will return an integer value to the operating system. The concepts will be clear to us when we read this chapter in toto. So even if you do not understand certain things, do not worry.

`{ }` The two curly brackets are used to group all the related statements of the `main` function.

`printf("\n Welcome to the world of C ");` The `printf` function is defined in the `stdio.h` file and is used to print text on the screen. The message that has to be displayed on the screen is enclosed within double quotes and put inside brackets.

`\n` is an escape sequence and represents a newline character. It is used to print the message on a new line on the screen. Other escape sequences supported by C language are shown in Table 1.4.

Table 1.4 Escape sequences

Escape Sequence	Purpose
\a	Audible signal
\b	Backspace
\t	Tab
\n	New line
\v	Vertical tab
\f	New page\Clear screen
\r	Carriage return

`return 0;` This is a return command that is used to return value 0 to the operating system to give an indication that there were no errors during the execution of the program.

Note Every statement in the main function ends with a semi-colon (;).

`first.c.` If you are a Windows user, then open the command prompt by clicking Start→Run and typing "command" and clicking Ok. Using the command prompt, change to the directory in which you saved your file and then type:

```
C:\>tc first.c
```

In case you are working on UNIX/Linux operating system, then exit the text editor and type

```
$cc first.c -ofirst
```

The `-o` is for the output file name. If you leave out the `-o`, then the file name `a.out` is used.

This command is used to compile your C program. If there are any mistakes in the program, then the compiler will tell you what mistake(s) you have made and on which line the error has occurred. In case of errors, you need to re-open your `.c` file and correct the mistakes. However, if everything is right, then no error(s) will be reported and the compiler will create an `.exe` file for your program. This `.exe` file can be directly run by typing

`"first.exe"` for Windows and `"./first"` for UNIX/Linux operating system

When you run the `.exe` file, the output of the program will be displayed on screen. That is,

```
Welcome to the world of C
```

Note The printf and return statements have been indented or moved away from the left side. This is done to make the code more readable.

Using Comments

Comments are a way of explaining what a program does. C supports two types of comments.

- // is used to comment a single statement.
- /* is used to comment multiple statements. A /* is ended with */ and all statements that lie between these characters are commented.

Note that comment statements are not executed by the compiler. Rather, they are ignored by the compiler as they are simply added in programs to make the code understandable by programmers as well as other users. It is a good habit to always put a comment at the top of a program that tells you what the program does. This helps in defining the usage of the program the moment you open it.

Standard Header Files

Till now we have used `printf()` function, which is defined in the `stdio.h` header file. Even in other programs that we will be writing, we will use many functions that are not written by us. For example, to use the `strcmp()` function that compares two strings, we will pass string arguments and retrieve the result. We do not know the details of how these functions work. Such functions that are provided by all C compilers are included in standard header files. Examples of these standard header files include:

- `string.h` : for string handling functions
- `stdlib.h` : for some miscellaneous functions
- `stdio.h` : for standardized input and output functions
- `math.h` : for mathematical functions
- `alloc.h` : for dynamic memory allocation
- `conio.h` : for clearing the screen

All the header files are referenced at the start of the source code file that uses one or more functions from these files.

1.6 INPUT AND OUTPUT FUNCTIONS

The most fundamental operation in a C program is to accept *input* values from a standard input device and *output* the data produced by the program to a standard output device. As shown in Section 1.4, we can assign values to variables using the assignment operator '='. For example,

```
int a = 3;
```

What if we want to assign value to variable `a` that is inputted from the user at run-time? This is done by using the `scanf` function that reads data from the keyboard. Similarly, for outputting results of

the program, `printf` function is used that sends results to a terminal. Like `printf` and `scanf`, there are different functions in C that can carry out the input–output operations. These functions are collectively known as Standard Input/Output Library. A program that uses standard input/output functions must contain the following statement at the beginning of the program:

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

scanf()

The `scanf()`function is used to read formatted data from the keyboard. The syntax of the `scanf()` function can be given as,

```
scanf ("control string", arg1, arg2, arg3...argn);
```

The control string specifies the type and format of the data that has to be obtained from the keyboard and stored in the memory locations pointed by the arguments, `arg1, arg2, ...,argn`. The prototype of the control string can be given as,

```
%[*][width][modifier]type
```

***** is an optional argument that suppresses assignment of the input field. That is, it indicates that data should be read from the stream but ignored (not stored in the memory location).

width is an optional argument that specifies the maximum number of characters to be read. However, if the `scanf` function encounters a white space or an unconvertible character, input is terminated.

modifier is an optional argument (**h, l,** or **L)** , which modifies the type specifier. Modifier **h** is used for `short int` or `unsigned short int`, **l** is used for `long int`, `unsigned long int`, or `double` values. Finally, **L** is used for `long double` data values.

type specifies the type of data that has to be read. It also indicates how this data is expected to be read from the user. The type specifiers for `scanf` function are given in Table 1.5.

Table 1.5 Type specifiers

Type	Qualifying Input
%c	For single characters
%d, %i	For integer values
%e,%E,%f,%g,%G	For floating point numbers
%o	For octal numbers
%s	For a sequence of (string of) characters
%u	For unsigned integer values
%x,%X	For hexadecimal values

The `scanf` function ignores any blank spaces, tabs, and newlines entered by the user. The function simply returns the number of input fields successfully scanned and stored.

As we have not studied functions till now, understanding `scanf` function in depth will be a bit difficult here, but for now just understand that the `scanf` function is used to store values in memory locations associated with variables. For this, the function should have the address of the variables. The address of the variable is denoted by an & sign followed by the name of the variable. Look at the following code that shows how we can input value in a variable of `int` data type:

```
int num;
scanf(" %4d ", &num);
```

The `scanf` function reads first four digits into the address or the memory location pointed by `num`.

Note In case of reading strings, we do not use the & sign in the `scanf` function.

printf()

The `printf` function is used to display information required by the user and also prints the values of the variables. Its syntax can be given as:

```
printf ("control string", arg1,arg2,arg3,...,argn);
```

After the control string, the function can have as many arguments as specified in the control string. The control string contains format specifiers which are arranged in the order so that they correspond with the arguments in the variable list. It may also contain text to be printed such as instructions to the user, identifier names, or any other text to make the text readable.

Note that there must be enough arguments because if there are not enough arguments, then the result will be completely unpredictable. However, if by mistake you specify more number of arguments, the excess arguments will simply be ignored. The prototype of the control string can be given as below:

```
%[flags][width][.precision][modifier]type
```

Each control string must begin with a % sign.

flags is an optional argument, which specifies output justification like decimal point, numerical sign, trailing zeros or octadecimal or hexadecimal prefixes. Table 1.6 shows different types of flags with their descriptions.

Table 1.6 Flags in printf()

Flags	Description
–	Left-justify within the given field width
+	Displays the data with its numeric sign (either + or –)
#	Used to provide additional specifiers like o, x, X, 0, 0x, or 0X for octal and hexadecimal values respectively for values different than zero
0	The number is left-padded with zeroes (0) instead of spaces

width is an optional argument which specifies the minimum number of positions that the output characters will occupy. If the number of output characters is smaller than the specified width, then the output would be right justified with blank spaces to the left. However, if the number of characters is greater than the specified width, then all the characters would be printed.
precision is an optional argument which specifies the number of digits to print after the decimal point or the number of characters to print from a string.
modifier field is same as given for `scanf()` function.
type is used to define the type and the interpretation of the value of the corresponding argument. The type specifiers for `printf` function are given in Table 1.5.

The most simple `printf` statement is

```
printf ("Welcome to the world of C language");
```

The function when executed prompts the message enclosed in the quotation to be displayed on the screen.

For `float x = 8900.768`, the following examples show output under different format specifications:

```
printf ("%f", x)
```

8	9	0	0	.	7	6	8

```
printf("%10f", x);
```

		8	9	0	0	.	7	6	8

```
printf("%9.2f", x);
```

		8	9	0	0	.	7	7

```
printf("%6f", x);
```

8	9	0	0	.	7	6	8

1.7 OPERATORS AND EXPRESSIONS

C language supports different types of operators, which can be used with variables and constants to form expressions. These operators can be categorized into the following major groups:

- Arithmetic operators
- Equality operators
- Unary operators
- Bitwise operators
- Comma operator
- Relational operators
- Logical operators
- Conditional operator
- Assignment operators
- Sizeof operator

We will now discuss all these operators.

Arithmetic Operators

Consider three variables declared as,

```
int a=9, b=3, result;
```

We will use these variables to explain arithmetic operators. Table 1.7 shows the arithmetic operators, their syntax, and usage in C language.

Table 1.7 Arithmetic operators

Operation	Operator	Syntax	Comment	Result
Multiply	*	a * b	result = a * b	27
Divide	/	a / b	result = a / b	3
Addition	+	a + b	result = a + b	12
Subtraction	–	a – b	result = a – b	6
Modulus	%	a % b	result = a % b	0

In Table 1.7, `a` and `b` (on which the operator is applied) are called **operands**. Arithmetic operators can be applied to any integer or floating point number. The addition, subtraction, multiplication, and division (`+`, `–`, `*`, and `/`) operators are the usual arithmetic operators, so you are already familiar with these operators.

However, the operator % might be new to you. The modulus operator (%) finds the remainder of an integer division. This operator can be applied only on integer operands and cannot be used on float or double operands.

While performing modulo division, the sign of the result is always the sign of the first operand (the dividend). Therefore,

```
 16   %    3 =   1
-16   %    3 =  -1
 16   %   -3 =   1
-16   %   -3 =  -1
```

When both operands of the division operator `(/)` are integers, the division is performed as an integer division. Integer division always results in an integer result. So, the result is always rounded-off by ignoring the remainder. Therefore,

```
9/4 = 2  and  -9/4 = -3
```

Note It is not possible to divide any number by zero. This is an illegal operation that results in a run-time division-by-zero exception thereby terminating the program.

Except for modulus operator, all other arithmetic operators can accept a mix of integer and floating point numbers. If both operands are integers, the result will be an integer; if one or both operands are floating point numbers then the result would be a floating point number.

All the arithmetic operators bind from left to right. Multiplication, division, and modulus operators have higher precedence over addition and subtraction operators. Thus, if an arithmetic expression consists of a mix of operators, then multiplication, division, and modulus will be carried out first in a left to right order, before any addition and subtraction can be performed. For example,

```
      3 + 4 * 7
=     3 + 28
=     31
```

Relational Operators

A relational operator, also known as a comparison operator, is an operator that compares two values or expressions. Relational operators return true or false value, depending on whether the conditional relationship between the two operands holds or not.

For example, to test if `x` is less than `y`, relational operator `<` is used as `x < y`. This expression will return true value if `x` is less than `y`; otherwise the value of the expression will be false. C provides four relational operators which are illustrated in Table 1.8. These operators are evaluated from left to right.

Table 1.8 Relational operators

Operator	Meaning	Example
<	Less than	3 < 5 gives 1
>	Greater than	7 > 9 gives 0
<=	Less than or equal to	100 <= 100 gives 1
>=	Greater than equal to	50 >=100 gives 0

Equality Operators

C language also supports two equality operators to compare operands for strict equality or inequality. They are equal to (==) and not equal to (!=) operators. The equality operators have lower precedence than the relational operators.

The equal-to operator (==) returns true (1) if operands on both sides of the operator have the same value; otherwise, it returns false (0). On the contrary, the not-equal-to operator (**!=**) returns true (1) if the operands do not have the same value; else it returns false (0). Table 1.9 summarizes equality operators.

Table 1.9 Equality operators

Operator	Meaning
==	Returns 1 if both operands are equal, 0 otherwise
!=	Returns 1 if operands do not have the same value, 0 otherwise

Logical Operators

C language supports three logical operators. They are logical AND (&&), logical OR (||), and logical NOT (!). As in case of arithmetic expressions, logical expressions are evaluated from left to right.

Table 1.10 Truth table of logical AND

A	B	A && B
0	0	0
0	1	0
1	0	0
1	1	1

Logical AND (&&)

Logical AND is a binary operator, which simultaneously evaluates two values or relational expressions. If both the operands are true, then the whole expression is true. If both or one of the operands is false, then the whole expression evaluates to false. The truth table of logical AND operator is given in Table 1.10.

For example,

```
(a < b) && (b > c)
```

The whole expression is true only if both expressions are true, i.e., if `b` is greater than `a` and `c`.

Logical OR (||)

Logical OR returns a false value if both the operands are false. Otherwise it returns a true value. The truth table of logical OR operator is given in Table 1.11. For example,

Table 1.11 Truth table of logical OR

A	B	A || B
0	0	0
0	1	1
1	0	1
1	1	1

```
(a < b) || (b > c)
```

The whole expression is true if either `b` is greater than `a` or `b` is greater than `c` or `b` is greater than both `a` and `c`.

Logical NOT (!)

The logical NOT operator takes a single expression and produces a zero if the expression evaluates to a non-zero value and produces a 1 if the expression produces a zero. The truth table of logical NOT operator is given in Table 1.12. For example,

Table 1.12 Truth table of logical NOT

A	! A
0	1
1	0

```
int a = 10, b;
b = !a;
```

Now the value of `b = 0`. This is because value of `a = 10`. `!a = 0`. The value of `!a` is assigned to `b`, hence the result.

Unary Operators

Unary operators act on single operands. C language supports three unary operators. They are unary minus, increment, and decrement operators.

Unary Minus (–)

Unary minus operator negates the value of its operand. For example, if a number is positive then it becomes negative when preceded with a unary minus operator. Similarly, if the number is negative, it becomes positive after applying the unary minus operator. For example,

```
int a, b = 10;
a = -(b);
```

The result of this expression is `a = -10`, because variable `b` has a positive value. After applying unary minus operator (–) on the operand `b`, the value becomes –10, which indicates it has a negative value.

Increment Operator (++) and Decrement Operator (––)

The increment operator is a unary operator that increases the value of its operand by 1. Similarly, the decrement operator decreases the value of its operand by 1. For example, `--x` is equivalent to writing `x = x - 1`.

The increment/decrement operators have two variants: *prefix* and *postfix*. In a prefix expression (`++x` or `--x`), the operator is applied before the operand while in a postfix expression (`x++` or `x--`), the operator is applied after the operand.

An important point to note about unary increment and decrement operators is that `++x` is not same as `x++`. Similarly, `--x` is not the same as `x--`. Although, `x++` and `++x` both increment the value of `x` by 1, in the former case, the value of `x` is returned before it is incremented. Whereas in the latter case, the value of `x` is returned after it is incremented. For example,

```
int x = 10, y;
```

`y = x++;` is equivalent to writing

```
y = x;
x = x + 1;
```

Whereas `y = ++x;` is equivalent to writing

```
x = x + 1;
y = x;
```

The same principle applies to unary decrement operators. Note that unary operators have a higher precedence than the binary operators. And if in an expression we have more than one unary operator then they are evaluated from right to left.

Conditional Operator

The syntax of the conditional operator is

```
exp1 ? exp2 : exp3
```

`exp1` is evaluated first. If it is true, then `exp2` is evaluated and becomes the result of the expression, otherwise `exp3` is evaluated and becomes the result of the expression. For example,

```
large = (a > b) ? a : b
```

The conditional operator is used to find the larger of two given numbers. First `exp1`, that is `a > b`, is evaluated. If `a` is greater than `b`, then `large = a`, else `large = b`. Hence, `large` is equal to either `a` or `b`, but not both.

Conditional operators make the program code more compact, more readable, and safer to use as it is easier to both check and guarantee the arguments that are used for evaluation. Conditional operator is also known as ternary operator as it takes three operands.

Bitwise Operators

As the name suggests, bitwise operators perform operations at the bit level. These operators include: bitwise AND, bitwise OR, bitwise XOR, and shift operators.

Bitwise AND

Like boolean AND (&&), bitwise AND operator (&) performs operation on bits instead of bytes, chars, integers, etc. When we use the bitwise AND operator, the bit in the first operand is ANDed with the corresponding bit in the second operand. The truth table is the same as we had seen in logical AND operation. The bitwise AND operator compares each bit of its first operand with the corresponding bit of its second operand. If both bits are 1, the corresponding bit in the result is 1 and 0 otherwise. For example,

```
10101010 & 01010101 = 00000000
```

Bitwise OR

When we use the bitwise OR operator (|), the bit in the first operand is ORed with the corresponding bit in the second operand. The truth table is the same as we had seen in logical OR operation. The bitwise OR operator compares each bit of its first operand with the corresponding bit of its second operand. If one or both bits are 1, the corresponding bit in the result is 1 and 0 otherwise. For example,

```
10101010 | 01010101 = 11111111
```

Bitwise XOR

When we use the bitwise XOR operator, the bit in the first operand is XORed with the corresponding

Table 1.13 Truth table of bitwise XOR

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

bit in the second operand. The truth table of bitwise XOR operator is shown in Table 1.13. The bitwise XOR operator compares each bit of its first operand with the corresponding bit of its second operand. If one of the bits is 1, the corresponding bit in the result is 1 and 0 otherwise. For example,

```
10101010 ^ 01010101 = 11111111
```

Bitwise NOT (~)

The bitwise NOT or complement is a unary operator that performs logical negation on each bit of the operand. By performing negation of each bit, it actually produces the one's complement of the given binary value. Bitwise NOT operator sets the bit to 1 if it was initially 0 and sets it to 0 if it was initially 1. For example,

```
~10101011 = 01010100
```

Shift Operators

C supports two bitwise shift operators. They are shift left (<<) and shift right (>>). The syntax for a shift operation can be given as

```
operand op num
```

where the bits in the operand are shifted left or right depending on the operator (left, if the operator is << and right, if the operator is >>) by number of places denoted by `num`. For example, if we have

```
x = 0001 1101
```

then `x << 1` produces `0011 1010`

When we apply a left shift, every bit in `x` is shifted to the left by one place. So, the MSB (most significant bit) of `x` is lost, the LSB (least significant bit) of `x` is set to 0. Therefore, if we have `x = 0001 1101`, then

```
x << 3 gives result = 1110 1000
```

On the contrary, when we apply a right shift, every bit in `x` is shifted to the right by one place. So, the LSB of `x` is lost, the MSB of `x` is set to 0. For example, if we have `x = 0001 1101`, then

`x >> 1` gives result `= 0000 1110`

Similarly, if we have `x = 0001 1101`, then

`x >> 4` gives result `= 0000 0001`

Note The expression `x << y` is equivalent to multiplication of `x` by 2^y. And the expression `x >> y` is equivalent to division of `x` by 2^y if `x` is unsigned or has a non-negative value.

Assignment Operators

In C language, the assignment operator is responsible for assigning values to the variables. While the equal sign (=) is the fundamental assignment operator, C also supports other assignment operators that provide shorthand ways to represent common variable assignments.

When an equal sign is encountered in an expression, the compiler processes the statement on the right side of the sign and assigns the result to the variable on the left side. For example,

```
int x;
x = 10;
```

assigns the value 10 to variable `x`. The assignment operator has right-to-left associativity, so the expression

Table 1.14 Assignment operators

Operator	Example

/=	float a=9.0;
	float b=3.0;
	a /= b;

\=	int a= 9;
	int b = 3;
	a \= b;

*=	int a= 9;
	int b = 3;
	a *= b;

+=	int a= 9;
	int b = 3;
	a += b;

−=	int a= 9;
	int b = 3;
	a −= b;

&=	int a = 10;
	int b = 20;
	a &= b;

^=	int a = 10;
	int b = 20;
	a ^= b;

<<=	int a= 9;
	int b = 3;
	a <<= b;

>>=	int a= 9;
	int b = 3;
	a >>= b;

```
a = b = c = 10;
```

is evaluated as

```
(a = (b = (c = 10)));
```

First 10 is assigned to `c`, then the value of `c` is assigned to `b`. Finally, the value of `b` is assigned to `a`. Table 1.14 contains a list of other assignment operators that are supported by C.

Comma Operator

The comma operator, which is also called the sequential-evaluation operator, takes two operands. It works by evaluating the first expression and discarding its value, and then evaluates the second expression and returns the value as the result of the expression. Comma-separated expressions when chained together are evaluated in left-to-right sequence with the right-most value yielding the result of the expression. Among all the operators, the comma operator has the lowest precedence.

Therefore, when a comma operator is used, the entire expression evaluates to the value of the right expression. For example, the following statement first increments `a`, then increments `b`, and then assigns the value of `b` to `x`.

```
int a=2, b=3, x=0;
x = (++a, b+=a);
```

Now, the value of `x = 6`.

`sizeof` *Operator*

`sizeof` is a unary operator used to calculate the size of data types. This operator can be applied to all data types. When using this operator, the keyword `sizeof` is followed by a type name, variable, or expression. The operator returns the size of the data type, variable, or expression in bytes. That is, the `sizeof` operator is used to determine the amount of memory space that the data type/variable/expression will take.

When a type name is used, it is enclosed in parentheses, but in case of variable names and expressions, they can be specified with or without parentheses. A `sizeof` expression returns an unsigned value that specifies the size of the space in bytes required by the data type, variable, or expression. For example, `sizeof(char)` returns 1, that is the size of a character data type. If we have,

```
int a = 10;
unsigned int result;
result = sizeof(a);
```

then `result = 2`, that is, space required to store the variable *a* in memory. Since `a` is an integer, it requires 2 bytes of storage space.

Operator Precedence Chart

Table 1.15 lists the operators that C language supports in the order of their *precedence* (highest to lowest). The *associativity* indicates the order in which the operators of equal precedence in an expression are evaluated.

Table 1.15 Operators precedence chart

Operator	Associativity
() [] . –>	left-to-right
++(postfix) --(postfix)	right-to-left
++(prefix) --(prefix) +(unary) – (unary) ! ~ (type) *(indirection) &(address) sizeof	right-to-left
* / %	left-to-right
+ –	left-to-right
<< >>	left-to-right
< <= > >=	left-to-right
== !=	left-to-right
&	left-to-right
^	left-to-right
|	left-to-right
&&	left-to-right
||	left-to-right
?:	right-to-left
= += –= *= /= %= &= ^= |= <<= >>=	right-to-left
,(comma)	left-to-right

Examples of Expressions Using the Precedence Chart

If we have the following variable declarations:

```
int a = 0, b = 1, c = -1;
float x = 2.5, y = 0.0;
```

then,

```
(a)  a && b = 0
(b)  a < b && c < b = 1
(c)  b + c || ! a
     = ( b + c) || (!a)
     = 0 ||1
     = 1
(d)  x * 5 && 5 || ( b / c)
     = ((x * 5) && 5) || (b / c)
     = (12.5 && 5) || (1/-1)
     = 1
(e)  a <= 10 && x >= 1 && b
     = ((a <= 10) && (x >= 1)) && b
     = (1 && 1) && 1
     = 1
(f)  !x || !c || b + c
     = ((!x) || (!c)) || (b + c)
     = (0 || 0) || 0
     = 0
(g)  x * y < a + b || c
     = ((x * y) < (a + b)) || c
     = (0 < 1) || -1
     = 1
(h)  (x > y) + !a || c++
     = ((x > y) + (!a)) || (c++)
     = (1 + 1) || 0
     = 1
```


PROGRAMMING EXAMPLE

1. Write a program to calculate the area of a circle.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        float radius;
        double area;
        clrscr();
        printf("\n Enter the radius of the circle : ");
        scanf("%f", &radius);
        area = 3.14 * radius * radius;
        printf(" \n Area = %.2lf", area);
        return 0;
}
```

Output

```
Enter the radius of the circle : 7
Area = 153.86
```

1.8 TYPE CONVERSION AND TYPECASTING

Type conversion or typecasting of variables refers to changing a variable of one data type into another. While type conversion is done implicitly, casting has to be done explicitly by the programmer. We will discuss both these concepts here.

Type Conversion

Type conversion is done when the expression has variables of different data types. So to evaluate the expression, the data type is promoted from lower to higher level where the hierarchy of data types can be given as: `double`, `float`, `long`, `int`, `short`, and `char`. For example, type conversion is automatically done when we assign an integer value to a floating point variable. Consider the following code:

```
float x;
int y = 3;
x = y;
```

Now, `x = 3.0`, as integer value is automatically converted into its equivalent floating point representation.

Typecasting

Typecasting is also known as *forced conversion*. It is done when the value of one data type has to be converted into the value of another data type. The code to perform typecasting can be given as:

```
float salary = 10000.00;
int sal;
sal = (int) salary;
```

When floating point numbers are converted to integers, the digits after the decimal are truncated. Therefore, data is lost when floating point representations are converted to integral representations.

As we can see in the code, typecasting can be done by placing the destination data type in parentheses followed by the variable name that has to be converted. Hence, we conclude that typecasting is done to make a variable of one data type to act like a variable of another type.

PROGRAMMING EXAMPLE

2. Write a program to convert an integer into the corresponding floating point number.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        float f_num;
        int i_num;
        clrscr();
        printf("\n Enter any integer: ");
        scanf("%d", &i_num);
        f_num = (float)i_num;
        printf("\n The floating point variant of %d is = %f", i_num, f_num);
        return 0;
}
```

Output

```
Enter any integer: 56
The floating point variant of 56 is = 56.000000
```

1.9 CONTROL STATEMENTS

Till now we know that the code in the C program is executed sequentially from the first line of the program to its last line. That is, the second statement is executed after the first, the third statement is executed after the second, so on and so forth. Although this is true, in some cases we want only selected statements to be executed. Control flow statements enable programmers to conditionally execute a particular block of code. There are three types of control statements: decision control (branching), iterative (looping), and jump statements. While branching means deciding what actions have to be taken, looping, on the other hand, decides how many times the action has to be taken. Jump statements transfer control from one point to another point.

1.9.1 Decision Control Statements

C supports decision control statements that can alter the flow of a sequence of instructions. These statements help to jump from one part of the program to another depending on whether a particular condition is satisfied or not. These decision control statements include:

(a) `if` statement,
(b) `if-else` statement,
(c) `if-else-if` statement, and
(d) `switch-case` statement.

`if` *Statement*

`if` statement is the simplest decision control statement that is frequently used in decision making. The general form of a simple `if` statement is shown in Fig. 1.2.

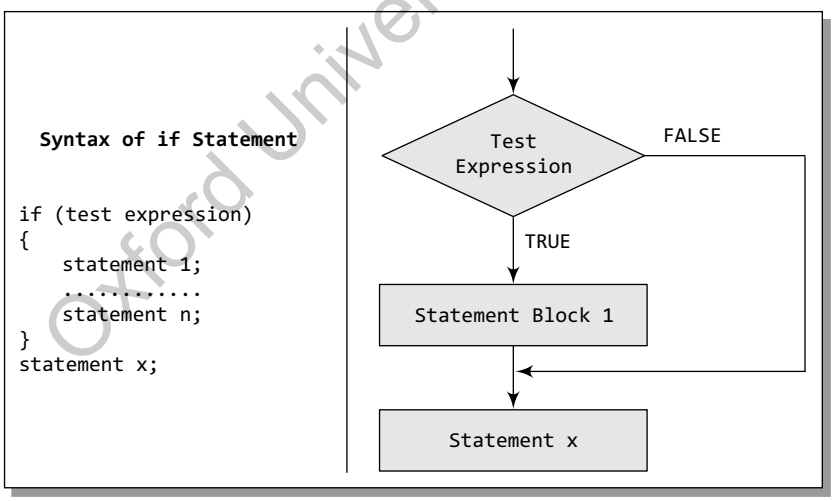


Figure 1.2 `if` statement construct

The `if` block may include 1 statement or *n* statements enclosed within curly brackets. First the test expression is evaluated. If the test expression is true, the statements of the `if` block are executed, otherwise these statements will be skipped and the execution will jump to statement `x`.

The statement in an `if` block is any valid C language statement, and the test expression is any valid C language expression that evaluates to either true or false. In addition to simple relational expressions, we can also use compound expressions formed using logical operators. Note that there is no semi-colon after the test expression. This is because the condition and statement should be put together as a single statement.

```
#include <stdio.h>
int main()
{
      int x=10;
      if (x>0) x++;
      printf("\n x = %d", x);
      return 0;
}
```

In the above code, we take a variable `x` and initialize it to 10. In the test expression, we check if the value of `x` is greater than 0. As 10 > 0, the test expression evaluates to true, and the value of `x` is incremented. After that, the value of `x` is printed on the screen. The output of this program is

```
x = 11
```

Observe that the `printf` statement will be executed even if the test expression is false.

Note In case the statement block contains only one statement, putting curly brackets becomes optional. If there are more than one statement in the statement block, putting curly brackets becomes mandatory.

`if-else` *Statement*

We have studied that using `if` statement plays a vital role in conditional branching. Its usage is very simple. The test expression is evaluated, if the result is true, the statement(s) followed by the expression is executed, else if the expression is false, the statement is skipped by the compiler.

What if you want a separate set of statements to be executed if the expression returns a false value? In such cases, we can use an `if-else` statement rather than using a simple `if` statement. The general form of simple `if-else` statement is shown in Fig. 1.3.

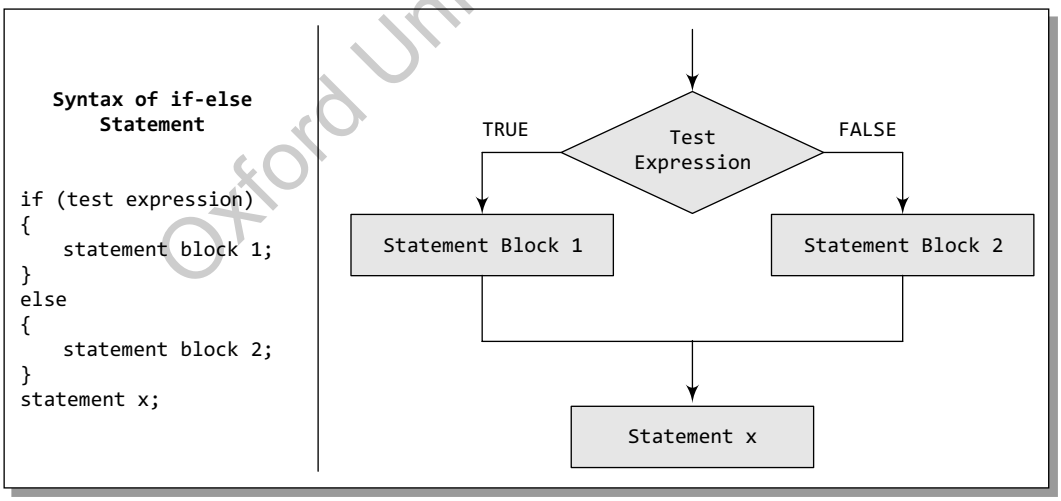


Figure 1.3 `if-else` statement construct

In the `if-else` construct, first the test expression is evaluated. If the expression is true, statement block 1 is executed and statement block 2 is skipped. Otherwise, if the expression is false, statement block 2 is executed and statement block 1 is ignored. In any case after the statement block 1 or 2 gets executed, the control will pass to statement `x`. Therefore, statement `x` is executed in every case.

PROGRAMMING EXAMPLE

3. Write a program to find whether a number is even or odd.

```
#include <stdio.h>
int main()
{
        int a;
        printf("\n Enter the value of a : ");
        scanf("%d", &a);
        if(a%2==0)
                printf("\n %d is even", a);
        else
                printf("\n %d is odd", a);
        return 0;
}
```

Output

```
Enter the value of a : 6
6 is even
```

`if-else-if` *Statement*

C language supports `if-else-if` statements to test additional conditions apart from the initial test expression. The `if-else-if` construct works in the same way as a normal `if` statement. Its construct is given in Fig. 1.4.

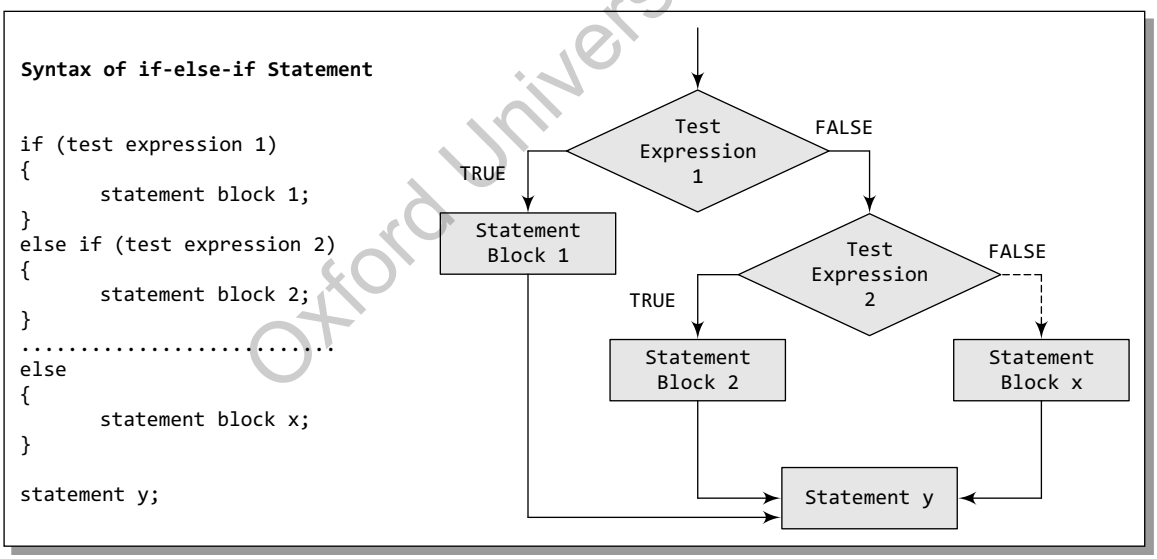


Figure 1.4 `if-else-if` statement construct

Note that it is not necessary that every `if` statement should have an `else` block as C supports simple `if` statements. After the first test expression or the first `if` branch, the programmer can have as many `else-if` branches as he wants depending on the expressions that have to be tested. For example, the following code tests whether a number entered by the user is negative, positive, or equal to zero.

```
#include <stdio.h>
int main()
{
```

```
        int num;
        printf("\n Enter any number : ");
        scanf("%d", &num);
        if(num==0)
                printf("\n The value is equal to zero");
        else if(num>0)
                printf("\n The number is positive");
        else
                printf("\n The number is negative");
        return 0;
}
```

Note that if the first test expression evaluates to a true value, i.e., `num=0`, then the rest of the statements in the code will be ignored and after executing the `printf` statement that displays 'The value is equal to zero', the control will jump to `return 0` statement.

`switch-case` *Statement*

A `switch-case` statement is a multi-way decision statement that is a simplified version of an `if-else-if` block. The general form of a `switch` statement is shown in Fig. 1.5.

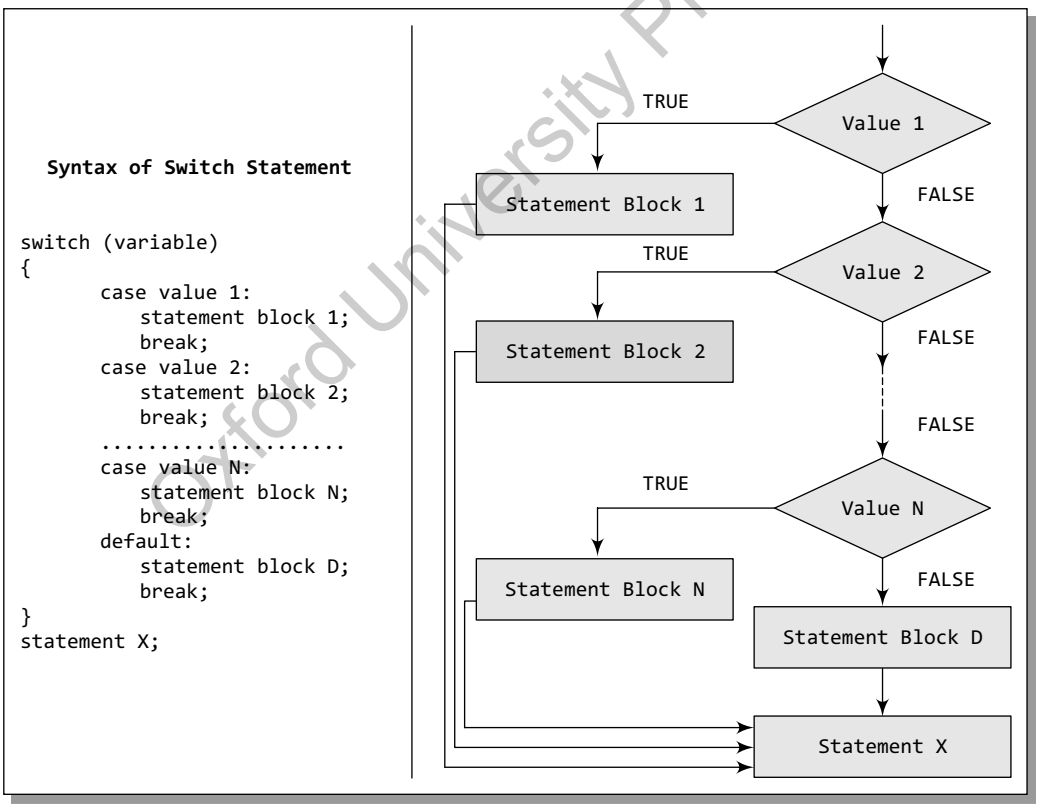


Figure 1.5 `switch-case` statement construct

The power of nested `if-else-if` statements lies in the fact that it can evaluate more than one expression in a single logical structure. `switch` statements are mostly used in two situations:

- When there is only one variable to evaluate in the expression
- When many conditions are being tested for

When there are many conditions to test, using the `if` and `else-if` constructs becomes complicated and confusing. Therefore, `switch case` statements are often used as an alternative to long `if` statements that compare a variable to several 'integral' values (integral values are those values that can be expressed as an integer, such as the value of a `char`). `Switch` statements are also used to handle the input given by the user.

We have already seen the syntax of the `switch` statement. The `switch case` statement compares the value of the variable given in the `switch` statement with the value of each case statement that follows. When the value of the `switch` and the `case` statement matches, the statement block of that particular case is executed.

Did you notice the keyword `default` in the syntax of the `switch case` statement? Default is the case that is executed when the value of the variable does not match with any of the values of the `case` statements. That is, default case is executed when no match is found between the values of `switch` and `case` statements and thus there are no statements to be executed. Although the default case is optional, it is always recommended to include it as it handles any unexpected case.

In the syntax of the `switch-case` statement, we have used another keyword `break`. The `break` statement must be used at the end of each case because if it is not used, then the case that matched and all the following cases will be executed. For example, if the value of `switch` statement matched with that of case 2, then all the statements in case 2 as well as the rest of the cases including default will be executed. The `break` statement tells the compiler to jump out of the switch case statement and execute the statement following the `switch-case` construct. Thus, the keyword `break` is used to break out of the case statements.

Advantages of Using a `switch-case` *Statement*

`Switch-case` statement is preferred by programmers due to the following reasons:

- Easy to debug
- Easy to read and understand
- Ease of maintenance as compared to its equivalent `if-else` statements
- Like `if-else` statements, `switch` statements can also be nested
- Executes faster than its equivalent `if-else` construct

PROGRAMMING EXAMPLE

4. Write a program to determine whether the entered character is a vowel or not.

```
#include <stdio.h>
int main()
{
        char ch;
        printf("\n Enter any character : ");
        scanf("%c", &ch);
        switch(ch)
        {
                case 'A':
                case 'a':
                        printf("\n %c is VOWEL", ch);
                        break;
                case 'E':
                case 'e':
                        printf("\n %c is VOWEL", ch);
                        break;
                case 'I':
                case 'i':
```

```
                        printf("\n %c is VOWEL", ch);
                        break;
                case 'O':
                case 'o':
                        printf("\n %c is VOWEL", ch);
                        break;
                case 'U':
                case 'u':
                        printf("\n %c is VOWEL", ch);
                        break;
                default: printf("\n %c is not a vowel", ch);
        }
        return 0;
}
```

Output

```
Enter any character : j
j is not a vowel
```

Note that there is no `break` statement after case `A`, so if the character `A` is entered then control will execute the statements given in case `a`.

1.9.2 Iterative Statements

Iterative statements are used to repeat the execution of a sequence of statements until the specified expression becomes false. C supports three types of iterative statements also known as looping statements. They are

- `while` loop
- `do-while` loop
- `for` loop

In this section, we will discuss all these statements.

`while` *Loop*

The `while` loop provides a mechanism to repeat one or more statements while a particular condition is true. Figure 1.6 shows the syntax and general form of a `while` loop.

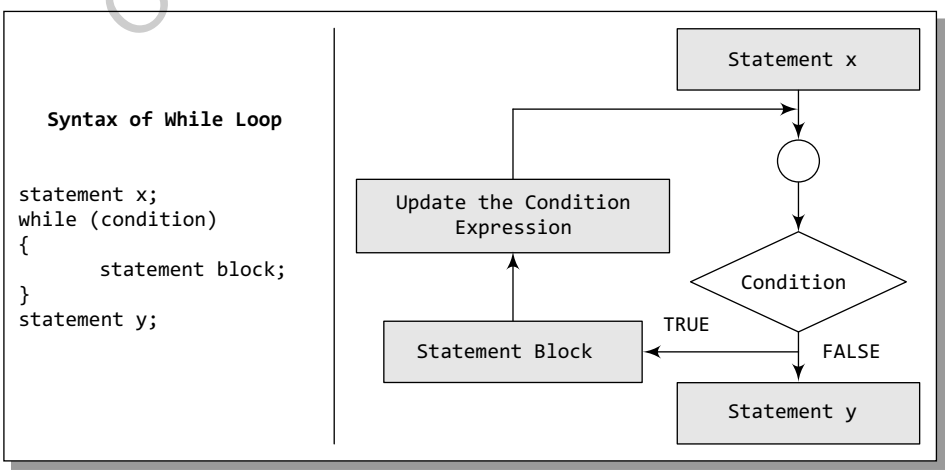


Figure 1.6 `While` loop construct

Note that in the `while` loop, the condition is tested before any of the statements in the statement block is executed. If the condition is true, only then the statements will be executed, otherwise if the condition is false, the control will jump to statement `y`, that is the immediate statement outside the `while` loop block.

In the flow diagram of Fig. 1.6, it is clear that we need to constantly update the condition of the `while` loop. It is this condition which determines when the loop will end. The `while` loop will execute as long as the condition is true. Note that if the condition is never updated and the condition never becomes false, then the computer will run into an infinite loop which is never desirable. For example, the following code prints the first 10 numbers using a `while` loop.

```
#include <stdio.h>
int main()
{
      int i = 1;
      while(i<=10)
      {
            printf("\n %d", i);
            i = i + 1;      // condition updated
      }
      return 0;
}
```

Note that initially `i = 1` and is less than 10, i.e., the condition is true, so in the `while` loop the value of `i` is printed and its value is incremented by 1. When `i=11`, the condition becomes false and the loop ends.

PROGRAMMING EXAMPLE

5. Write a program to calculate the sum of numbers from `m` to `n`.

```
#include <stdio.h>
int main()
{
        int n, m, i, sum =0;
        printf("\n Enter the value of m : ");
        scanf("%d", &m);
        i=m;
        printf("\n Enter the value of n : ");
        scanf("%d", &n);
        while(i<=n)
        {
                sum = sum + i;
                i = i + 1;
        }
        printf("\n The sum of numbers from %d to %d = %d", m, n, sum);
        return 0;
}
```

Output

```
Enter the value of m : 2
Enter the value of n : 10
The sum of numbers from 2 to 10  = 54
```

`do-while` *Loop*

The `do-while` loop is similar to the `while` loop. The only difference is that in a `do-while` loop, the test condition is tested at the end of the loop. As the test condition is evaluated at the end, this means that the body of the loop gets executed at least one time (even if the condition is false). Figure 1.7 shows the syntax and the general form of a `do-while` loop.

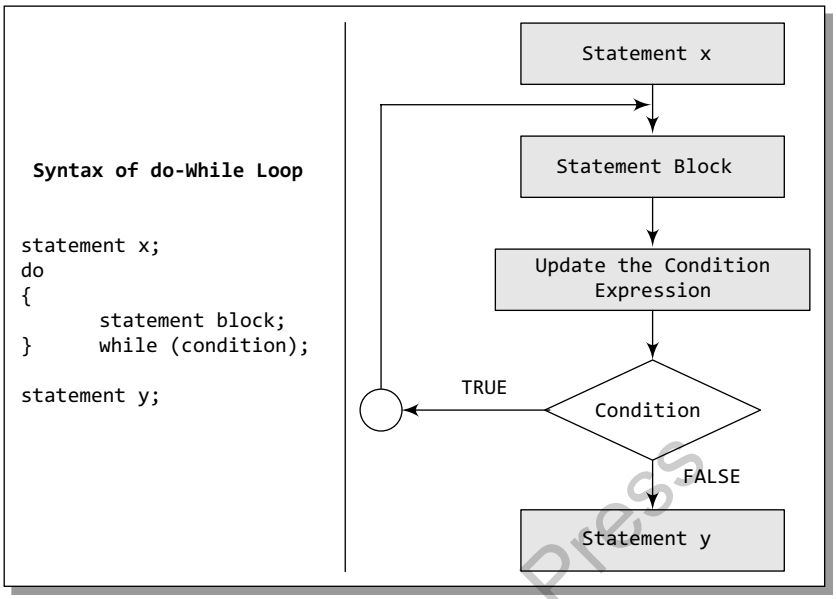


Figure 1.7 `Do-while` construct

Note that the test condition is enclosed in parentheses and followed by a semi-colon. The statements in the statement block are enclosed within curly brackets. The curly brackets are optional if there is only one statement in the body of the `do-while` loop.

The `do-while` loop continues to execute while the condition is true and when the condition becomes false, the control jumps to the statement following the `do-while` loop.

The major disadvantage of using a `do-while` loop is that it always executes at least once, so even if the user enters some invalid data, the loop will execute. However, `do-while` loops are widely used to print a list of options for menu-driven programs. For example, consider the following code.

```
#include <stdio.h>
int main()
{
      int i = 1;
      do
      {
            printf("\n %d", i);
            i = i + 1;
      } while(i<=10);
      return 0;
}
```

What do you think will be the output? Yes, the code will print numbers from 1 to 10.

Programming Example

6. Write a program to calculate the average of first *n* numbers.

```
#include <stdio.h>
int main()
{
        int n, i = 0, sum =0;
        float avg = 0.0;
```

```
            printf("\n Enter the value of n : ");
            scanf("%d", &n);
            do
            {
                    sum = sum + i;
                    i = i + 1;
            } while(i<=n);
            avg = (float)sum/n;
            printf("\n The sum of first %d numbers = %d",n, sum);
            printf("\n The average of first %d numbers = %.2f", n, avg);
            return 0;
    }
```

Output

```
Enter the value of n : 20
The sum of first 20 numbers = 210
The average of first 20 numbers = 10.05
```

`for` *Loop*

Like the `while` and `do-while` loops, the `for` loop provides a mechanism to repeat a task till a particular condition is true. The synax and general form of a `for` loop is given in Fig. 1.8.

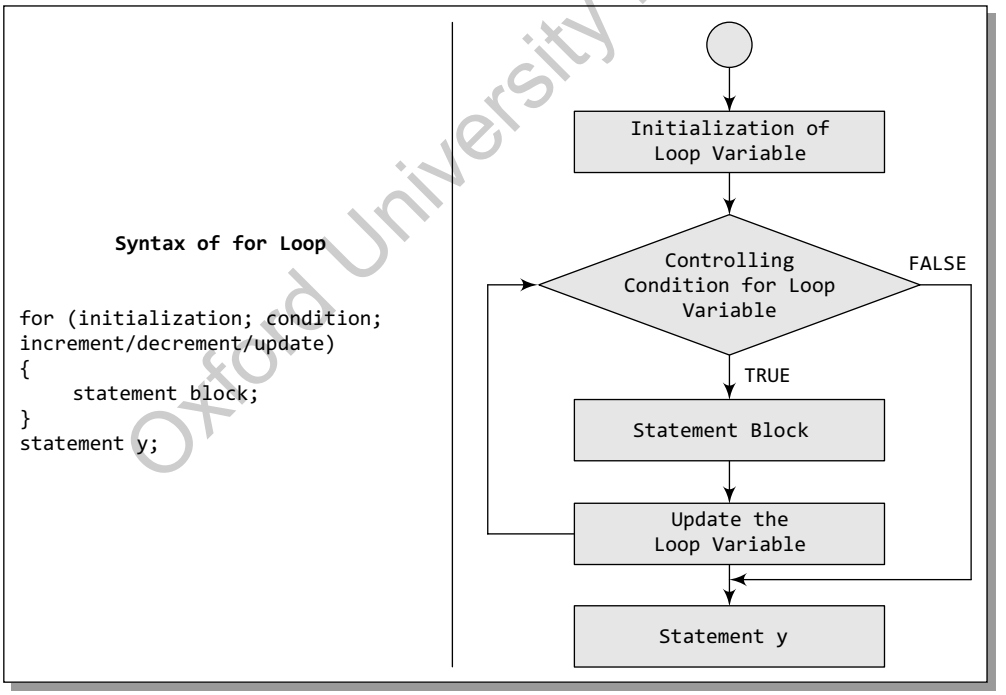


Figure 1.8 `for` loop construct

When a `for` loop is used, the loop variable is initialized only once. With every iteration, the value of the loop variable is updated and the condition is checked. If the condition is true, the statement block of the loop is executed, else the statements comprising the statement block of the `for` loop are skipped and the control jumps to the statement following the `for` loop body.

In the syntax of the `for` loop, initialization of the loop variable allows the programmer to give it a value. Second, the condition specifies that while the conditional expression is true, the loop

should continue to repeat itself. Every iteration of the loop must make the condition to exit the loop approachable. So, with every iteration, the loop variable must be updated. Updating the loop variable may include incrementing the loop variable, decrementing the loop variable or setting it to some other value like, `i +=2`, where `i` is the loop variable.

Note that every section of the `for` loop is separated from the other with a semi-colon. It is possible that one of the sections may be empty, though the semi-colons still have to be there. However, if the condition is empty, it is evaluated as true and the loop will repeat until something else stops it.

The `for` loop is widely used to execute a single or a group of statements for a limited number of times. The following code shows how to print the first *n* numbers using a `for` loop.

```
#include <stdio.h>
int main()
{
      int i, n;
      printf("\n Enter the value of n :");
      scanf("%d", &n);
      for(i=1;i<=n;i++)
            printf("\n %d", i);
      return 0;
}
```

In the code, `i` is the loop variable. Initially, it is initialized with 1. Suppose the user enters 10 as the value of `n`. Then the condition is checked, since the condition is true as `i` is less than `n`, the statement in the `for` loop is executed and the value of `i` is printed. After every iteration, the value of `i` is incremented. When `i` exceeds the value of `n`, the control jumps to the `return 0` statement.

Programming Example

7. Write a program to determine whether a given number is a prime or a composite number.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        int flag = 0, i, num;
        clrscr();
        printf("\n Enter any number : ");
        scanf("%d", &num);
        for(i=2; i<num/2;i++)
        {
                if(num%i == 0)
                {
                        flag =1;
                        break;
                }
        }
if(flag == 1)
        printf("\n %d is a composite number", num);
else
        printf("\n %d is a prime number", num);
return 0;
}
```

Output

```
Enter any number : 37
37 is a prime number
```

1.9.3 Break and Continue Statements

`break` *Statement*

In C, the `break` statement is used to terminate the execution of the nearest enclosing loop in which it appears. We have already seen its use in the `switch` statement. The `break` statement is widely used with `for`, `while`, and `do-while` loops. When the compiler encounters a `break` statement, the control passes to the statement that follows the loop in which the `break` statement appears. Its syntax is quite simple, just type keyword `break` followed by a semi-colon.

```
break;
```

The example given below shows the manner in which `break` statement is used to terminate the loop in which it is embedded.

```
#include <stdio.h>
int main()
{
      int i = 0;
      while(i<=10)
      {
            if (i==5)
                  break;
            printf("\t %d", i);
            i = i + 1;
      }
      return 0;
}
```

Output

```
0  1  2  3  4
```

As soon as `i` becomes equal to 5, the `break` statement is executed and the control jumps to the statement following the `while` loop.

Hence, the `break` statement is used to exit a loop from any point within its body, bypassing its normal termination expression.

`continue` *Statement*

Like the `break` statement, the `continue` statement can only appear in the body of a loop. When the compiler encounters a `continue` statement, then the rest of the statements in the loop are skipped and the control is unconditionally transferred to the loop-continuation portion of the nearest enclosing loop. Its syntax is quite simple, just type keyword `continue` followed by a semi-colon.

```
continue;
```

Again like the `break` statement, the `continue` statement cannot be used without an enclosing `for`, `while`, or `do-while` loop. When the `continue` statement is encountered in the `while` loop and in the `do-while` loop, the control is transferred to the code that tests the controlling expression. However, if placed within a `for` loop, the `continue` statement causes a branch to the code that updates the loop variable. For example, consider the following code:

```
#include <stdio.h>
int main()
{
      int i;
      for(i=0; i<= 10; i++)
      {
            if (i==5)
```

```
                    continue;
                printf("\t %d", i);
            }
            return 0;
        }
```

Output

```
0  1  2  3  4  6  7  8  9  10
```

Note that the code is meant to print numbers from 0 to 10. But as soon as `i` becomes equal to 5, the `continue` statement is encountered, so the `printf()` statement is skipped and the control passes to the expression that increments the value of `i`.

Hence, we conclude that 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 itself and the test condition of the loop. It is generally used to restart a statement sequence when an error occurs.

1.10 FUNCTIONS

C enables its programmers to break up a program into segments commonly known as *functions*, each of which can be written more or less independently of the others. Every function in the program is supposed to perform a well-defined task. Therefore, the program code of one function is completely insulated from the other functions.

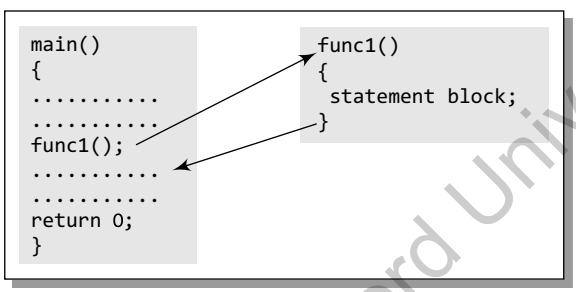


Figure 1.9 `main()` calls `func1()`

Every function interfaces to the outside world in terms of how information is transferred to it and how results generated by the function are transmitted back from it. This interface is basically specified by the function name. For example, look at Fig. 1.9 which explains how the `main()` function calls another function to perform a well-defined task.

In the figure, we can see that `main()` calls a function named `func1()`. Therefore, `main()` is known as the *calling function* and `func1()` is known as the *called function*. The moment the compiler encounters a function call, the control jumps to the statements that are a part of the called function. After the called function is executed, the control is returned to the calling program.

The `main()` function can call as many functions as it wants and as many times as it wants. For example, a function call placed within a `for` loop, `while` loop, or `do-while` loop may call the same function multiple times till the condition holds true.

Not only `main()`, any function can call any other function. For example, look at Fig. 1.10 which shows one function calling another, and the other function in turn calling some other function.

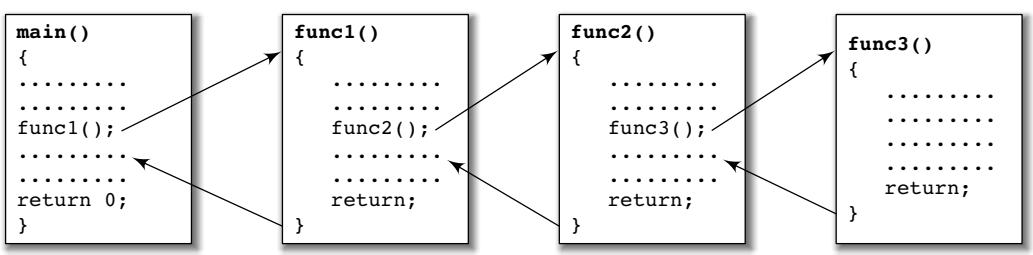


Figure 1.10 Function calling another function

1.10.1 Why are Functions Needed?

Let us analyse the reasons why segmenting a program into manageable chunks is an important aspect of programming.

- Dividing the program into separate well-defined functions facilitates each function to be written and tested separately. This simplifies the process of getting the total program to work.
- Understanding, coding, and testing multiple separate functions is easier than doing the same for one big function.
- If a big program has to be developed without using any function other than `main()`, then there will be countless lines in the `main()` function and maintaining that program will be a difficult task.
- All the libraries in C contain a set of functions that the programmers are free to use in their programs. These functions have been pre-written and pre-tested, so the programmers can use them without worrying about their code details. This speeds up program development, by allowing the programmer to concentrate only on the code that he has to write.
- Like C libraries, programmers can also write their own functions and use them from different points in the main program or any other program that needs its functionalities.
- When a big program is broken into comparatively smaller functions, then different programmers working on that project can divide the workload by writing different functions.

1.10.2 Using Functions

A function can be compared to a *black box* that takes in inputs, processes it, and then outputs the result. However, we may also have a function that does not take any inputs at all, or a function that does not return any value at all. While using functions, we will be using the following terminologies:

- A function *f* that uses another function *g* is known as the *calling function,* and *g* is known as the *called function*.
- The inputs that a function takes are known as *arguments.*
- When a called function returns some result back to the calling function, it is said to *return* that result.
- The calling function may or may not pass *parameters* to the called function. If the called function accepts arguments, the calling function will pass parameters, else not.
- *Function declaration* is a declaration statement that identifies a function's name, a list of arguments that it accepts, and the type of data it returns.
- *Function definition* consists of a function header that identifies the function, followed by the body of the function containing the executable code for that function.

Function Declaration

Before using a function, the compiler must know the number of parameters and the type of parameters that the function expects to receive and the data type of value that it will return to the calling program. Placing the function declaration statement prior to its use enables the compiler to make a check on the arguments used while calling that function.

The general format for declaring a function that accepts arguments and returns a value as result can be given as:

```
return_data_type function_name(data_type variable1, data_type variable2,..);
```

Here, **`function_name`** is a valid name for the function. Naming a function follows the same rules that are followed while naming variables. A function should have a meaningful name that must specify the task that the function will perform.

`return_data_type` specifies the data type of the value that will be returned to the calling function as a result of the processing performed by the called function.

`(data_type variable1, data_type variable2, ...)` is a list of variables of specified data types. These variables are passed from the calling function to the called function. They are also known as arguments or parameters that the called function accepts to perform its task.

Note A function having void as its return type cannot return any value. Similarly, a function having void as its parameter list cannot accept any value.

Function Definition

When a function is defined, space is allocated for that function in the memory. A function definition comprises of two parts:

- Function header
- Function body

The syntax of a function definition can be given as:

```
return_data_type function_name(data_type variable1, data_type variable2,..)
{
      .............
      statements
      .............
      return(variable);
}
```

Note that the number of arguments and the order of arguments in the function header must be the same as that given in the function declaration statement.

While **`return_data_type function_name(data_type variable1, data_type variable2,...)`** is known as the function header, the rest of the portion comprising of program statements within the curly brackets { } is the function body which contains the code to perform the specific task.

Note that the function header is same as the function declaration. The only difference between the two is that a function header is not followed by a semi-colon.

Function Call

The function call statement invokes the function. When a function is invoked, the compiler jumps to the called function to execute the statements that are a part of that function. Once the called function is executed, the program control passes back to the calling function. A function call statement has the following syntax:

```
function_name(variable1, variable2, ...);
```

The following points are to be noted while calling a function:

- Function name and the number and the type of arguments in the function call must be same as that given in the function declaration and the function header of the function definition.
- Names (and not the types) of variables in function declaration, function call, and header of function definition may vary.
- Arguments may be passed in the form of expressions to the called function. In such a case, arguments are first evaluated and converted to the type of formal parameter and then the body of the function gets executed.
- If the return type of the function is not `void`, then the value returned by the called function may be assigned to some variable as given below.

```
variable_name = function_name(variable1, variable2, ...);
```

PROGRAMMING EXAMPLE

8. Write a program to find whether a number is even or odd using functions.

```
#include <stdio.h>
int evenodd(int);   //FUNCTION DECLARATION
int main()
{
        int num, flag;
        printf("\n Enter the number : ");
        scanf("%d", &num);
        flag = evenodd(num);  //FUNCTION CALL
        if (flag == 1)
                printf("\n %d is EVEN", num);
        else
                printf("\n %d is ODD", num);
        return 0;
}
int evenodd(int a) // FUNCTION HEADER
{
                // FUNCTION BODY
        if(a%2 == 0)
                return 1;
        else
                retun 0;
}
```

Output

```
Enter the number : 78
78 is EVEN
```

1.10.3 Passing Parameters to Functions

There are two ways in which arguments or parameters can be passed to the called function.

Call by value The values of the variables are passed by the calling function to the called function.

Call by reference The addresses of the variables are passed by the calling function to the called function.

Call by Value

In this method, the called function creates new variables to store the value of the arguments passed to it. Therefore, the called function uses a copy of the actual arguments to perform its intended task.

If the called function is supposed to modify the value of the parameters passed to it, then the change will be reflected only in the called function. In the calling function, no change will be made to the value of the variables. This is because all the changes are made to the copy of the variables and not to the actual variables. To understand this concept, consider the code given below. The function `add()` accepts an integer variable `num` and adds 10 to it. In the calling function, the value of `num = 2`. In `add()`, the value of `num` is modified to 12 but in the calling function, the change is not reflected.

```
#include <stdio.h>
void add(int n);
int main()
{
      int num = 2;
      printf("\n The value of num before calling the function = %d", num);
      add(num);
```

```
        printf("\n The value of num after calling the function = %d", num);
        return 0;
}
void add(int n)
{
        n = n + 10;
        printf("\n The value of num in the called function = %d", n);
}
```

Output

```
The value of num before calling the function = 2
The value of num in the called function = 12
The value of num after calling the function = 2
```

Following are the points to remember while passing arguments to a function using the call-by-value method:

- When arguments are passed by value, the called function creates new variables of the same data type as the arguments passed to it.
- The values of the arguments passed by the calling function are copied into the newly created variables.
- Values of the variables in the calling functions remain unaffected when the arguments are passed using the call-by-value technique.

Pros and cons

The biggest advantage of using the call-by-value technique is that arguments can be passed as variables, literals, or expressions, while its main drawback is that copying data consumes additional storage space. In addition, it can take a lot of time to copy, thereby resulting in performance penalty, especially if the function is called many times.

Call by Reference

When the calling function passes arguments to the called function using the call-by-value method, the only way to return the modified value of the argument to the caller is explicitly using the return statement. A better option is to pass arguments using the call-by-reference technique. In this method, we declare the function parameters as references rather than normal variables. When this is done, any changes made by the function to the arguments it received are also visible in the calling function.

To indicate that an argument is passed using call by reference, an asterisk (*) is placed after the type in the parameter list.

Hence, in the call-by-reference method, a function receives an implicit reference to the argument, rather than a copy of its value. Therefore, the function can modify the value of the variable and that change will be reflected in the calling function as well. The following code illustrates this concept.

```
#include <stdio.h>
void add(int *);
int main()
{
        int num = 2;
        printf("\n The value of num before calling the function = %d", num);
        add(&num);
        printf("\n The value of num after calling the function = %d", num);
        return 0;
}
void add(int *n)
```

```
{
      *n = *n + 10;
      printf("\n The value of num in the called function = %d", *n);
}
```

Output

```
The value of num before calling the function = 2
The value of num in the called function = 12
The value of num after calling the function = 12
```

Advantages

The advantages of using the call-by-reference technique of passing arguments include:

- Since arguments are not copied into the new variables, it provides greater time and space-efficiency.
- The function can change the value of the argument and the change is reflected in the calling function.
- A function can return only one value. In case we need to return multiple values, we can pass those arguments by reference, so that the modified values are visible in the calling function.

Disadvantages

However, the drawback of using this technique is that if inadvertent changes are caused to variables in called function then these changes would be reflected in calling function as original values would have been overwritten.

Consider the code given below which swaps the value of two integers. Note the value of integers in the calling function and called function.

```
//This function swaps the value of two variables
#include <stdio.h>
void swap_call_val(int, int);
void swap_call_ref(int *, int *);
int main()
{
      int a=1, b=2, c=3, d=4;
      printf("\n In main(), a = %d and b = %d", a, b);
      swap_call_val(a, b);
      printf("\n In main(), a = %d and b = %d", a, b);
      printf("\n\n In main(), c = %d and d = %d", c, d);
      swap_call_ref(&c, &d);
      printf("\n In main(), c = %d and d = %d", c, d);
      return 0;
}
void swap_call_val(int a, int b)
{
      int temp;
      temp = a;
      a = b;
      b = temp;
      printf("\n In function (Call By Value Method) - a = %d and b = %d", a, b);
}
void swap_call_ref(int *c, int *d)
{
      int temp;
      temp = *c;
      *c = *d;
      *d = temp;
      printf("\n In function (Call By Reference Method) - c = %d and d = %d", *c, *d);
}
```

Output

```
In main(), a = 1 and b = 2
In function (Call By Value Method) - a = 2 and b = 1
In main(), a = 1 and b = 2
In main(), c = 3 and d = 4
In function (Call By Reference Method) - c = 4 and d = 3
In main(), c = 4 and d = 3
```

1.11 POINTERS

Every variable in C has a name and a value associated with it. When a variable is declared, a specific block of memory within the computer is allocated to hold the value of that variable. The size of the allocated block depends on the data type.

Consider the following statement.

```
int x = 10;
```

When this statement executes, the compiler sets aside 2 bytes of memory to hold the value 10. It also sets up a symbol table in which it adds the symbol `x` and the relative address in the memory where those 2 bytes were set aside.

(Note the size of integer may vary from one system to another. On 32 bit systems, integer variable is allocated 4 bytes while on 16 bit systems it is allocated 2 bytes.)

Thus, every variable in C has a value and also a memory location (commonly known as *address*) associated with it. We will use terms `rvalue` and `lvalue` for the value and the address of the variable, respectively.

The `rvalue` appears on the right side of the assignment statement (10 in the above statement) and cannot be used on the left side of the assignment statement. Therefore, writing `10 = k;` is illegal. If we write,

```
int x, y;
x = 10;
y = x;
```

then, we have two integer variables `x` and `y`. The compiler reserves memory for the integer variable `x` and stores the `rvalue` 10 in it. When we say `y = x`, then `x` is interpreted as its `rvalue` since it is on the right hand side of the assignment operator `=`. Therefore, here `x` refers to the value stored at the memory location set aside for `x`, in this case 10. After this statement is executed, the `rvalue` of `y` is also 10.

You must be wondering why we are discussing addresses and `lvalues`. Actually pointers are nothing but memory addresses. A pointer is a variable that contains the memory location of another variable. Therefore, a pointer is a variable that represents the location of a data item, such as a variable or an array element. Pointers are frequently used in C, as they have a number of useful applications. These applications include:

- Pointers are used to pass information back and forth between functions.
- Pointers enable the programmers to return multiple data items from a function via function arguments.
- Pointers provide an alternate way to access the individual elements of an array.
- Pointers are used to pass arrays and strings as function arguments. We will discuss this in subsequent chapters.
- Pointers are used to create complex data structures, such as trees, linked lists, linked stacks, linked queues, and graphs.

- Pointers are used for the dynamic memory allocation of a variable (refer Appendix A on memory allocation in C programs).

1.11.1 Declaring Pointer Variables

The general syntax of declaring pointer variables can be given as below.

```
data_type *ptr_name;
```

Here, `data_type` is the data type of the value that the pointer will point to. For example,

```
int *pnum;
char *pch;
float *pfnum;
```

In each of the above statements, a pointer variable is declared to point to a variable of the specified data type. Although all these pointers (`pnum`, `pch`, and `pfnum`) point to different data types, they will occupy the same amount of space in the memory. But how much space they will occupy will depend on the platform where the code is going to run. Now let us declare an integer pointer variable and start using it in our program code.

```
int x= 10;
int *ptr;
ptr = &x;
```

In the above statement, `ptr` is the name of the pointer variable. The `*` informs the compiler that `ptr` is a pointer variable and the `int` specifies that it will store the address of an integer variable. An integer pointer variable, therefore, 'points to' an integer variable. In the last statement, `ptr` is assigned the address of `x`. The `&` operator retrieves the `lvalue` (address) of `x`, and copies that to the contents of the pointer `ptr`. Consider the memory cells given in Fig. 1.11.

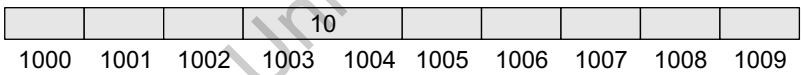


Figure 1.11 Memory representation

Now, since `x` is an integer variable, it will be allocated 2 bytes. Assuming that the compiler assigns it memory locations 1003 and 1004, the address of `x` (written as `&x`) is equal to 1003, that is the starting address of `x` in the memory. When we write, `ptr = &x`, then `ptr = 1003`.

We can 'dereference' a pointer, *i.e.*, we can refer to the value of the variable to which it points by using the unary `*` operator as in `*ptr`. That is, `*ptr = 10`, since 10 is the value of `x`. Look at the following code which shows the use of a pointer variable:

```
#include <stdio.h>
int main()
{
      int num, *pnum;
      pnum = &num;
      printf("\n Enter the number : ");
      scanf("%d", &num);
      printf("\n The number that was entered is : %d", *pnum);
      return 0;
}
```

Output

```
Enter the number : 10
The number that was entered is : 10
```

What will be the value of `*(&num)`? It is equivalent to simply writing `num`.

1.11.2 Pointer Expressions and Pointer Arithmetic

Like other variables, pointer variables can also be used in expressions. For example, if `ptr1` and `ptr2` are pointers, then the following statements are valid:

```
int num1 = 2, num2 = 3, sum = 0, mul = 0, div = 1;
int *ptr1, *ptr2;
ptr1 = &num1;
ptr2 = &num2;
sum = *ptr1 + *ptr2;
mul = sum * (*ptr1);
*ptr2 += 1;
div = 9 + (*ptr1)/(*ptr2) - 30;
```

In C, the programmer may add integers to or subtract integers from pointers as well as subtract one pointer from the other. We can also use shorthand operators with the pointer variables as we use them with other variables.

C also allows comparing pointers by using relational operators in the expressions. For example, `p1>p2`, `p1==p2` and `p1!=p2` are all valid in C.

Postfix unary increment `(++)` and decrement `(--)` operators have greater precedence than the dereference operator `(*)`. Therefore, the expression `*ptr++` is equivalent to `*(ptr++)`, as `++` has greater operator precedence than `*`. Thus, the expression will increase the value of `ptr` so that it now points to the next memory location. This means that the statement `*ptr++` does not do the intended task. Therefore, to increment the value of the variable whose address is stored in `ptr`, you should write `(*ptr)++`.

1.11.3 Null Pointers

So far, we have studied that a pointer variable is a pointer to a variable of some data type. However, in some cases, we may prefer to have a *null pointer* which is a special pointer value and does not point to any value. This means that a `null` pointer does not point to any valid memory address.

To declare a null pointer, you may use the predefined constant `NULL` which is defined in several standard header files including `<stdio.h>`, `<stdlib.h>`, and `<string.h>`. After including any of these files in your program, you can write

```
int *ptr = NULL;
```

You can always check whether a given pointer variable stores the address of some variable or contains `NULL` by writing,

```
if (ptr == NULL)
{
      Statement block;
}
```

You may also initialize a pointer as a null pointer by using the constant 0

```
int *ptr,
ptr = 0;
```

This is a valid statement in C as `NULL` is a preprocessor macro, which typically has the value or replacement text 0. However, to avoid ambiguity, it is always better to use `NULL` to declare a null pointer. A function that returns pointer values can return a null pointer when it is unable to perform its task.

1.11.4 Generic Pointers

A generic pointer is a pointer variable that has *void* as its data type. The *void pointer*, or the generic pointer, is a special type of pointer that can point to variables of any data type. It is declared like a normal pointer variable but using the `void` keyword as the pointer's data type. For example,

```
void *ptr;
```

In C, since you cannot have a variable of type `void`, the void pointer will therefore not point to any data and, thus, cannot be dereferenced. You need to cast a void pointer to another kind of pointer before using it.

Generic pointers are often used when you want a pointer to point to data of different types at different times. For example, take a look at the following code.

```
#include <stdio.h>
int main()
{
      int x=10;
      char ch = 'A';
      void *gp;
      gp = &x;
      printf("\n Generic pointer points to the integer value = %d", *(int*)gp);
      gp = &ch;
      printf("\n Generic pointer now points to the character= %c", *(char*)gp);
      return 0;
}
```

Output

```
Generic pointer points to the integer value = 10
Generic pointer now points to the character = A
```

It is always recommended to avoid using void pointers unless absolutely necessary, as they effectively allow you to avoid type checking.

PROGRAMMING EXAMPLE

9. Write a program to add two integers using pointers and functions.

```
#include <stdio.h>
void sum (int*, int*, int*);
int main()
{
        int num1, num2, total;
        printf("\n Enter the first number : ");
        scanf("%d", &num1);
        printf("\n Enter the second number : ");
        scanf("%d", &num2);
        sum(&num1, &num2, &total);
        printf("\n Total = %d", total);
        return 0;
}
void sum (int *a, int *b, int *t)
{
        *t = *a + *b;
}
```

Output

```
Enter the first number : 23
Enter the second number : 34
Total = 57
```

1.11.5 Pointer to Pointers

In C, you can also use pointers that point to pointers. The pointers in turn point to data or even to other pointers. To declare pointers to pointers, just add an asterisk * for each level of reference.

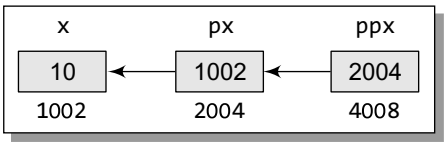


Figure 1.12 Pointer to pointer

For example, consider the following code:

```
int x=10;
int *px, **ppx;
px = &x;
ppx = &px;
```

Let us assume, the memory locations of these variables are as shown in Fig. 1.12.

Now if we write,

```
printf("\n %d", **ppx);
```

Then, it would print 10, the value of `x`.

1.11.6 Drawbacks of Pointers

Although pointers are very useful in C, they are not free from limitations. If used incorrectly, pointers can lead to bugs that are difficult to unearth. For example, if you use a pointer to read a memory location but that pointer is pointing to an incorrect location, then you may end up reading a wrong value. An erroneous input always leads to an erroneous output. Thus however efficient your program code may be, the output will always be disastrous. Same is the case when writing a value to a particular memory location.

Let us try to find some common errors when using pointers.

```
int x, *px;
x=10;
*px = 20;
```

Error: Un-initialized pointer. `px` is pointing to an unknown memory location. Hence it will overwrite that location's contents and store 20 in it.

```
int x, *px;
x=10;
px = x;
```

Error: It should be `px = &x;`

```
int x=10, y=20, *px, *py;
px = &x, py = &y;
if(px<py)
printf("\n x is less than y");
else
printf("\n y is less than x");
```

Error: It should be `if(*px< *py)`

Points to Remember

- C was developed in the early 1970s by Dennis Ritchie at Bell Laboratories.
- Every word in a C program is either an identifier or a keyword. Identifiers are the names given to program elements such as variables and functions. Keywords are reserved words which cannot be used as identifiers.
- C provides four basic data types: `char`, `int`, `float`, and `double`.
- A variable is defined as a meaningful name given to a data storage location in computer memory.
- Standard library function `scanf()` is used to input data in a specified format.`printf()`function is used to output data of different types in a specified format.
- C supports different types of operators which can be classified into following categories: arithmetic, relational, equality, logical, unary, conditional, bitwise, assignment, comma, and sizeof operators.

- Modulus operator (%) can only be applied on integer operands, and not on float or double operands.
- Equality operators have lower precedence than relational operators.
- Like arithmetic expressions, logical expressions are evaluated from left to right.
- Both x++ and ++x increment the value of x, but in the former case, the value of x is returned before it is incremented. Whereas in the latter case, the value of x is returned after it is incremented.
- Conditional operator is also known as ternary operator as it takes three operands.
- Bitwise NOT or complement produces one's complement of a given binary number.
- Among all the operators, comma operator has the lowest precedence.
- `sizeof` is a unary operator used to calculate the size of data types. This operator can be applied to all data types.
- While type conversion is done implicitly, typecasting has to be done explicitly by the programmer. Typecasting is done when the value of one data type has to be converted into the value of another data type.
- C supports three types of control statements: decision control statements, iterative statements, and jump statements.
- In a `switch` statement, if the value of the variable does not match with any of the values of case statements, then default case is executed.
- Iterative statements are used to repeat the execution of a list of statements until the specified expression becomes false.
- The `break` statement is used to terminate the execution of the nearest enclosing loop in which it appears.
- When the compiler encounters a `continue` statement, then the rest of the statements in the loop are skipped and the control is unconditionally transferred to the loop-continuation portion of the nearest enclosing loop.
- A C program contains one or more functions, where each function is defined as a group of statements that perform a specific task.
- Every C program contains a `main()` function which is the starting point of the program. It is the function that is called by the operating system when the user runs the program.
- Function declaration statement identifies a function's name and the list of arguments that it accepts and the type of data it returns.
- Function definition, on the other hand, consists of a function header that identifies the function, followed by the body of the function containing the executable code for that function. When a function is defined, space is allocated for that function in the memory.
- The moment the compiler encounters a function call, the control jumps to the statements that are a part of the called function. After the called function is executed, the control is returned back to the calling function.
- Placing the function declaration statement prior to its use enables the compiler to make a check on the arguments used while calling that function.
- A function having `void` as its return type cannot return any value. Similarly, a function having `void` as its parameter list cannot accept any value.
- Call by value method passes values of the variables to the called function. Therefore, the called function uses a copy of the actual arguments to perform its intended task. This method is used when the function does not need to modify the values of the original variables in the calling function.
- In call by reference method, addresses of the variables are passed by the calling function to the called function. Hence, in this method, a function receives an implicit reference to the argument, rather than a copy of its value. This allows the function to modify the value of the variable and that change is reflected in the calling function as well.
- A pointer is a variable that contains the memory address of another variable.
- The & operator retrieves the address of the variable.
- We can 'dereference' a pointer, i.e., refer to the value of the variable to which it points by using unary * operator.
- Null pointer is a special pointer variable that does not point to any variable. This means that a null pointer does not point to any valid memory address. To declare a null pointer we may use the predefined constant NULL.
- A generic pointer is pointer variable that has `void` as its data type. The generic pointer can point to variables of any data type.
- To declare pointer to pointers, we need to add an asterisk (*) for each level of reference.

EXERCISES

Review Questions

1. Discuss the structure of a C program.
2. Differentiate between declaration and definition.
3. How is memory reserved using a declaration statement?
4. What do you understand by identifiers and keywords?
5. Explain the terms variables and constants. How many types of variables are supported by C?
6. What does the data type of a variable signify?
7. Write a short note on basic data types that the C language supports.
8. Why do we include <stdio.h> in our programs?
9. What are header files? Explain their significance.
10. Write short notes on printf and scanf functions.
11. Write a short note on operators available in C language.
12. Draw the operator precedence chart.
13. Differentiate between typecasting and type conversion.
14. What are decision control statements? Explain in detail.
15. Write a short note on the iterative statements that C language supports.
16. When will you prefer to work with a switch statement?
17. Define function. Why are they needed?
18. Differentiate between function declaration and function definition.
19. Why is function declaration statement placed prior to function definition?
20. Explain the concept of making function calls.
21. Differentiate between call by value and call by reference using suitable examples.
22. Write a short note on pointers.
23. Explain the difference between a null pointer and a void pointer.
24. How are generic pointers different from other pointer variables?
25. Write a short note on pointers to pointers.

Programming Exercises

1. Write a program to read 10 integers. Display these numbers by printing three numbers in a line separated by commas.
2. Write a program to print the count of even numbers between 1–200. Also print their sum.
3. Write a program to count the number of vowels in a text.
4. Write a program to read the address of a user. Display the result by breaking it in multiple lines.
5. Write a program to read two floating point numbers. Add these numbers and assign the result to an integer. Finally, display the value of all the three variables.
6. Write a program to read a floating point number. Display the rightmost digit of the integral part of the number.
7. Write a program to calculate simple interest and compound interest.
8. Write a program to calculate salary of an employee given his basic pay (to be entered by the user), HRA = 10% of the basic pay, TA = 5% of basic pay. Define HRA and TA as constants and use them to calculate the salary of the employee.
9. Write a program to prepare a grocery bill. Enter the name of the items purchased, quantity in which it is purchased, and its price per unit. Then display the bill in the following format:

```
************ B I L L ************
  Item  Quantity  Price  Amount
_____________________________________

_____________________________________
     Total Amount to be paid
_____________________________________
```

10. Write a C program using printf statement to print BYE in the following format:

```
BBB      Y     Y      EEEE
B  B      Y    Y      E
BBB         Y         EEEE
B  B        Y
```

11. Write a program to read an integer. Display the value of that integer in decimal, octal, and hexadecimal notation.
12. Write a program that prints a floating point value in exponential format with the following specifications:
    (a) correct to two decimal places;
    (b) correct to four decimal places; and

(c) correct to eight decimal places.

13. Write a program to find the smallest of three integers using functions.

14. Write a program to calculate area of a triangle using function.

15. Write a program to find whether a number is divisible by two or not using functions.

16. Write a program to print 'Programming in C is Fun' using pointers.

17. Write a program to read a character and print it. Also print its ASCII value. If the character is in lower case, print it in upper case and vice versa. Repeat the process until a '*' is entered.

18. Write a program to add three floating point numbers. The result should contain only two digits after the decimal.

19. Write a program to take input from the user and then check whether it is a number or a character. If it is a character, determine whether it is in upper case or lower case. Also print its ASCII value.

20. Write a program to display sum and average of numbers from `1` to `n`. Use `for` loop.

21. Write a program to print all odd numbers from `m` to `n`.

22. Write a program to print all prime numbers from `m` to `n`.

23. Write a program to read numbers until –1 is entered and display whether it is an Armstrong number or not.

24. Write a program to add two floating point numbers using pointers and functions.

25. Write a program to calculate area of a triangle using pointers.

Multiple-choice Questions

1. The operator which compares two values is
(a) Assignment (b) Relational
(c) Unary (d) Equality

2. Ternary operator operates on how many operands?
(a) 1 (b) 2
(c) 3 (d) 4

3. Which operator produces the one's complement of the given binary value?
(a) Logical AND (b) Bitwise AND
(c) Logical OR (d) Bitwise NOT

4. Which operator has the lowest precedence?
(a) Sizeof (b) Unary
(c) Assignment (d) Comma

5. Which of the following is the conversion character associated with short integer?
(a) `%c` (b) `%h`
(c) `%e` (d) `%f`

6. Which of the following is not a character constant?
(a) `'A'` (b) "A"
(c) `' '` (d) `'*'`

7. Which of the following is a valid variable name?
(a) Initial.Name (b) A+B
(c) $amt (d) Floats

8. Which operator cannot be used with floating point numbers?
(a) `+` (b) `-`
(c) `%` (d) `*`

9. Identify the erroneous expression.
(a) `X=y=2, 4;` (b) `res = ++a * 5;`
(c) `res = /4;` (d) `res = a++ -b *2`

10. Function declaration statement identifies a function with its
(a) Name
(b) Arguments
(c) Data type of return value
(d) All of these

11. Which return type cannot return any value to the calling function?
(a) int (b) float
(c) void (d) double

12. Memory is allocated for a function when the function is
(a) declared (b) defined
(c) called (d) returned

13. `*(&num)` is equivalent to writing
(a) `&num` (b) `*num`
(c) `num` (d) None of these

14. Which operator retrieves the `lvalue` of a variable?
(a) `&` (b) `*`
(c) `->` (d) None of these

15. Which operator is used to dereference a pointer?
(a) `&` (b) `*`
(c) `->` (d) None of these

True or False

1. We can have only one function in a C program.

2. Keywords are case sensitive.

3. Variable 'first' is the same as 'First'.

4. Signed variables can increase the maximum positive range.

5. `Comment` statements are not executed by the compiler.

6. Equality operators have higher precedence than the relational operators.
7. Shifting once to the left multiplies the number by 2.
8. `Decision control` statements are used to repeat the execution of a list of statements.
9. `printf("%d", scanf("%d", &num));` is a valid C statement.
10. 1,234 is a valid integer constant.
11. A `printf` statement can generate only one line of output.
12. `stdio.h` is used to store the source code of the program.
13. The closing brace of `main()` is the logical end of the program.
14. The declaration section gives instructions to the computer.
15. Any valid printable ASCII character can be used for a variable name.
16. Underscore can be used anywhere in the variable name.
17. `void` is a data type in C.
18. All arithmetic operators have same precedence.
19. The modulus operator can be used only with integers.
20. The calling function always passes parameters to the called function.
21. The name of a function is global.
22. No function can be declared within the body of another function.
23. The `&` operator retrieves the `lvalue` of the variable.
24. Unary increment and decrement operators have greater precedence than the dereference operator.
25. On 32-bit systems, an integer variable is allocated 4 bytes.

Fill in the Blanks

1. C was developed by _____.
2. The execution of a C program begins at _____.
3. In the memory, characters are stored as _____.
4. `return 0` returns 0 to the _____.
5. _____ finds the remainder of an integer division.
6. `sizeof` is a _____ operator used to calculate the sizes of data types.
7. _____ is also known as forced conversion.
8. _____ is executed when the value of the variable does not match with any of the values of the case statement.
9. _____ function prints data on the monitor.
10. A C program ends with a _____.
11. _____ causes the cursor to move to the next line.
12. A variable can be made constant by declaring it with the qualifier _____ at the time of initialization.
13. _____ operator returns the number of bytes occupied by the operand.
14. The _____ specification is used to read/write a short integer.
15. The _____ specification is used to read/write a hexadecimal integer.
16. To print the data left-justified, _____ specification is used.
17. After the function is executed, the control passes back to the _____.
18. A function that uses another function is known as the _____.
19. The inputs that the function takes are known as _____.
20. Function definition consist of _____ and _____.
21. In _____ method, address of the variable is passed by the calling function to the called function.
22. Size of character pointer is _____.
23. _____ pointer does not point to any valid memory address.
24. The _____ appears on the right side of the assignment statement.
25. The _____ operator informs the compiler that the variable is a pointer variable.