According to the Syllabus of Computer Science of
Board of Intermediate Education, Karachi, Hyderabad,
Sukkur and Larkana,

COMPUTER STUDIES

(Theory and Practical)

By sir Ahad

For Class XII
(General Science Group)

Gul Shahzad Sarwar
Ph.D.
Department of Physics & Computer Science
Government National College
Karachi.

IQRA PUBLISHERS
URDU BAZAR KARACHI

CamScanner

Contents

	page
Preface	xv
CHAPTER 1: CONCEPT OF COMPUTER PROGRAMMING	1
Step 1: Defining the Problem	3
Step 2: Planning the Solution	3
Step 3: Coding the Program	4
Step 4: Testing the Program	5
Step 5: Documenting the Program	7
Programming Languages in General	8
Characteristics of Programming Languages	9
Programming Languages	9
Object-Oriented Programming	14
Visual Programming	16
Exercise	17
Objectives and MCQs	18
CHAPTER 2: ALGORITHM AND FLOWCHARTS	22
Algorithm	22
Pseudocode	24
Mathematical Notations and Functions	24
Algorithmic Notations	25
Flowcharts	26
Categories of Flowcharts	29
Control Structures	30
Sequence	30
Selection	31
Iteration	32
Types of Flowchart	35
Flowcharting Rules	36
Advantages of Flowcharts	42
Disadvantages of Flowcharts	43
Exercise	44
Objectives and MCQs	48

CamScanner

CHAPTER 3: AN OVERVIEW OF 'C' LANGUAGE 50
Introduction to C 50
Why Learn C? 51
The IDE of Turbo C++ Compiler 52
Preparing to Program 60
Your Development Environment 60
Compiling the Source Code 60
Creating an Executable File with the Linker 60
The Development Cycle 61
The Basic Structure of a C Program 61
Understanding Arguments 63
Compiling a Program 64
Running a Program 64
Using Comments 65
Escape Sequences 66
Good Programming Practice 68
Exercise 69
Objectives and MCQs 70

CHAPTER 4: C FUNDAMENTALS 72
The C Character Set 72
Identifiers and Keywords 72
Data Types 73
Choosing the Data Type 78
Constants 78
Symbolic Constants 81
Variables 84
Declaration of Variables 84
Arrays 87
Exercise 89
Objectives and MCQs 97

CHAPTER 5: OPERATORS AND EXPRESSIONS 99
Expressions 99
Arithmetic Operators 99
Order of Precedence 99
Assignment Operators 93
Relational Operators 94
Confusing Equality (==) and Assignment (=) Operators 97
Logical/Boolean Operators 98
Increment and Decrement Operators 99
Conversions of Data Type 101
Casts 103
Library Functions 104
Exercise 105
Objectives and MCQs 107
109

(used to perform mathematicle operation)
(order of mathematicle arthematic place)
(used to compare two numeric operands)

CHAPTER 6: INPUT AND OUTPUT STATEMENTS	**111**
Unformatted I/O Functions	112
Single Character Input	112
The `getchar()` Function	112
The `getch()` and `getche()` Functions	114
Single Character Output	115
The `putchar()` and `putch()` Functions	116
String Input	117
The `gets()` Function	117
String Output	119
The `puts()` Function	119
Formatted I/O Functions	119
The `scanf()` Function	120
The `printf()` Function	125
Exercise	136
Objectives	141
CHAPTER 7: SELECTION CONTROL STRUCTURES	**143**
The `if` Selection Structure	144
Multiple Statements	147
Nested `if` Statements	148
Logical Operators	149
The `if-else` Selection Structure	152
Nested `if-else` Statements	156
The `switch` Statement	159
Exercise	168
Objectives and MCQs	171
CHAPTER 8: ITERATION CONTROL STRUCTURES	**173**
The `for` Loop	173
Multiple Statements	176
Nested `for` Loops	177
The `for` Loop Variations	181
The `while` Loop	187
The Relationship of the `for` and `while` Loops	190
Nested `while` Loops	193
The `do-while` Loop	193
Nested `do-while` Loops	196
Combining Loop Types	197
The `break` Statement	198
The `continue` Statement	200
Exercise	205
Objectives and MCQs	210
CHAPTER 9: FUNCTIONS	**213**
Advantages of Functions	

	214
Classification of Functions	215
General Form of Function	217
Functions with or without Arguments	218
Passing Arguments to a Function	219
Passing Multiple Arguments to a Function	221
Function that Return a Value	225
Using More Than One Function in a Program	229
Recursive Function	233
Exercise	235
Objectives and MCQs	
CHAPTER 10: ARRAYS (Optional)	**237**
One-Dimensional Arrays	237
Declaration of more than One Array	238
Array Initialisation	239
Entering Values into an Array	239
Manipulating Arrays	240
Examining an Array	243
Sorting an Array	247
Searching an Array	249
Passing an Array to a Function	251
Multi-Dimensional Array	255
Exercise	257
Objectives and MCQs	259
CHAPTER 11: STRINGS (Optional)	**262**
String Initialisation	263
String Input/Output	263
Built-in String Functions	264
Array of Strings	268
Exercise	272
Objectives and MCQs	273
CHAPTER 12: STRUCTURES AND UNIONS (Optional)	**275**
Defining a Structure	275
A simple Structure	276
Initialising Structure	279
Entering Data into Structure	280
Array of Structures	282
Nested Structure	285
Structures and Functions	287
Unions	289
An easy Union	290
Union of Structures	291
Exercise	293
Objectives and MCQs	295

CamScanner

CHAPTER 13: POINTERS (Optional) 296
Why Pointers are Used? 296
Pointer Operators 297
The Address Operator (&) 297
The Indirection Operator (*) 297
Pointers and Functions 303
Functions Returning Pointers 307
Pointers and Arrays 307
Arrays of Pointers 312
Pointers and Strings 313
Exercise 316
Objectives and MCQs 319

CHAPTER 14: DATA FILES 321
Declaring a File 321
Opening a File 322
Error Checking 323
Closing a File 324
Simple File Operations 325
Working with Characters 325
Reading Characters from a File 326
Working with Lines 328
Reading Strings 330
Formatted Input and Output 331
Reading Formatted Files 332
Adding Data to a File 334
Copying a File 334
Exercise 336
Objectives and MCQs 337

CHAPTER 15: DATA MANAGEMENT SYSTEMS 338
Character 339
Entity 339
Field or Attribute 339
The Key Field 340
Primary Key and Secondary Key 340
Record 340
File 341
Database 341
Database Management System 342
Advantages of a DBMS 342
Disadvantages of a DBMS 343
Objectives of DBMS 343
Components of a Database Management System 344
Query Language 344
Data Dictionary 345

CamScanner

	343
	346
Report Generator	346
Access Security	346
System Recovery	347
The Basic Database Models	348
The Hierarchical Database Model	349
The Network Database Model	352
The Relational Database Model	354
Exercise	
Objectives and MCQs	
CHAPTER 16: BASICS OF MS ACCESS 2000	**357**
Launching Microsoft Access and Creating a New Database	359
TASK 1: To Launch Microsoft Access and Create a New Database	359
Identifying Microsoft Access Screen Elements	361
Closing Your Database File and Exiting Microsoft Access	362
TASK 2: To Close Your Database File and Exit Microsoft Access	362
Opening an Access Database	362
TASK 3: To Open a Database	362
Creating Tables	364
Naming Fields and Determining Data Types	364
Creating a Table using Table Design View	365
TASK 4: To Create the Students Table Using Table Design View	365
Creating a Table Using the Table Wizard	368
TASK 5: To Create the Address Table Using the Table Wizard	368
Creating a Table by Entering Data Directly	371
TASK 6: To Create the Enrolment Table by Entering Data Directly	371
Inserting Records	374
TASK 7: To Insert a Record to Students Table Using Datasheet View	374
Deleting Records	375
TASK 8: To Delete a Record from Students Table Using Datasheet View	375
Inserting Fields	376
TASK 9: To Insert a Field to Enrolment Table	376
Deleting Fields	376
TASK 10: To Delete a Field from Enrolment Table	376
Traversing the Table Records Using Datasheet View	377
TASK 11: To Traverse among Records in Datasheet View	378
Searching for Records	379
TASK 12: To Search a Table for a Specific Record	379
Updating the Records in a Table	380
TASK 13: To Update Records	380
Sorting Records	381
TASK 14: To Sort Records in a Table	381
Exercise	383
Objectives and MCQs	384
CHAPTER 17: ADVANCED MS ACCESS 2000	**388**
Creating and Modifying Relationships	388

TASK 1: To Open the Relationships Window and Add Tables 388
Using the Relationships Window 389
TASK 2: To create Relationships among Tables 390
TASK 3: To Modify the Relationship between the Students and the Address Tables to Enforce Referential Integrity 391
Creating Select Queries 392
TASK 4: To Create a Select Query Based upon One Table Using Simple Query Wizard 392
TASK 5: To Create a Select Query Based upon One Table Using Query Design View 394
Creating a Query Displaying Data from Multiple Tables 396
TASK 6: To Create a Query Based upon Multiple Tables 396
Adding a Calculated Field to the Query 398
TASK 7: To Add an Expression to Calculate Average Marks 399
Specifying Query Conditions 403
TASK 8: To Specify a Single Query Condition 403
TASK 9: To Specify an AND Condition 404
TASK 10: To Specify an OR Condition 405
Creating Forms 405
Task 11: To Create a New Form Using Form Wizard 406
Creating a New Form Using Design View 407
TASK 12: To Create a New Form Using Design View 407
Modifying a Form's Controls 410
TASK 13: To Delete Label Controls from the Students2 Form 410
Creating Reports 411
TASK 14: To Create a Report Using the Report Wizard 412
Creating a Report Using Report Design View 414
TASK 17: To Create a New Report Using Design View 415
Exercise 417
Objectives and MCQs 418

APPENDIX A: ASCII Character codes for IBM and Compatibles 423
APPENDIX B: Syllabus of Computer Science for Class XII 425
APPENDIX C: Examination Technique 423
APPENDIX D: Fundamental Control Constructions in C Language 430

BIBLIOGRAPHY 437

List of Practicals

page

Practical 1: Writing a Program which prints a Text of four lines consisting of Characters, integers values and floating point values using printf statement. 133

Practical 2: Writing a Program that reads and prints the data using escape sequence (asking the name, age, height and gender of student using scanf and printf statement. 134

Practical 3: Writing a Program which uses operator (calculate the area of triangle, volume of sphere and arrange the resultant values in ascending order). 154

Practical 4: Writing a Program which uses 'for' loop statement, (generate the multiplication tables from 2 to 20). 182

Practical 5(a): Writing a program which uses 'while' loop and nested while loop, (use 'for' loop and continue the process in 'while' loop satisfying this condition.
Write a program to calculate average marks of a student using while and nested while loop. 201

Practical 5(b): Writing a program which uses 'while' loop and nested while loop, (use 'for' loop and continue the process in 'while' loop satisfying this condition.
Write a program to calculate average marks of a student using for and nested while loop. 203

Practical 6: Finding the Factorial of n using 'while' loop, read value of n using scanf, and print the factorial of n. 191

Practical 7(a): Draw a Check-board and print it using if-else statement, and extend the program using nested if-else. 184

Practical 7(b): Extend the Check-board using nested if-else statement. 185

Practical 8: Writing a Program which uses a switch statement and breaks the program if certain condition is observed.
Writing a Program which uses a switch, case and break statements and interconvert the temperature scales using Menu options. 164

Practical 9: Writing a function, which generates factorial of n and calls this function in the 'main' program. 223

Practical 10: Writing a program which uses multiple arguments in a function. (Develop a user-defined function to generate a rectangle. Use the function for passing arguments to draw different sizes of rectangles and squares). 227

CamScanner

By: Sir Ahad

Preface

Thanks to God Almighty, Who enabled me to present the book "Computer Studies".

This book is designed for the beginner, the very beginner. In fact, it assumes that you know nothing about computer programming. It carries you, regardless of your mathematics background and computing experience, from the beginning of programming through the use of advance functions. Apart from this the main aim of this book is to meet the curriculum requirements of the computer science at second year level.

The text of the book has been divided into two parts. The first part presents the fundamentals of programming technique and C programming using Turbo C++ version 3.0. C is one of the most wonderful programming languages. The reason for this is simple: programmers like it. Once a person learns to program in C, it is very uncommon for him or her to switch to another language. C combines delicacy and elegance with raw power and flexibility. It is a structured language that does not confine. C also is a language that puts you firmly in charge. C was created by a programmer for programmers. It is not an abstract product of a committee, but rather the outcome of programmers looking for a better language.

The second part introduces database management systems and Microsoft Access 2000. Access is a versatile, Windows based relational database management system (RDBMS) and it is a part of Microsoft Office 2000.

The book contains very carefully chosen multiple-choice objective types questions and programming exercises which serve as an aid to test one's expertise in C and make one proficient through the feedback obtained from the answers.

There are many useful and interesting programs contained in this book. You probably would like to use them but dislike typing them into the computer. When I type a program, I spend hours trying to get the program to work. For this reason, I am offering the source code on CD-ROM for all the functions and programs contained in this book.

CamScanner

CHAPTER 1

CONCEPT OF COMPUTER PROGRAMMING

Programming is both an art and a science. It is an art because there are so many different ways to accomplish a data processing task and write specific coding instructions. There is room for considerable creativity. It is a science because there are several rules to follow, because logic is required, and because rigorous approaches to programming have been developed to ensure that programs are economical, efficient, and effective.

A *program* is a set of instructions that computer must follow in order to process data into information. The instructions consist of *statements* used in a programming language, such as BASIC, Pascal or C.

There are at least two good reasons for learning programming:

1. Programming helps you understand computers. The computer is only a tool. Learning to write simple programs as you master the machine increases your confidence level. Most people find great personal satisfaction in creating a set of instructions that solve a problem.
2. Learning programming lets you find out quickly whether you like programming and whether you have the analytical turn of mind programmers need. Even if you decide that programming is not for you, understanding the process certainly will increase your appreciation of what programmers and computers can do.

Programming, also called *software engineering*, involves steps similar to any problem-solving task. But before examining these activities, let's see the characteristics of a successful computer program.

CamScanner

1. *correct* it does what it is supposed to do
2. *usable* it is easy to use
3. *reliable* it works without failing
4. *understandable* it can be easily comprehended
5. *modifiable* it can be enhanced and updated
6. *maintainable* it can be corrected when errors are found
7. *flexible* it can be modified to supply other information as required
8. *general* it is not too specialised
9. *efficient* it does not waste computer resources

The five main steps in the programming process are as follows (see Fig. 1.1):

1. Defining the problem
2. Planning the solution
3. Coding the program
4. Testing the program
5. Documenting the program

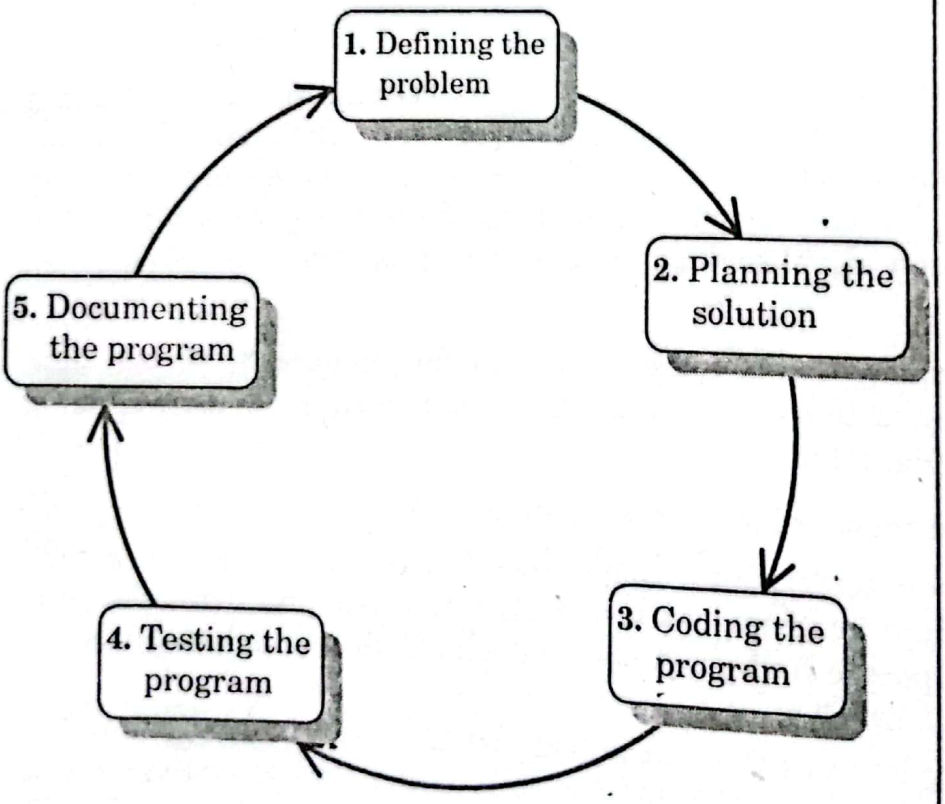


Fig. 1.1. Main steps in the programming process

Let us discuss each of these in turn.

A Programming Pioneer: Grace M. Hooper

When Grace M. Hopper died in 1992 at the age of 85, she left behind the legacy of a true programming pioneer—involvement with the first computers and the first business programming language.

As an M.A. and Ph. D. from Yale University, Hopper joined the U.S. Navel Reserve in 1943. She was assigned to the Bureau of Ordinance Computation Project at Harvard, where she learned to program the first large-scale digital computer, the Mark I. In 1948, Hopper became senior mathematician at the Eckert-Mauchly Computer Corporation. Later she became senior programmer on the team that created the first commercial large-scale electronic computer, UNIVAC I. In the 1950s, she co-authored the COBOL compiler and programming language. Among all of her achievements, the one she liked best was her promotion to the rank of Rear Admiral in the U.S. Naval Reserve.

CamScanner

Step 1: Defining the Problem

It is impossible to develop a solution to a problem if you do not have a clear understanding of the problem itself. The programmer's first task is, therefore, to determine just what the problem is and how to solve it. This can often be accomplished simply by talking with the people who work in an organisation, learning about the facilities and the shortcomings of the current way of working. Defining the problem requires that the programmer specify four mini tasks. They are (1)the program's objectives, (2)the desired output, (3)the input data required, and (4)the processing requirements. Let us consider these four tasks.

Determining Program Objectives: You solve problems every day. A problem might be deciding whether to attend a party at night or not. Or you might try to change your class section in order to adjust in a better way. Thus, every day you determine the *objectives* of the problems you are trying to solve. Programming works the same way. You need to write a statement of the problem you are trying to solve. An example would be "I want to see a payroll report for all employees in our I.T. laboratory".

Determining the Desired Output: It is always better to understand the outputs of a program before you specify the inputs. For example, what kind of hardcopy is wanted? What information should the outputs include? This step may require several meetings with system designers and users to make sure you are creating what they want.

Determining the Input Data: Once you know the output you want, you can determine input data requirements. What kind of input data is needed? What form should it appear in? What is its source? What type or style of user interface is required by the users? For example, for a payroll report, you can specify that the source of the data should be employee's time cards.

Determining the Processing Requirements: Here you make sure you understand the operations which must be performed in order to process the data. Exactly what computers, peripheral equipment, and other programs will be needed to run the program? The software equipments should list the hardware resources that must be available if the program is to run successfully.

Step 2: Planning the Solution

Once the programmer has determined *what* the program must do, the next step is to plan a solution, preferably using *structured programming techniques* or *program design tools*. These techniques consist of the following: (1)top-down program design, (2)algorithm, and (3)flowcharts.

Top-Down Program Design: Once you have determined the outputs and inputs, you can use *top-down program design* to identify the program's processing steps. Such steps are called program *modules*. Each module is made up of logically related program statements. Fig. 1.2 shows an example of a top-down program design for a payroll report. Each of the boxes shown is a module or, in the lower part, a sub-module.

CamScanner

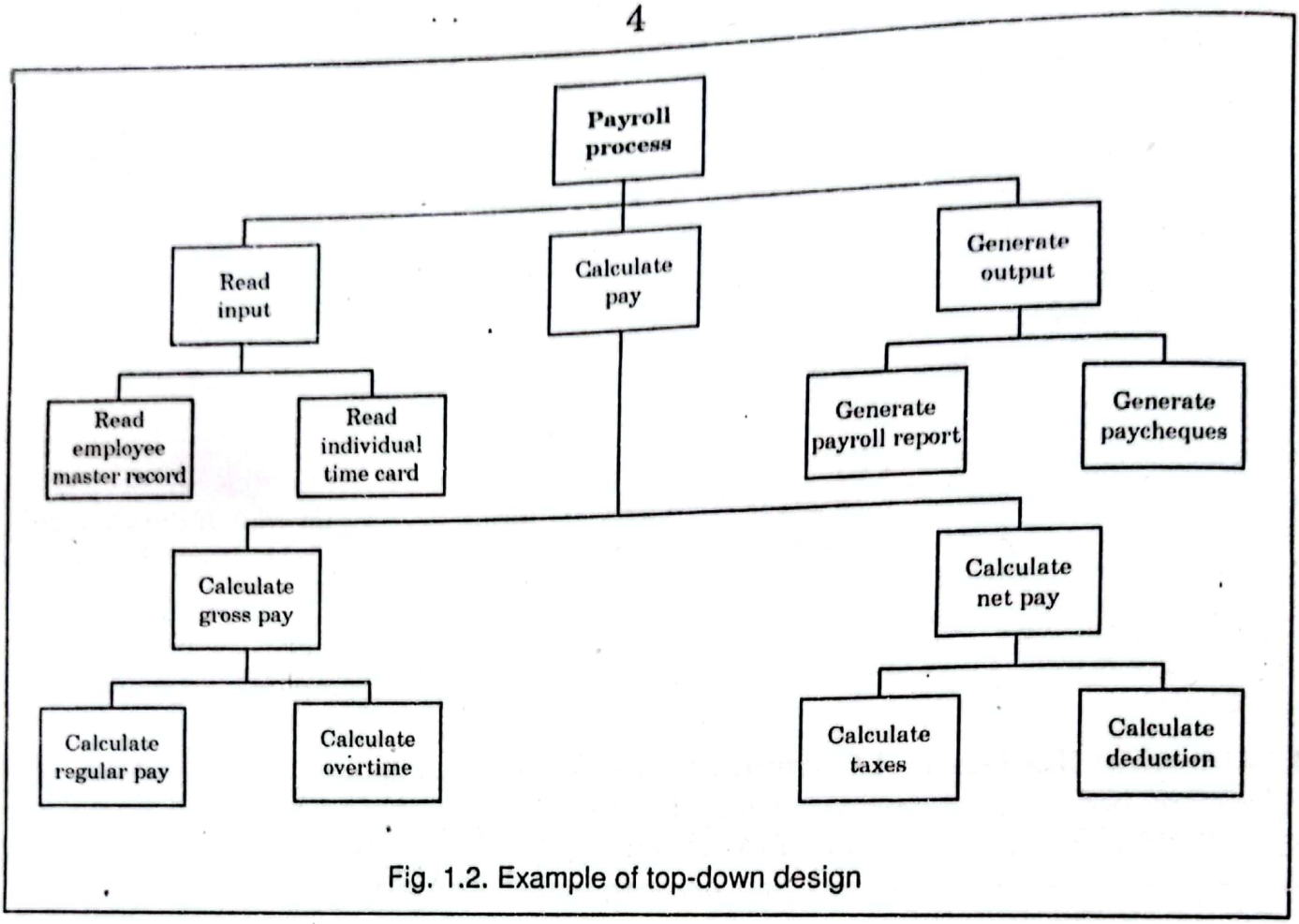


Fig. 1.2. Example of top-down design

According to the rules of top-down design, each module can have only one entry point and one exit point. It is best if each module has only a single function, just as an English paragraph should have a single, complete thought. This rule limits the module's size and complexity, which facilitates testing and debugging. Now-a-days most programmers use structured programming for top-down approach. The theme of structured programming is to make programs more efficient (with fewer lines of code) and better organised (more readable), and to have better notations so that they have clear and correct descriptions.

Algorithm: Algorithm is a step-by-step method of designing a program using normal human-language statements. It describes the logic and processing flow of a program. Algorithm is like an outline or summary form of the program you will write.

Flowchart: A flowchart is a pictorial representation of logic used in a computer program. It illustrates the major program elements and how they are logically integrated. The flowchart uses standard symbol—called ANSI symbols, after the *American National Standard Institute*, which developed them. Chapter 2 describes the algorithm and flowchart in detail.

Step 3: Coding the Program

Once an algorithm has been developed and its flowchart has been completed, the next step is to code the program. *Coding* is the process of writing instructions in a programming language. The language used depends on the system resources available, the expertise of the programmer, and the type of application. As you will learn later in this chapter, each programming language has tasks for which it is particularly well suited.

CamScanner

In order to execute, a program must be in *machine language*. A machine language is a complex language that uses actual machine addresses and difficult operation codes. Because of the complexity of machine language, programs are written in *high-level* languages, such as BASIC, Pascal, C, C++ or PL/1.

Using high-level languages makes software much easier to code. The program is not, however, executable in that form. That is, it cannot be executed until it has been translated or converted into machine language. The computer system itself performs this translation process, using a translator program that reads a high-level program as input and converts it to machine language. The high-level program is called the *source program*, and the translated program in machine language is called the *object program*.

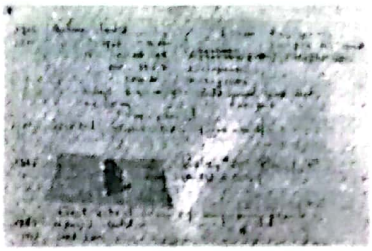
The two main types of translator programs used to translate programs written in high-level languages into machine language are *interpreters* and *compilers*. Interpreters and compilers break down a single high-level instruction or language statement into several machine-language instructions.

Interpreters translate the program one statement at a time as the program is actually being run on the computer. Each statement is translated and executed before the interpreter proceeds to the next statement. This process continues until the end of the program is reached. *Compilers* translate the entire source program into machine language in one process, creating an object program. Another program, referred to as the *linking loader* or *linkage editor*, prepares the object program for execution.

Interpreters are commonly used in the simpler programming languages such as BASIC, whereas most other programming languages use compilers. Interpreters do not take up as much space in the computer's primary storage, but they are slower than compilers. Compilers are capable of creating an object program that can be stored in executable form so that if the program is to be run again, it need not be retranslated each time. Thus, programs that are to be run on a regularly scheduled basis are nearly always written in languages that can be compiled.

Step 4: Testing the Program

After the program has been coded comes the stage to check whether the program works or not? Testing a



Computer literacy books are bursting with bits and bytes and disks and chips and lessons on writing programs in C. All this is to provide quick enlightenment for the computer illiterate. But the average newly literate person has not been told about the bugs.

It is a bit of a surprise, then, to find that the software you are using does not always work quite right. Or, perhaps the programmer who is doing some work for you cannot seem to get the program to work correctly. Both problems are "bugs", errors that were introduced unintentionally into a program when it was written.

The term *bug* comes from an experience in the early days of computing. One summer day in 1945, according to Grace M. Hopper (see the essay "A Programming Pioneer", earlier in this chapter), the Mark I computer came to a halt. Working to find the problem, computer personnel actually found a moth (bug) inside a relay of the machine (see photo above). They removed the offending bug, and the computer was fine. From that day forward, any mysterious problem or glitch was said to be a bug. Hopper wrote in her log book, 'Relay #70 Panel F (moth) in relay. First actual case of bug being found'. That moth carcass is taped to the page in her log book, housed in Navy museum in Virginia.

CamScanner

program to see if it does what it is supposed to do is usually the hardest, longest and most frustrating stage in the program development process. Programming errors are known as *bugs* and the process of detecting and correcting these errors is called *debugging*. In general, *testing* is the process of making sure that the program performs the intended task, and *debugging* is the process of locating and eliminating program errors. Testing and debugging are vital steps in developing computer programs.

In general, there are three types of errors that occur in a computer program: syntax errors, logic errors and runtime errors.

Syntax Errors: A *syntax error* is a violation of the rules of whatever language the program is being written. These are often caused by simple typing mistakes. For example, the C statement

$$C = (A + B/2$$

has a syntax error. In this example, the problem is a missing closing parenthesis which should be placed in the appropriate place depending on the logic of the program. Another example is typing PRINT rather than PRINTF.

Logic Errors: The logical error is an error in planning the program's logic. In this case, the language processor successfully translates the source code into machine code. The computer actually does not know that an error has been made. It follows the program instructions and outputs the results, but the output is not correct. When a logical error occurs, all you know is that the computer is not printing the correct output. The computer does not tell you what is wrong.

For example, you wanted to do the sum of A and B and put the result in the variable C. This is accomplished by typing:

$$C = A + B$$

But while typing, the programmer has, however typed the following expression:

$$C = A - B$$

Such a program will run properly but will give unexpected result because the value of C will not be the *sum* of A and B but it will be the *difference* of A and B which is quite different from the actual requirement. Such errors can only be detected with the help of test data that will give the input values of A and B and the resultant value that should come out of the program as a result of the operation performed on A and B. If the result given by the computer and the result calculated manually with the help of a *calculator* are the same, then you can say that there is no logical error. Otherwise a logic error may be present in the program.

Runtime Errors: Runtime errors occur when a program is run on the computer and the results are not achieved due to some misinterpretation of a particular instruction. This could be something like dividing a number by zero which results in a very large value of quotient. It may also be any other instruction which a computer is not able to understand.

The Debugging Process: Several tools have been devised for finding and removing three types of errors, as follows:

CamScanner

1. *Desk Checking:* In *desk checking,* a programmer checks a printout of the program while sitting at a desk. The programmer goes through the listing line by line looking for syntax, logic and runtime errors. In other words, desk checking is a sort of proofreading.
2. *Manual Testing with Sample Data:* Both correct and incorrect data are run through the program—manually, not with a computer—to test for correct processing results. This is called "dry run of program". If any error is found then it is removed before further testing.
3. *Structured Walkthrough:* A *structured walkthrough* is a process in which three or four programmers, including the creator of the program, review ("walk through") the program. They analyse it for completeness, accuracy, and quality of design and then discuss their conclusions.
4. *Attempt at Translation:* The program is run through a computer, using a translator program. The translator attempts to translate the written program from the programming language (such as C or Pascal) into the machine language of 1s and 0s. Before the program will run, it must be free of syntax errors. These syntax errors will be identified by the translating program.
5. *Testing Sample Data on the Computer:* After all syntax errors have been corrected, the program is tested for logic errors. Sample data is used that will test each program statement. You must plan the sample data carefully to make sure you test every part of the program. This step is sometimes called *alpha testing.*
6. *Run Real-World Data:* After testing with sample data, the program may run fine—in the laboratory. However, if the program is going to be sold commercially, it is released to a selected set of users to be tested with real data, called *beta testing*. The selected users locate errors that were overlooked during the regular testing phase and report them to the company that is developing the program. After beta testing, the program may need additional debugging before it is released to stores.

Step 5: Documenting the Program

Documentation is a written detailed description of the programming cycle and specific facts about the program. Typical program documentation materials include the origin and nature of the problem, a brief narrative description of the program, algorithm and flowcharts, data-record descriptions, program listings, and testing results. Comments in the program itself are also considered an essential part of documentation. Actually, documentation is not something done just at the end of the programming process. It should be carried on throughout all the steps in the program. Documentation is important for several different kinds of people—users, operators, and programmers—who may be involved with the program in the future.

1. *User Documentation:* End users need to know how to use the program. Some organizations may offer training courses, in which human trainers guide users about the working of the program. However, other organizations may expect users to be able to learn just from the written documentation. An example of this sort of documentation is written manuals provided with software and online help system.

CamScanner

2. *Operator Documentation:* The people who run large computers are called *computer operators*. Documentation must be provided for computer operators. If the program sends them error messages, for instance, they need to know what to do about them.
3. *Programmers Documentation:* Even the creator of the original program may not remember many things about it. Other programmers wishing to update and modify it—that is, do what is called *program maintenance*—may find themselves frustrated without adequate documentation. This kind of documentation should include text and program flowcharts. It might also include system flowcharts to show how the particular program relates to other programs within an information system.

Programming Languages in General

A language, as you know, is a means of communicating, usually via human speech or written symbols. A computer *programming language* is a set of symbols and usage rules employed to direct the operations of a computer. *Programmers*, people who write programs, use a variety of programming languages, such as C, C++, Visual BASIC, and Java, to communicate instructions to the computers. Just like any human language, a programming language consists of a set of symbols; rules of *syntax* or *grammar* governing how those symbols are combined; and *semantics*, or meanings associated with certain symbol combinations. However, unlike human languages, programming languages have very limited vocabularies and very specific functions. Human languages are used to express the vast, rich world of human experience and thus are full of colourful, context-dependent words and ambiguous rules. The single purpose of programming languages is the construction of computer programs.

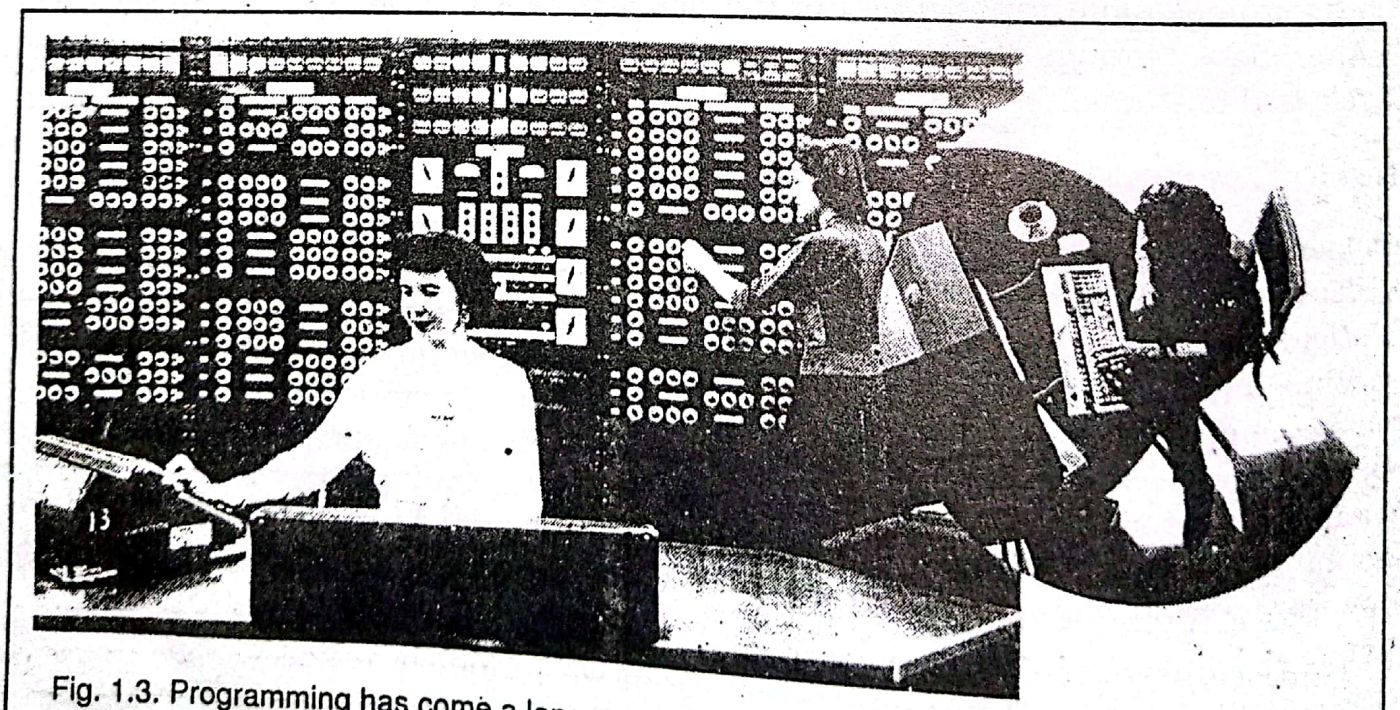


Fig. 1.3. Programming has come a long way in more than half a century. Early programmers had to set hundreds of switches manually to enter a program. To run another program, they had to reset the switches. Today, the technology assists us in programming effort while providing us with a user-friendly interface.

CamScanner

Characteristics of Programming Languages

Although there are many different programming languages, each one designed to solve certain types of problems, all of them have the basic function of directing the operations of a computer. Therefore, certain general classes of instructions are present in every programming language, which can be categorised as follows:

Input/Output Instructions: These instructions direct the computer to "read from" or "write to" a peripheral device (for example, disk drive or a printer).

Computation Instructions: These instructions direct the computer to perform arithmetic operations (add, subtract, multiply, divide, and raise a number to a power). For example, PAY = HOURS × RATE computes gross earnings for hourly employees.

Control Instructions: These instructions can alter the sequence of the program's execution or terminate execution. For example, in an accounts receivable program, different sequences of instructions are executed for customers who pay on time and for those who pay after the due date.

Assignment Instructions: These instructions transfer data internally from one RAM location to another. For example, a data item may be copied from one location to another, the result of a computation may be stored at a specified location, or an intermediate result may be retrieved from a given location.

All programming languages must be able, in one way or another, to perform the above types of operations. In addition to the above kinds of statements, almost every computer language includes some provision for programmers to insert explanatory notes within their programs. These descriptive sections of text are known as *comments* and are ignored by the computer during program translation and execution. Their purpose is to help clarify the program's operation for users.

Programming Languages

Over one hundred programming languages exist. Different languages exist because each was designed for specific purposes. For example, some languages solve typical business problems; others perform large, complex calculations; and still others create graphics. No matter which high-level language a program is written in, it must be translated into machine language before it can be executed. System software programs called *compilers* and *interpreters* perform this task.

Early high-level languages fell in the *procedure-oriented language* category. Procedure-oriented languages require programmers to solve programming problems using traditional programming logic; that is, programmers write the instructions in the sequence in which they must be executed to solve the problem. Procedure-oriented languages are intended to solve *general* problems and are classified as *business, scientific,* or *multipurpose.*

Some popular high-level languages are described below.

CamScanner

FORTRAN: The Language of Mathematics & the First High-Level Language

FORTRAN, or FORmula TRANslator, was developed by IBM in 1954 and was the *first high-level language*. Originally designed to express mathematical formulas, it is still the most widely used language for mathematical, scientific, and engineering problems. It is also useful for complex business applications, such as forecasting and modelling. However, because it cannot handle a large volume of input/output operations or file processing, it is not used for more typical business problems. The newest version of FORTRAN is FORTRAN 90.

FORTRAN has both advantages and disadvantages:

Advantages

1. FORTRAN can handle complex mathematical and logical expressions.
2. Its statements are relatively short and simple.
3. FORTRAN programs developed on one type of computer can often be easily modified to work on other types.

Disadvantages

1. FORTRAN does not handle input and output operations to storage devices as efficiently as some other higher-level languages.
2. It has only a limited ability to express and process nonnumeric data.
3. It is not as easy to read and understand as some other high-level languages.

COBOL: The Language of Business

The name COBOL stands for COmmon Business Oriented Language. COBOL was introduced in 1960, after having been developed by the Conference on Data Systems Languages (CODASYL), a committee of government and industry representatives called together by the U.S. Department of Defence. COBOL is the most frequently used business programming language for large computers. It is designed to store, retrieve, and process corporate accounting information and to automate such functions as inventory control, billing, and payroll.

The most significant attribute of COBOL is that it is extremely readable; even if you know nothing about programming, you may still understand what the program does. However, the feature that makes COBOL so useful—its English-like appearance and easy readability—is also a weakness, because a COBOL program can be incredibly verbose. Versions of COBOL are available for personal computers; the newest, Visual COBOL, is an object-oriented version of the language.

COBOL, too, has both advantages and disadvantages:

Advantages

1. It is machine independent.
2. Its English-like statements are easy to understand, even for a nonprogrammer.
3. It can handle many files, records, and fields.
4. It easily handles input/output operations.

CamScanner

FORTRAN

```
IF (XINVO.GT. 500.00) THEN
    DISCNT = 0.07*XINVO
ELSE
    DISCNT = 0.0
ENDIF
XINVO = XINVO - DISCNT
```

COBOL

```
OPEN-INVOICE-FILE.
    OPEN I-O INVOICE FILE.
READ-INVOICE-PROCESS.
    PERFORM READ-NEXT-REC THROUGH READ-NEXT-REC-EXIT UNTIL END-OF-FILE.
    STOP RUN.
READ-NEXT-REC.
    READ INVOICE-REC
        INVALID KEY
            DISPLAY 'ERROR READING INVOICE FILE'
            MOVE 'Y' TO EOF-FLAG
            GOTO READ-NEXT-REC-EXIT.
    IF INVOICE-AMT > 500
        COMPUTE INVOICE-AMT = INVOICE-AMT - (INVOICE-AMT*.07)
        REWRITE INVOICE-REC.
READ-NEXT-REC-EXIT.
    EXIT.
```

BASIC

```
10  REM       This Program Calculates a Discount Based on the Invoice Amount
20  REM       If Invoice Amount is Greater Than 500, Discount is 7%
30  REM       Otherwise Discount is 0
40  REM
50  INPUT "What is the Invoice Amount"; INV.AMT
60  IF INV.AMT A > 500 THEN LET DISCOUNT = 0.7 ELSE LET DISCOUNT = 0
70  REM         Display Results
80  PRINT "Original Amt", "Discount", "Amt after Discount"
90  PRINT INV.AMT, INV.AMT * DISCOUNT, INV.AMT-INV.AMT*DISCOUNT
100 END
```

Pascal

```
if INVOICEAMOUNT > 500.00 then
    DISCOUNT :=  0.07*INVOICEAMOUNT
else
    DISCOUNT := 0.0;
INVOICEAMOUNT := INVOICEAMOUNT - DISCOUNT
```

C

```
if(invoice_amount > 500.00)
    DISCOUNT =  0.07*invoice_amount;
else
    discount = 0.00;
invoice_amount = invoice_amount - discount;
```

Fig. 1.4. This shows how five languages handle the same statement. The statement specifies that a customer gets a discount of 7% of the invoice amount if the invoice is greater than Rs. 500; if the invoice is lower, there is no discount.

CamScanner

Disadvantages

1. Because it is so readable, it is wordy. Thus, even simple programs are lengthy, and programmer productivity is slowed.
2. It cannot handle mathematical processing.

BASIC: For Beginners and Others

BASIC—Beginner's All-purpose Symbolic Instruction Code—was developed by Dr. John Kemeny and Dr. Thomas Kurtz at Dartmouth College in 1965 for use in training their students. By the late 1960s, it was widely used in academic settings on all kinds of computers, from mainframes to PCs.

BASIC is the most popular microcomputer language and is considered the easiest programming language to learn. Although it is available in compiler form, the interpreter form is more popular with first time and casual users. This is because it is interactive, meaning that user and computer can communicate with each other during the writing and running of the program. Several versions of BASIC are available such as GWBASIC, Quick BASIC and Visual BASIC.

The advantages and disadvantages of BASIC are as follows:

Advantage

1. The primary advantage of BASIC is its ease of use.

Disadvantages

1. Its processing speed is relatively slow, although compiler versions are faster than interpreter versions.
2. There is no one version of BASIC, although in 1987 ANSI adopted a new standard that eliminated portability problems—that is, problems with running it on different computers.

Pascal: The Simple Language

Named after the seventeenth-century French mathematician Blaise Pascal, *Pascal* was developed by a Swiss computer scientist, Niklaus Wirth, and became available in 1971. It is an alternative to BASIC and was originally intended as a general-purpose, high-level language for teaching the concepts of structured programming and top-down design. Pascal's simplicity, elegance, and embodiment of structured programming principles have made it quite popular with computer scientists and students, scientific programmers, and microcomputers users.

A compiled language, Pascal offers these advantages and disadvantages:

Advantages

1. Pascal is easy to learn.
2. It has extensive capabilities for graphics programming.
3. It is excellent for scientific use.

Disadvantage

1. Pascal has limited input/output programming capabilities, which limits its business applications.

CamScanner

C: A Portable Language

"C" is the language's entire name, and it does not "stand" for anything. C was developed by Dennis Ritchie at Bell (now AT & T's Bell) Laboratories of USA in 1972. It was originally developed for writing system software (most of the UNIX operating system was written using C) but is now considered a general-purpose language. Today it is widely used for writing applications including word processing, spreadsheets, games, robotics, and graphics programs. It is now considered a necessary language for programmers to know.

Because it provides the advantages of both assembly language and high-level languages, C is often referred to as a "middle-level" language. It is a structured language and uses high-level commands, but it also allows the programmer to interact directly with the hardware, as in assembly language. These capabilities make C well suited for systems programming that requires the programmer to write extremely efficient code that is not typically needed in normal business programming. In addition, code written in C can easily be transported from one system to another.

Here are the advantages and disadvantages of C:

Advantages

1. C works well with microcomputers.
2. It has a high degree of portability—it can be run without change on a variety of computers.
3. It is fast and efficient.
4. It enables the programmer to manipulate individual bits in main memory.

Disadvantages

1. C is considered difficult to learn.
2. Because of its conciseness, the code can be difficult to follow.
3. It is not suited to applications that require a lot of report formatting and data file manipulation.

PL/1: The Best of FORTRAN & COBOL?

PL/1 was introduced in 1964 by IBM. PL/1 *(for Programming Language 1)* is a third-generation language designed to process both business and scientific applications. It contains many of the best features of FORTRAN and COBOL and is quite flexible and easy to learn. However, it is also considered to have so many options as to diminish its usefulness. As a result, it has not given FORTRAN and COBOL much competition.

Today there are several versions of PL/1, including PL/C and PL/M that can run on a variety of computers.

Advantage

1. PL/1 is most effective in organisations that require both scientific and commercial applications.

CamScanner

Disadvantage

1. PL/1 is difficult to learn.

Logo: Using the Turtle to Teach Children

Logo was developed at the Massachusetts Institute of Technology in 1967 by Seymour Papert, using a dialect of LISP (*LIS*t *P*rocessing language, 1959). Logo is a third-generation language designed primarily to teach children problem—solving and programming skills. At the basis of Logo is a triangular pointer, called a "turtle", which responds to a few simple commands such as forward, left, and right. The pointer produces similar movements on the screen, enabling users to draw geometric patterns and pictures on screen. Because of its highly interactive nature, Logo is used not only by children but also to produce graphic reports in business.

Advantages

1. It helps to build programming skills in children.
2. Logo is very easy to learn.
3. It has extensive capabilities for graphics programming.

Disadvantage

1. It has only a limited ability to express and process data.

Ada: A Possible New Standard

Ada is an extremely powerful structured programming language designed by the U.S. Department of Defence to ensure portability of programs from one application to another. Ada was named for Countess Ada Lovelace, considered the world's "first programmer" and was introduced in 1980. Based on Pascal, Ada was originally intended to be a standard language for weapons systems. However, it has been used in successful commercial applications.

Advantages

1. Ada is a powerful, comprehensive language applicable to a wide range of problems.
2. Because it is an structured language, modules (sections) of a large program can be written, compiled, and tested separately before the entire program is put together.
3. It has features that permit the compiler to check errors before the program is run, programmers are more apt to write error free programs.

Disadvantages

1. Ada has a high level of complexity and difficult to learn.
2. Presently, the number of machines that can compile Ada programs is limited, and this is the language's major weakness. A few compilers that translate subsets of the full language are available for microcomputers.

Object-Oriented Programming

Programming is difficult work and takes time. Computer professionals have been working on ways to make programming simpler, easier, and faster. Object-oriented programming languages are a recent development to improve the programming process.

CamScanner

Object-oriented programming (OOP) is a programming method that combines data with instructions for processing that data to create a self-sufficient "object", or block of pre-assembled programming code, that can be used in other programs.

Object-oriented programming attempts to completely alter traditional software development methods, and many computer professionals believe it has the potential for satisfying some of the enormous demand for more sophisticated software. Object-oriented programming requires a new way of programming, one more closely related to how we actually think.

We typically regard the computer as a machine and data as the raw material that the computer processes. The programmer is the technician who controls the machine. The program lists all the steps the computer must take to obtain the needed output from the input. However, an object-oriented program defines the data and the set of operations that can act on that data as one unit called an *object*. The object is thought of as an actor with a specific set of skills. The programmer is the director of the show. The programmer no longer has to tell each actor exactly what steps to take but instead simply explains the ultimate task to be performed.

Once you have written a block of program code (that computes overtime pay, for example), it can be reused in any number of programs. Thus, unlike traditional programming, with OOP you don't have to start from scratch each time (see Fig. 1.4).

Object-oriented programming takes longer to learn than traditional programming because it means training oneself to a new way of thinking. Once learned, however, the beauty of OOP is that an object can be used repeatedly in different applications and by different programmers, speeding up development time and lowering costs.

Some examples of OOP programming languages are Smalltalk, C++, and Hypertalk.

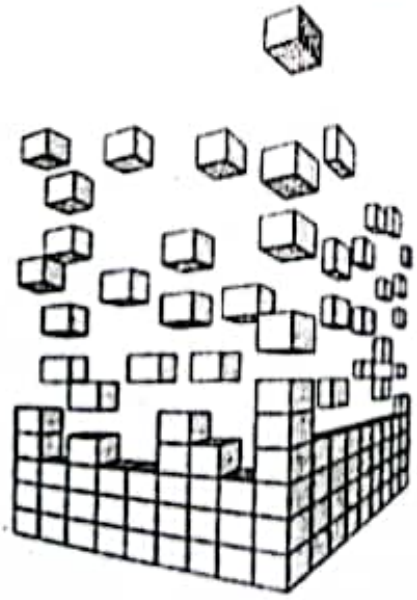


Fig. 1.4(a). Conventional programs: when building conventional programs, programmers write every line of code from scratch.

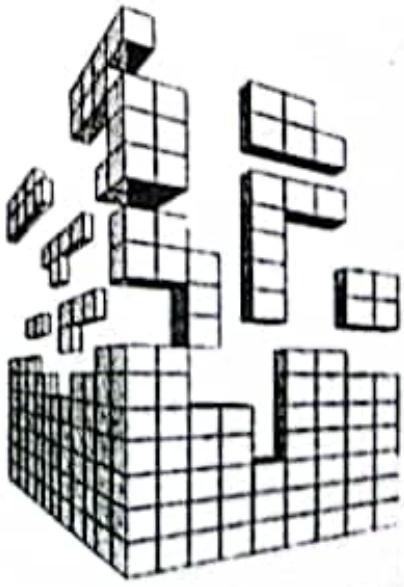


Fig. 1.4(b). Object-oriented programs: with object-oriented programs, programmers can use blocks, of "objects", of pre-assembled modules containing data and the associated processing instructions.

CamScanner

Visual Programming

Visual programming is a method of creating programs in which the programmer makes connections between objects by drawing, pointing, and clicking on diagrams and icons.

Essentially, visual programming takes OOP to the next level. The goal of visual programming is to make programming easier for programmers and more accessible to nonprogrammers by borrowing the object orientation of OOP languages but exercising it in a graphical or visual way. Visual programming enables users to think more about the problem solving than about handling the programming language. There is no learning of syntax.

Visual BASIC, developed by Microsoft in the early 1990s, is the most popular visual programming language. It offers a visual environment for program construction, allowing users to build various application components using drag-and-drop tools, buttons, scroll bars, and menus. Visual BASIC, a fourth generation OOP-based language, works with Microsoft Windows to create other Windows-compatible applications.

CamScanner

Exercise

1.1. What is the difference between human languages and computer languages?
1.2. Descried the four general classes of instructions found in all computer languages.
1.3. What is the difference between a compiler and an interpreter? Why is each used?
1.4. What is the definition of a program?
1.5. What is the definition of programming?
1.6. Explain the steps in a programming project.
1.7. What is an algorithm?
1.8. Distinguish between syntax errors and logic errors.
1.9. Explain how and why a program is tested and debugged?
1.10. Explain some of the debugging tools, such as desk checking and structured walk-through.
1.11. Why is proper documentation of a program necessary?
1.12. List the most commonly used microcomputer high-level programming languages.
1.13. What were the reasons behind the development of high-level programming languages?
1.14. What is the difference between procedure-oriented languages and object-oriented languages?
1.15. What is visual programming?

CamScanner

Objectives and MCQs

Choose the answer that best completes the statement.

1.1. _____ is a relatively simple high-level language that was developed to help students learn programming.
   (a) BASIC
   (b) COBOL
   (c) C
   (d) All of the above

1.2. In preparing a program, one should first
   (a) plan the solution
   (b) document the program
   (c) code the program
   (d) define the problem

1.3. During the development of a program, drawing a flowchart is a means to
   (a) plan the solution
   (b) define the problem
   (c) code the program
   (d) analyse the problem

1.4. Which of the following is used to design a program using English-like statements?
   (a) Algorithm
   (b) program flowchart
   (c) control structures
   (d) Non of the above

1.5. Compilers and interpreters are types of _____.
   (a) programming languages
   (b) language translators
   (c) alpha testers
   (d) application generators

1.6. Which of the following type of error is detected by a language translator?
   (a) program design
   (b) logic
   (c) syntax
   (d) both b and c

1.7. You test a program to find which of the following?
   (a) flowcharting errors
   (b) logic errors
   (c) algorithmic errors
   (d) syntax errors

CamScanner

1.8. In preparing a program, desk-checking and translating are examples of
(a) coding
(b) testing
(c) planning
(d) documenting

1.9. Which of the following is not one of the steps in a programming project?
(a) planning the solution
(b) coding the program
(c) testing the program
(d) selecting the hardware

1.10. In preparing a program, desk-checking and translating are examples of
(a) coding
(b) planning
(c) testing
(d) documenting

1.11. Comments in the program itself are part of
(a) compiling
(b) translating
(c) linking
(d) documenting

1.12. The first high-level language to be introduced was
(a) COBOL
(b) FORTRAN
(c) Pascal
(d) BASIC

1.13. The language named for a French mathematician is
(a) C
(b) Pascal
(c) FORTRAN
(d) COBOL

1.14. Specifying the kind of input, processing, and output required for a program occurs when
(a) planning the solution
(b) coding the program
(c) flowcharting the problem
(d) defining the problem

1.15. Error messages provided by a compiler are called
(a) bugs
(b) translations
(c) diagnostics
(d) mistakes

CamScanner

1.16. After stating a solution in algorithm, you would next
(a) test the program
(b) implement the program
(c) code the program
(d) translate the program

1.17. Software that translates assembly language into machine language is a(n)
(a) binary translator
(b) assembler
(c) compiler
(d) link-loader

1.18. A standardized business language is
(a) Pascal
(b) COBOL
(c) BASIC
(d) FORTRAN

1.19. In developing a program, documentation should be done
(a) as the last step
(b) only to explain errors
(c) throughout the process
(d) only during the design phase

1.20. Which of the following languages is frequently used in object-oriented programming?
(a) COBOL
(b) C
(c) C++
(d) Pascal

1.21. Popular object-oriented languages are
(a) Pascal, COBOL
(b) C++, FORTRAN
(c) C++, Smalltalk
(d) COBOL, BASIC

CamScanner

By Sir Ahad

Objectives

1.1. Desk-checking is the first phase of testing a program.

1.2. The rules for using a programming language are called syntax.

1.3. Debugging is the process of locating program logic errors.

1.4. It is correct to refer to programming as software engineering.

1.5. A syntax error can be caused by a simple typographical error.

1.6. A syntax error occurs when you do not follow the rules of the programming language.

1.7. Top-down program design identifies the program's processing steps. Such steps are called program modules.

1.8. The programming process step that is best done throughout the process is documentation.

1.9. Two common methods of planning the solution to a problem are (a) algorithm and (b) flowcharting.

1.10. The next step after a programmer has planned the solution is to code the program.

1.11. Programming languages that are portable can be executed on many different computers with minimal changes.

1.12. Low-level languages are tied more closely to the computer than are high-level languages.

1.13. Traditional programming languages like COBOL and Pascal are known as procedure-oriented languages.

1.14. COBOL is the language most commonly used for business programs that need to access large files of data.

1.15. Niklaus Wirth developed Pascal to teach structured programming techniques to students.

1.16. Pascal is particularly easy to use because it has fewer features than most languages.

1.17. FORTRAN is used primarily in scientific environments.

1.18. PL/1 was developed to combine the advantages of FORTRAN and COBOL.

1.19. Objects found in an object-oriented program can be reused in other programs.

CamScanner

CHAPTER 2

ALGORITHM AND FLOWCHARTS

Algorithm

The word algorithm is derived from the well known ninth century mathematician of Iran Abu Jafar Ibn Musa Al-Khwarizmi, who developed methods to solve problems using specific instructions step-by-step. Originally the term was *algorism* but later on, it changed to *algorithm*.

Consequently the *algorithm* can be defined as a finite sequence of unambiguous instructions designed to solve a particular problem and these instructions ultimately terminate if followed. You may think that the latter part of this definition (requiring a finite sequence of instructions to terminate) is extra. However, the fact that a list of directions contains only a finite number of instructions does not necessarily mean that the described task will ever be completed if followed. For example, the single instruction that reads "Do this step again" results in an endless process, whereas "If you have done this step four times, then stop; otherwise, do it again" terminates. In summary, the latter part of the definition of an algorithm means that an algorithm must be both definable in a finite number of instructions and executable in a finite amount of time.

The use of the term *unambiguous* in the definition of an algorithm means that at each step the action to be performed next must be uniquely determined by the instruction and the data available at that time. Observe that this does not mean that the instruction alone must be enough to determi...

knowing that a person is about to execute the "if" instruction from the previous paragraph is not enough information to determine what that person's action is going to be. We must also know how many times the person has already executed the instruction.

The requirement that each instruction in an algorithm be executable eliminates the possibility of an algorithm containing instructions whose execution is impossible. For instance, the sequence of instructions

1. Make a list of all the positive integers.
2. Arrange this list in descending order (from largest to smallest).
3. Extract the first integer from the resulting list.
4. Stop.

is not an algorithm because the execution of some of its instructions is impossible. One cannot make a list of all the positive integers as requested by instruction 1, and the positive integers cannot be arranged in descending order as requested by instruction 2.

To gain insight into algorithms, let us consider a simple example.

Example 2.1

There are 50 students in a class who appeared in their final examination. Their mark-sheets have been given to you. Write an algorithm to calculate and print the total number of students who passed in first division.

Algorithm 2.1

Step 1. Initialise TOTAL FIRST DIVISION and TOTAL MARKS SHEETS CHECKED to zero.

Step 2. Take the mark-sheet of the next student.

Step 3. Check the division column of the mark-sheet to see if it is FIRST DIVISION: if no, go to step 5.

Step 4. Add 1 to TOTAL FIRST DIVISION.

Step 5. Add 1 to TOTAL MARK-SHEETS CHECKED.

Step 6. Is TOTAL MARK-SHEETS CHECKED = 50: if no, go to step 2.

Step 7. Write TOTAL FIRST DIVISION.

Step 8. Exit.

It must be clear to the readers from this example that even for very simple problem, the development of algorithms is not so simple as it might initially appear and requires some thinking. It may also be noted from the given example that in order to solve a given problem, each and every instruction must be strictly carried out in a particular sequence. It is this fact which a beginner to problem solving by computers finds difficult to appreciate.

Example 2.2

Compute the distance travelled by a train in a specified time and speed. You are

CamScanner

given with the data that a train travels at a speed of 160 km per hour and reaches its destination in one hour and forty-five minutes. Write an algorithm to compute the distance travelled by train in the specific time.

Algorithm 2.2

Step 1. Start the calculation.
Step 2. Enter a value for SPEED.
Step 3. Enter a value for TIME.
Step 4. Calculate DISTANCE by using formula SPEED × TIME.
Step 5. Write the value of DISTANCE.
Step 6. Exit.

Pseudocode

Pseudocode is an English-like way of representing the solution to a problem. The prefix *pseudo* means *fake;* pseudocode, therefore literally means *fake code*—that is, not the code that is actually entered into the computer. It is considered a "first draft" because the pseudocode eventually has to be translated into a programming language. In general, a pseudocode is a notational system in which ideas can be expressed informally during the algorithm development process.

When using pseudocode to plan a program, you can concentrate on the logic and not worry about the rules of a specific language. It is also easy to change pseudocode if your discover a mistake in your logic—once it is coded in a programming language, most people find that it is more difficult to change logic. Pseudocode can be translated into a variety of programming languages, such as C, C++, Pascal or COBOL. Therefore, pseudocode instructions can be considered as midway between English and high-level languages but free from ambiguities of English language. Now from this point onward we will develop all the algorithms in pseudocodes.

Mathematical Notations and Functions

This section gives various mathematical functions which appear very often in the analysis of algorithms and in computer science in general, together with their notation.

Remainder Function

Let k be any integer and let M be a positive integer, then

$$k(\text{MOD } M) = \text{remainder}$$

(read *k modulo M*) will denote the integer remainder when k is divided by M.

Example 2.3

11(MOD 2) = 1
22(MOD 3) = 1
24(MOD 7) = 3
3(MOD 7) = 3
20(MOD 4) = 0
4(MOD 10) = 4

Arithmetic modulo M refers to the arithmetic operations of additions, multiplication and subtraction where the arithmetic value is replaced by its equivalent value in the set

$$\{1, 2, 3, \ldots, M\}$$

CamScanner

For example, in arithmetic modulo 12, sometimes called 'clock' arithmetic,

$$6 + 9 \equiv 3 \qquad 7 \times 5 \equiv 11 \qquad 1 - 5 \equiv 8 \qquad 2 + 10 \equiv 0 \equiv 12$$

(The use of 0 or M depends on the application.)

Integer Value Function

Let X be any real number then the *integer value* of X, written as INT(X), converts X into an integer by deleting (truncating) the fractional part of the number. Thus

INT (4.32) = 4 INT(5.392) = 5 INT (−8.7) = −8 INT(9) = 9

Absolute Value Function

The absolute value of the real number X, written as ABS(X) or $|X|$, is defined as the greater of X or $-X$. Hence

ABS (0) = 0
ABS (X) = X
ABS ($-X$) = X

Thus

ABS(−15) = 15 ABS(6) = 6 ABS(3.86) = 3.86 ABS(−0.21) = 0.21

Note that $|X| = |-X|$ and, for $X \neq 0$, $|X|$ is positive.

Factorial Function

The product of the positive integers from 1 to n is denoted by $n!$ (read 'n factorial'). That is,

$$n! = 1 \times 2 \times 3 \times \ldots \times (n-2) \times (n-1) \times n$$

It is also convenient to define 0! = 1.

Example 2.4

$2! = 1 \times 2 = 2$ $3! = 1 \times 2 \times 3 = 6$ $4! = 1 \times 2 \times 3 \times 4 = 24$

Algorithmic Notations

An algorithm, as stated above, is a finite step-by-step list of well-define instructions for solving a particular problem. Different symbols are used to define the different activities and which can be understood easily by the following formal presentation of an algorithm.

Algorithm 2.3

This algorithm takes two numbers X and Y and output their sum.

1. Read X. [To get first Number.]
2. Read Y. [To get second Number.]
3. Set SUM = $X + Y$. [Summing two Numbers.]
4. Write SUM. [Displaying result on Screen.]
5. Exit. [Termination of Program.]

CamScanner

The format for the formal presentation of an algorithm consists of two parts. The first part is a paragraph which tells the purpose of the algorithm, identifies the variables which occur in the algorithm and lists the input data. The second part of the algorithm consists of the list of steps that is to be executed.

Following paragraphs summarises certain points and terminology used in making algorithm.

1. *Algorithm Name:* Each algorithm must be given a unique name for identification (*e.g.* algorithm 2.3 refer to the third algorithm in chapter 2).
2. *Step Number:* Each statement of the algorithm must be given a unique number for identification (*e.g.* 1, 2, 3, etc.).
3. *Variable Names:* Variable names will use capital letters, as *X, Y*, SUM.
4. *Getting Input from the User:* 'Read' is used to get input from the user (computer ask the user for value).
5. *Assigning Value to a Variable:* 'Set' is used to assign value to a variable (computer itself sets the value for variable).
6. *Algorithm Termination:* 'Exit' is used to terminate the algorithm.
7. *Comments:* '[' and ']' are used for providing comments or remarks.
8. *Control Transfer:* Use 'Goto Step *n*' to transfer control to step *n*.

Flowcharts

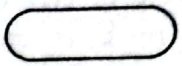
If the planning done for the programs is retained as a part of the program documentation, it can then be used as a basis for future changes in the program. Once a computer program has been written to solve a particular problem, that program will continue to solve the problem in the same way for as long as it is used. A computer solution will continue to be a good solution until the nature of the problem changes. Unfortunately, the nature of all organisations is to be in a continual state of change. With the passage of time, the computer program will be solving problems that no longer exist—at least not in the same form as when the program was written.

When the time comes for a modification of the computer solution that will match the evolved problem, the original program plans can be used by the programmer to identify those portions of the program that require changes.

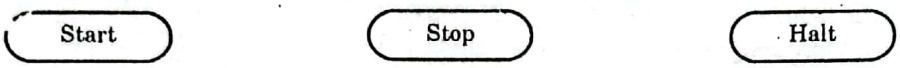
Flowcharting is one technique used in program planning. A *flowchart* is a pictorial representation of logic used in a computer program. It illustrates the major program elements and how they are logically integrated. The flowchart uses *arrows* to represent the direction of the program flow and *boxes* and other shapes to display actions. The *American National Standard Institute* (ANSI) has established standards for the symbols and boxes that are used in flowcharting. The shape of the boxes represents the type of activity that is to be performed. These symbols and examples of their use are illustrated below. Although many more symbols are included in the standard set, those shown below are encountered most frequently.

CamScanner

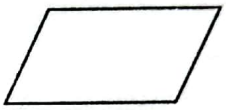
Terminal Symbol: The terminal symbol is used to indicate the beginning (START), ending (STOP) and pauses (HALT) in the program logic flow. It is the first symbol and the last symbol in the program logic. In addition, if the program logic calls for a pause in the program, that also is indicated with a terminal symbol. A pause is normally used in the program logic under some error conditions or if forms had to be changed in the computer's line printer during the processing of that program.



(Oval)

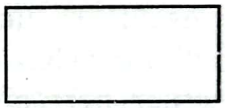


Input/Output Symbol: The input/output symbol is used to denote any type of input data or output information. This symbol represents generic input/output functions of all kinds.



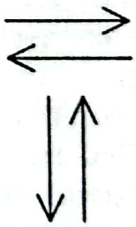
(Parallelogram)

Process Symbol: A process symbol is used in a flowchart to represent an operation involved in actual processing of data. The process symbol is used for all arithmetic and data transfer (MOVE) operations. When more than one arithmetic and data movement instructions are to be executed consecutively, they are normally placed in the same processing box and they are assumed to be executed in the order of their appearance.

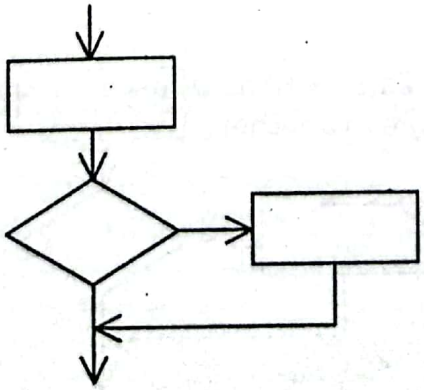


(Rectangle)

Flowlines: Flowlines with arrowheads are used to indicate the flow of operation, that is, the exact sequence in which the instructions are to be executed. The normal flow of flowchart is from top to bottom and left to right. Arrowheads are required only when the normal top to bottom flow is not to be followed. However, as a good practice and in order to avoid ambiguity, flowlines are usually drawn with an arrowhead at the point of entry to a symbol. Good practice also dictates that flowlines should not cross each other and that such intersections should be avoided whenever possible.

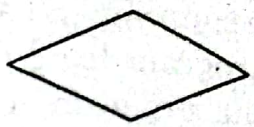


(Arrow)

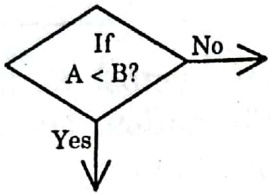


CamScanner

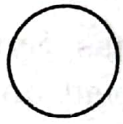
Decision Symbol: The decision symbol is used in a flowchart to indicate a point at which a decision has to be made and a branch to one of two or more alternative points is possible. The criterion for making the decision should be indicated clearly within the decision box. Moreover, the condition upon which each of the possible exit paths will be executed should be identified and all the possible paths should be accounted for. During execution, the appropriate path is followed depending upon the result of the decision.



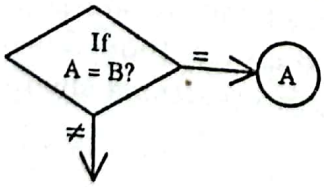
(Diamond)



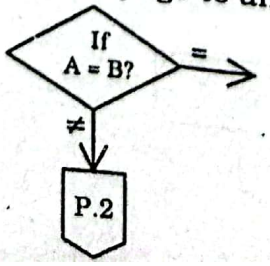
On-Page Connector Symbol: If a flowchart becomes very long, the flowlines start crossing at many places that causes confusion and reduces understandability of the flowchart. In this situation it is useful to utilise the connector symbol as a substitute for flowlines. This symbol represents an entry from, or an exit to another part of the flowchart. An on-page connector symbol is represented by a circle and a letter or digit is placed within the circle to indicate the link. A pair of identically labelled connector symbols is commonly used to indicate a continued flow when the use of a line is confusing. So two connectors with identical labels serve the same function as a long flowline. That is, they show an exit to some other chart section, or they indicate an entry from another part of the chart. How is it possible to determine if an on-page connector is used as an entry or an exit point? It is very simple: if an arrow enters but does not leave a connector, it is an exit point and program control is transferred to the identically labelled connector that does have an outlet. It may be noted that connectors do not represent any operation and their use in a flowchart is only for the sake of convenience and clarity.



(Circle)

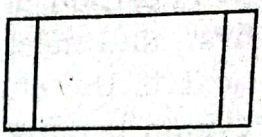


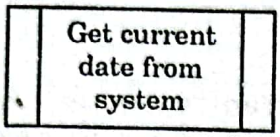
Off-Page Connector Symbol: This symbol is used to connect parts of the flowchart from one page to another.

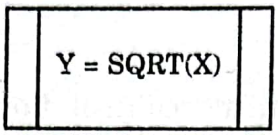


CamScanner

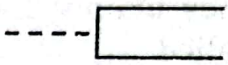
Predefined Process Symbol: This symbol is used to access ("perform" or "call") a subroutine not detailed at that particular point but usually defined in a separate micro flowchart. May be either procedures and functions that are used repeatedly or steps that would overly complicate the flowchart.





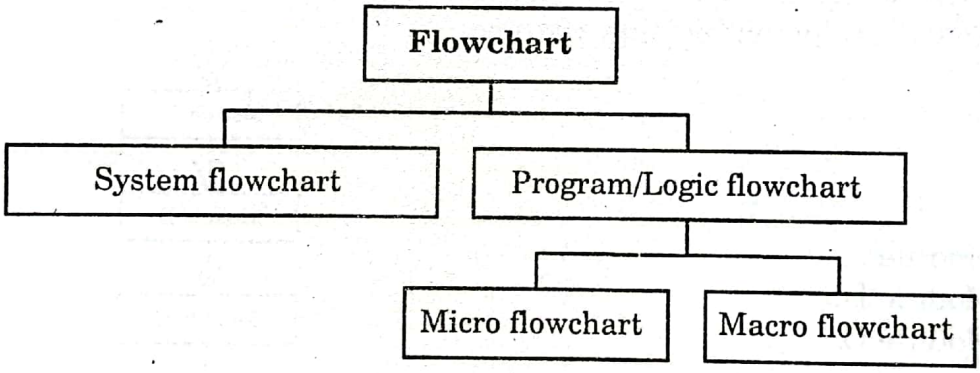


Annotation Symbol: This symbol holds additional descriptive comments, explanatory notes, or clarifications; connected by dashed line to the symbol being commented on. It is used as needed to aid understanding.



Categories of Flowcharts

The flowchart may be categorised as



Note that in this discussion we are talking about the *program/logic flowchart,* a flowchart that represents the flow of logic in a program. This flowchart is further classified as *micro,* a flowchart in which only major events are shown, and *macro,* a flowchart in which all the events (major as well as minor) are shown. A program flowchart is different from a *system flowchart,* which shows the flow of data through an entire computer system.

Flowcharts, which present a map of a solution, were the primary planning device for many years. They were favoured over other methods because logic is easier to follow with pictures than with words. But flowcharts have some drawbacks: They are not easy to change, and they tend to be too detailed. A programmer draws a flowchart, then writes a program based on that flowchart; both the flowchart and the program listing are part of the program documentation. As the program changes—and it will—corresponding changes should be made on the flowchart. And therein lies the problem: Most programmers simply do not keep flowcharts up-to-date. However, there is a new procedure that makes flowcharts more easy to plot: They can be drawn and revised using flowcharting software.

CamScanner

Composition of a flowchart requires consideration of the following four points. The first, and most obvious, is that the objective or problem, must be well defined. The next is the choice of starting point, followed by determination of the sequence of events leading towards the end result. Finally, there will be decisions resulting in more than one possible action.

Control Structures

In a program, *control structures* control how the program executes. Algorithms, flowcharts and their equivalent computer programs are more easily understood if they mainly use self-contained modules and three types of logic, or flow of control, called

1. Sequence logic, or sequential flow.
2. Selection logic, or conditional flow.
3. Iteration logic, or repetitive flow.

These three are considered the basic building blocks of all program construction.

Sequence

The *sequence control structure* is the most straightforward: One module simply follows another in sequence. The sequence may be presented by means of numbered steps or by the order in which the modules are written. Generally most of the problems follow this elementary flow pattern.

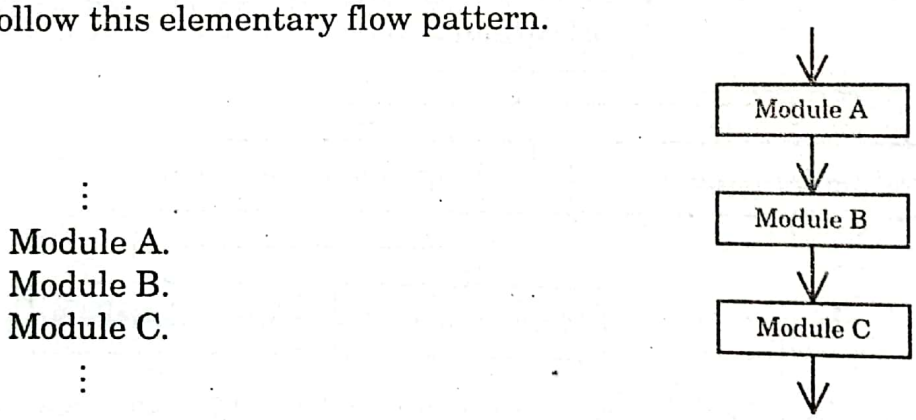


Fig. 2.1. Flowchart general format of sequence control structure

We say that such an algorithm is a sequence of steps, meaning that

(a) At a time only one step is to be carried out.

(b) Every step is to be carried out once and only once; none is repeated and none is omitted.

(c) The order of executing the steps is the same as that in which they are written.

(d) Termination of the last step indicates the end of the algorithm.

Fig. 2.1 shows the general format of a sequence control structure as it is used in flowcharting: The three steps follow in sequence.

Example 2.5

This is the algorithm required to input three numbers from the keyboard of the personal computer and output the result.

CamScanner

1. Read X.
2. Read Y.
3. Read Z.
4. Set SUM = X + Y + Z.
5. Write SUM.
6. Exit.

By: Sir Ahad

This program starts with a declaration of variables followed by an instruction *Read* which tells the computer to take the numbers from the keyboard and store them as the variables X, Y, Z.

The next line instructs the computer to add together the contents of the stores identified by the name *SUM*. Lastly the instruction *Write SUM* outs the answer on the monitor.

Selection

The *selection control structure* employs a number of conditions which lead to a selection of one out of several alternative modules. The structures which implement this logic are called conditional structures or IF structures. The end of IF structure is frequently indicated by the statement "[End of IF structure.]" or some equivalent statement.

This control structure has two forms: IF and IF-ELSE. The IF selection is less complicated: IF the condition is true, then do something—but if it is not true, then do nothing. For example, IF the shift worked is shift 2, then add bonus of Rs. 500/-.

This structure has the form

IF condition, then
[Module A.]
[End of IF structure.]

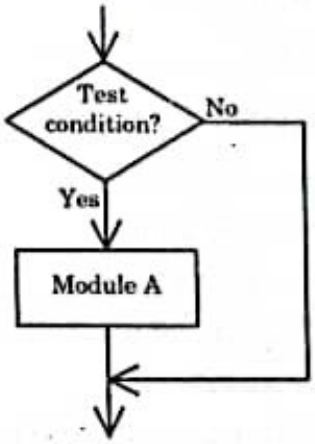


Fig. 2.2. Flowchart general format of IF structure

If the condition holds, then Module A, which may consist of one or more statements, is executed; otherwise Module A is skipped and control transfers to the next step of the algorithm. Fig. 2.2 shows the general format of IF structure in a flowchart.

IF-ELSE is more complicated then IF. The IF-ELSE control structure works as follows: IF (a condition is true), then (do something), ELSE (do something different). For example, IF a student is a resident, then the fee equals number of subjects times Rs. 700/-, ELSE fee is number of subjects times Rs. 900/-.

This structure has the form

CamScanner

IF condition, then
    [Module A.]
ELSE
    [Module B.]
[End of IF structure.]

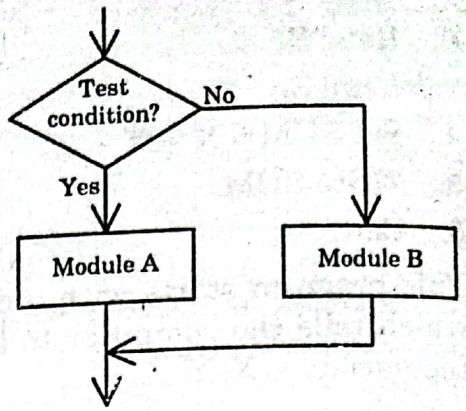


Fig. 2.3. Flowchart general format of IF-ELSE structure

If the condition holds, then Module A is executed; otherwise B is executed. Fig. 2.3 shows the general format of IF-ELSE structure in a flowchart.

Example 2.6

The problem is to input an examination mark and test it for the award of a grade. The mark is a whole number between 1 and 100. Grades are awarded according to the following criteria:

>= 80 distinction

>= 60 merit

>= 40 pass

< 40 fail

The algorithm is

1. Read mark.
2. IF mark >= 80 then
       Write distinction.
   ELSE IF mark >= 60 AND mark < 80 then
       Write merit.
   ELSE IF mark >= 40 AND mark < 60 then
       Write pass.
   ELSE mark < 40
       Write fail.
   [End of IF structure.]
3. Exit.

Iteration

Iteration is often referred to as a loop, because the program will keep repeating the activity until the condition becomes false. Loops simplify programming because they enable the programmer to specify certain instructions only once to have the computer execute them many times. WHILE, DO-WHILE, and FOR are three types of structures involved in loops. Each type begins with a Repeat statement and is

CamScanner

followed by a module, called the *body of the loop*. The end of structure is frequently indicated by the statement "[End of loop.]" or some equivalent statement.

Each type of loop structure is discussed below:

(1) WHILE Loop: The *WHILE loop* uses a condition to control the loop at the beginning. The loop will usually have the form

```
Repeat WHILE condition
    [Module.]
[End of WHILE loop.]
```

Fig. 2.4 shows the general format of WHILE structure in a flowchart.

(2) DO-WHILE Loop: The *DO-WHILE loop* uses a condition to control the loop at the end. The loop will usually have the form

```
DO
    [Module.]
Repeat WHILE condition
[End of WHILE loop.]
```

(3) FOR Loop: The *FOR loop* uses an index variable, such as K, to control the loop. The loop will usually have the form:

```
FOR K = R to S by T
    [Module.]
[End of FOR loop.]
```

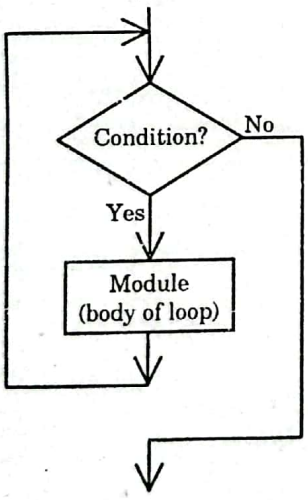


Fig. 2.4. Flowchart format of WHILE loop

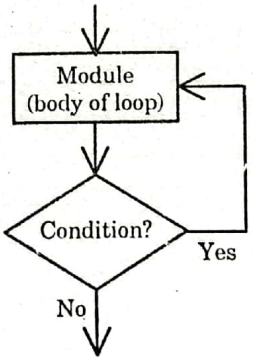


Fig. 2.5. Flowchart format of DO-WHILE loop

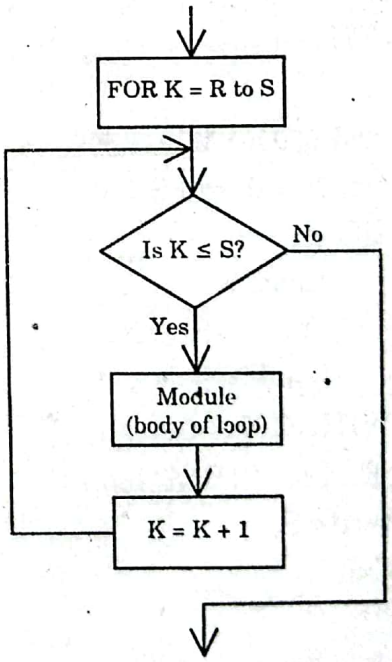


Fig. 2.6. Flowchart format of FOR loop

The WHILE and DO-WHILE loops are used for *indefinite repetitions*. That is, they do not define the number of repetitions that will occur when the loop is executed. They merely give conditions for looping to stop. The FOR loop is used for *definite repetition*. It defines the number of repetitions that will occur during execution. In Fig. 2.6 R is called the initial value, S the end value or test value. Observe that the body of the loop is executed first with K = R, then with K = R + 1, then with K = R + 2, and so on. The cycling ends when K = S.

CamScanner

If we put the loop-ending decision at the beginning, it is called a *leading decision;* if we put it at the end, it is called a *trailing decision.* The position of the decision constitutes the basic difference between WHILE and DO-WHILE structures. As Fig. 2.4 shows, WHILE tests at the beginning of the loop—the diamond-shaped decision box is the first action of the loop process. The DO-WHILE loop tests at the end, as we can see in Fig. 2.5. The DO-WHILE loop, by the way, guarantees that the loop statements are executed *at least once* because the loop statements are executed before we make any test about whether to get out. This guarantee is not desirable, depending on the program logic.

Example 2.7

Write the algorithm to add ten numbers using WHILE, DO-WHILE and FOR loops.

(a) Using **WHILE** loop.

1. Set X = 0 and S = 0.
2. Repeat WHILE X ≤ 10
   Read N.
   Set S = S + N.
   Set X = X + 1.
   [End of WHILE loop.]
3. Write S.
4. Exit.

(b) Using **DO-WHILE** loop.

1. Set X = 0 and S = 0.
2. DO
   Read N.
   Set S = S + N.
   Set X = X + 1.
   WHILE X ≤ 10
   [End of WHILE loop.]
3. Write S.
4. Exit.

(c) Using **FOR** loop.

1. Set S = 0.
2. FOR I = 1 to 10 by 1
   Read N.
   Set S = S + N.
   [End of FOR loop.]
3. Write S.
4. Exit.

CamScanner

Types of Flowchart

There are four main types of flowchart which can be combined in different way to represent any given problem. Each type can be recognised by the features given in the following descriptions.

Sequential Flowchart: This is the simplest type of flowchart, involving no decisions, branches or loops. An example is given in Fig. 2.7, which relates to a diner buying in a restaurant.

Branch (or Jump) Flowchart: In this type, one or more decision boxes are introduced, and the action taken depends upon whether the result of the decision is 'yes' or 'no'. In the example shown in Fig. 2.8, by the introduction of the extra three boxes: the 'leave a tip' operation is jumped over if the 'no' branch is taken from the decision box.

Loop Flowchart: A loop represents a procedure which may be repeated a definite or indefinite number of times. An example is given in Fig. 2.9, where the definition of a loop will be read over and over again until it is fully understood. This may require only one or two cycles, or if the reader has exceptional difficulty, a large number of cycles.

Nested Loops in a Flowchart: Consider the following problem. A man has a number of metallic rods, each of which is greater than two metres in length. Each rod is to be cut into lengths of two metres: the objective is to construct a flowchart which will represent a logical and systematic method of carrying out this task.

The appropriate flowchart is given by Fig. 2.10, where inner loop represents the procedure involved in cutting one rod into several lengths of two metres, and the outer loop represents the procedure involved in taking a new rod for the inner loop to process.

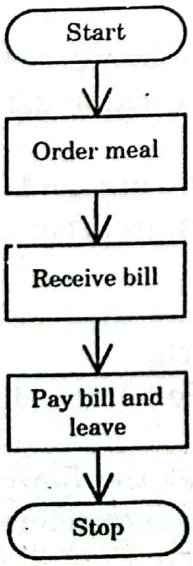


Fig. 2.7. Sequential flowchart

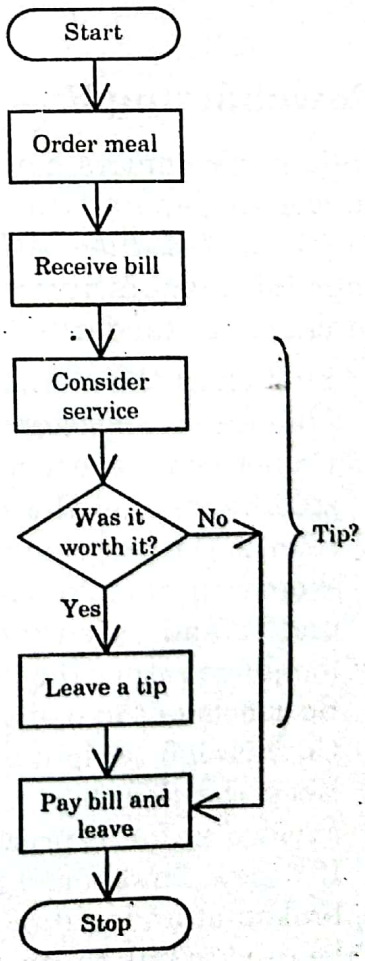


Fig. 2.8. Branch flowchart

CamScanner

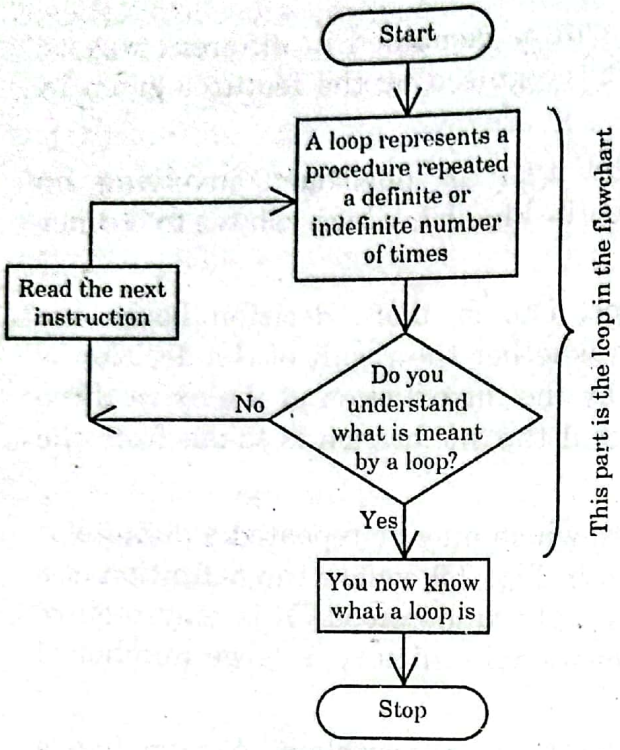


Fig. 2.9. Loop flowchart

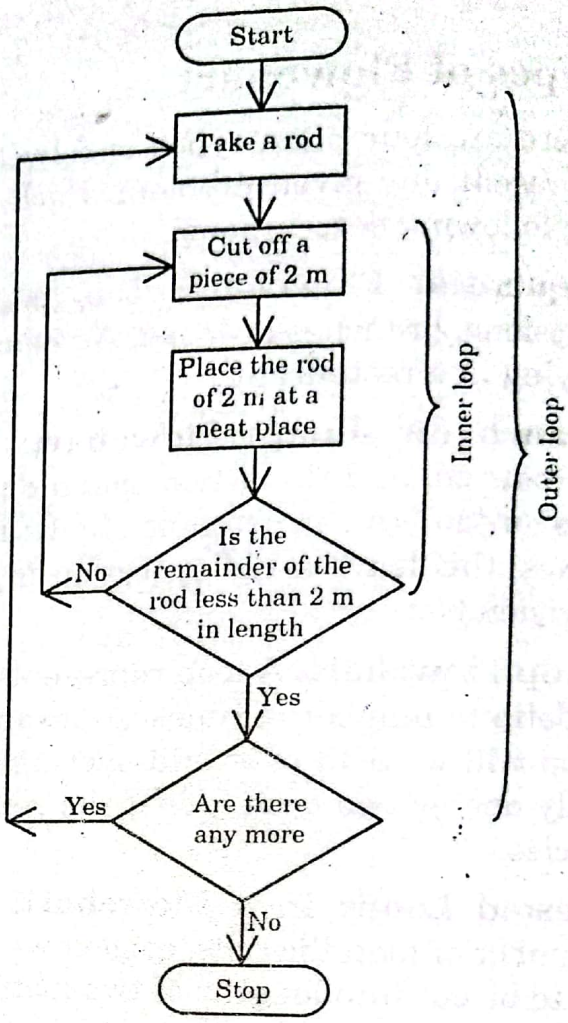


Fig. 2.10. Nested loops in a flowchart

Flowcharting Rules

While programmers have a good deal of freedom in creating flowcharts, there are a number of general rules and guidelines recommended by the *American National Standards Institute* (ANSI) to help standardise the flowcharting process. Various computer manufacturers and data processing departments usually have similar flowcharting standards. Some of these rules and guidelines are as follows:

1. First chart the main line of logic, then incorporate detail.
2. Maintain a consistent level of detail for a given flowchart.
3. Do not chart every detail or the flowchart will only be a graphic representation, step by step, of the program. A reader who is interested in greater details can refer to the program itself.
4. Words in the flowchart symbols should be common statements and easy to understand. It is recommended to use descriptive titles written in designer's own language rather then in machine oriented language.
5. Be consistent in using names and variables in the flowchart.
6. Go from left to right and top to bottom in constructing flowcharts.
7. Keep the flowchart as simple as possible. The crossing of flow-lines should be avoided as far as practicable.
8. If a new flowcharting page is needed, it is recommended that the flowchart be broken at an input or output point. Moreover properly labelled connectors should be used to link the portions of the flowchart on different pages.

CamScanner

Example 2.8

Draw a flowchart to add two numbers.

Solution

See Fig. 2.11.

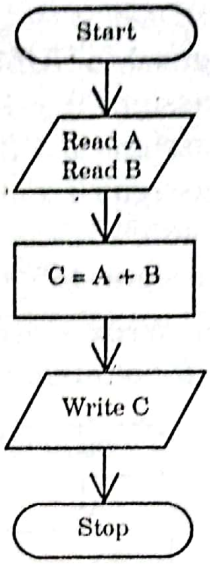


Fig. 2.11

Example 2.9

Draw a flowchart to generate numbers from 1 to 10.

Solution

See Fig. 2.12.

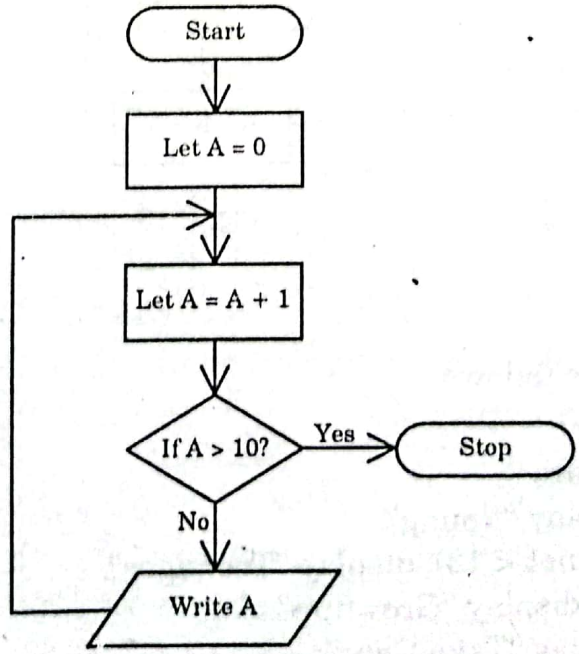


Fig. 2.12

CamScanner

Example 2.10

Draw a flowchart for summing the first thirty integers.

Solution

Fig. 2.13 is the flowchart for summing the first thirty integers. There we use SUM as the accumulator, and the variable K to produce the numbers 1, 2, 3, ..., 30. Observe that

SUM = 0 first assigns 0 to SUM.
When K = 1 SUM = SUM + K assigns 0 + 1 = 1 to SUM.
When K = 2 SUM = SUM + K assigns 1 + 2 = 3 to SUM.
When K = 3 SUM = SUM + K assigns 3 + 3 = 6 to SUM, and so on.
When K = 30, the value of SUM is 435, so
SUM = SUM + K assigns 435 + 30 = 465 to SUM.

This is the value of SUM when the Write statement is executed, so the output of the program is 465, which is the sum of the integers from 1 to 30.

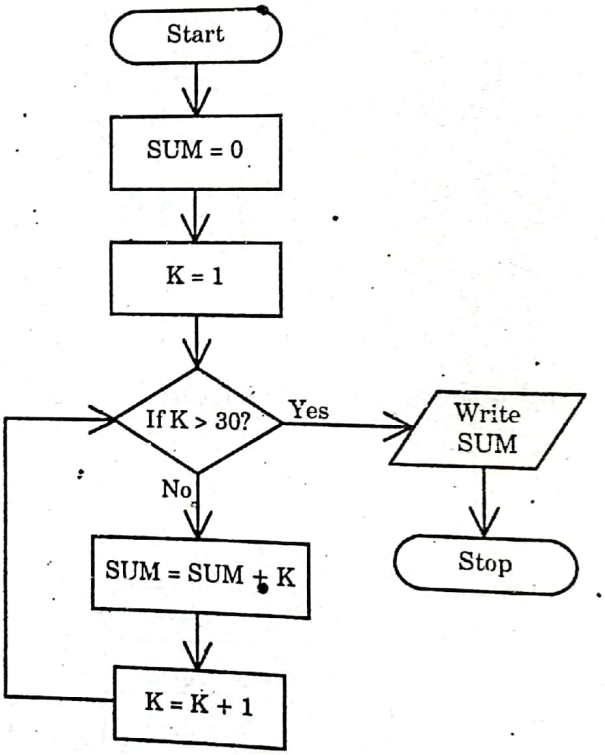


Fig. 2.13

Example 2.11

Draw a flowchart as follows:

- Start.
- Input age in years.
- if age < 13, display "Young".
- if age < 20 (but not < 13), display "Teenager".
- if age not < 20, display "Grownup".
- in the end, display "Take Care".
- Stop.

CamScanner

Solution

See Fig. 2.14.

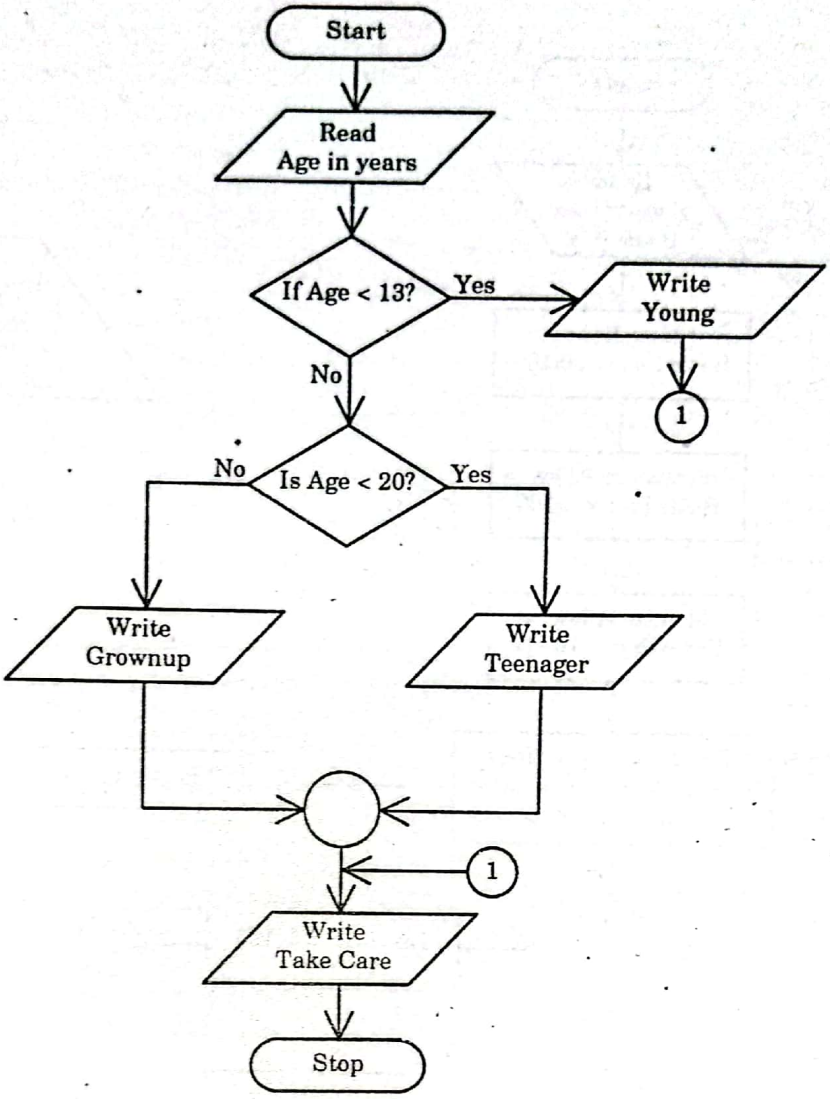


Fig. 2.14

Example 2.12

Compute the Net Pay (NP) of employees of a certain company. Data input for problem is Emp. Name and Basic Pay. Output is required Emp. Name, Basic Pay, Gross Pay (GP), Net Pay (NP), House Rent, Conveyance Allowance, Tax and Medical Allowance. Formula applied here is

1. House Rent = 20% × Basic Pay
2. Conveyance Allowance = 5% × Basic Pay
3. Medical Allowance = 10% × Basic Pay
4. Gross Pay = Basic Pay + House Rent + Conveyance Allowance + Medical Allowance
5. Tax depends upon Gross Pay as:

   If Gross Pay < 2000 then Tax = 0
   If Gross Pay > 2000 then Tax = 2% of Gross Pay
   If Gross Pay > 3000 then Tax = 3% of Gross Pay
   If Gross Pay > 4000 then Tax = 4% of Gross Pay
   If Gross Pay > 5000 then Tax = 5% of Gross Pay

CamScanner

Solution

See Fig. 2.15.
or Fig. 2.16.

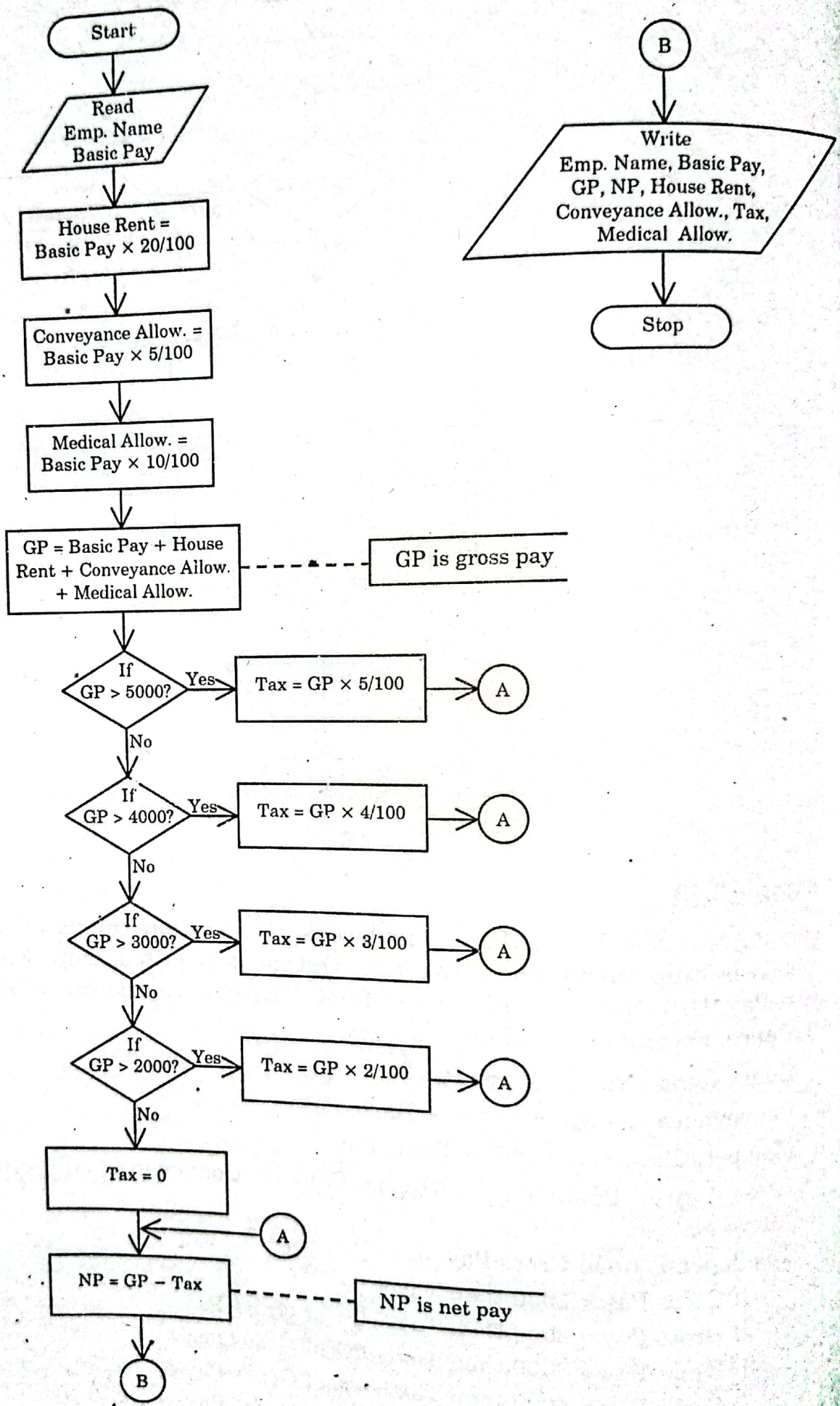


Fig. 2.15

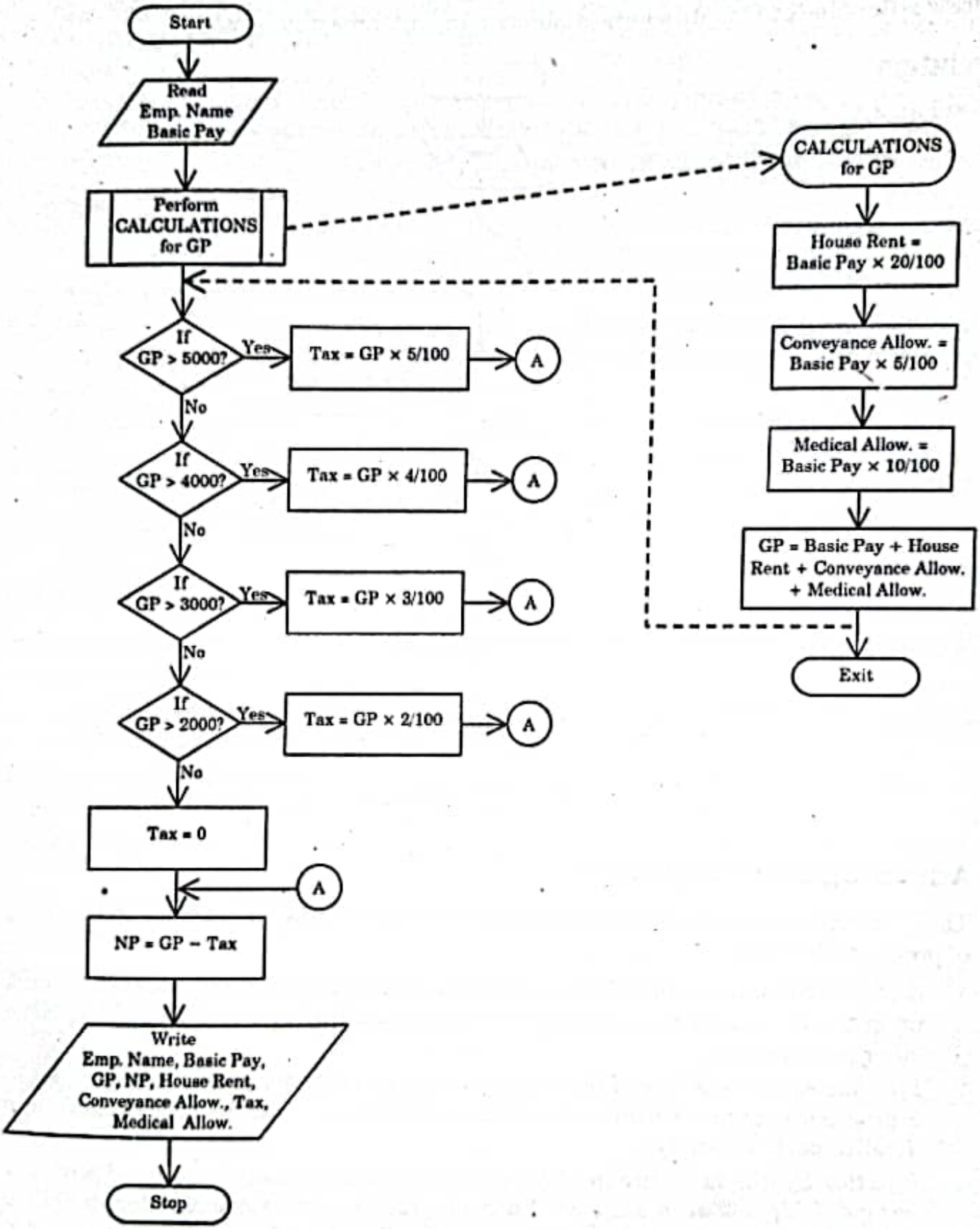


Fig. 2.16

CamScanner

Example 2.13

Draw a flowchart to identify largest number among three numbers?

Solution

See Fig. 2.17.

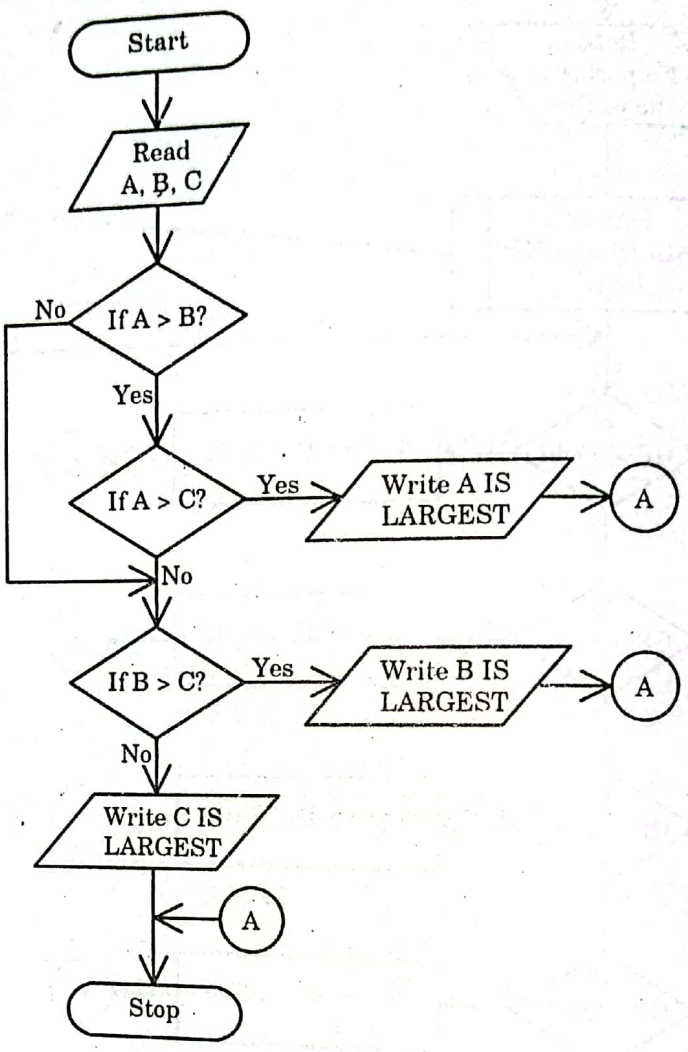


Fig. 2.17

Advantages of Flowcharts

The following advantages may be obtained when flowcharts are used for the purpose of program planning:

1. *Better Communication:* Since, a flowchart is a pictorial representation of a program, it is easier for a programmer to explain the logic of a program to some other programmer.
2. *Effective Analysis:* A macro-flowchart that presents the main lines of logic used in a program becomes a model of a program/system that can be broken down into detailed parts for study.
3. *Effective Synthesis:* A group of programmers are normally associated with the design of big software systems. Each programmer is responsible for designing only a part of the entire system. So initially, if each programmer draws a flowchart for his part of design, the flowcharts of all the programmers can be placed together to visualise the overall system design. Any problem in linking the various parts of the system can be easily detected at this stage and the design can be accordingly modified.

CamScanner

4. *Proper Program Documentation:* Program documentation involves collecting, organising and storing a complete historical record of programs and the other documents associated with a system. Good documentation is needed for the following reasons:
   (a) Documented knowledge belongs to an organisation and does not disappear with the departure (resignation/retirement) of a programmer.
   (b) If projects are postponed, documented work will not have to be duplicated.
   (c) If programs are modified in the future, the programmer will have a more understandable record of what was originally done.

   The flowchart obviously provide a valuable documentation support.
5. *Efficient Coding:* Once a flowchart is ready, programmers find it very easy to write the concerned program. Flowcharts act as a guide during program writing stage.
6. *Systematic Debugging:* Even after taking full care in program design, some errors may remain in the program. It is easier to find out the reason for the error by going through the logic of the program in flowchart form. A flowchart is very helpful in detecting, locating, and removing errors (bugs) in a program in a systematic manner.

Disadvantages of Flowcharts

In spite of their many obvious advantages, flowcharts have several disadvantages which are as follows:

1. Flowcharts are very time consuming and laborious to draw with proper symbols especially for large complex programs.
2. Flowchart tend to take up a lot of space, filling multiple sheets of paper to capacity. This can make them difficult to read and understand.
3. Owing to the symbol-string nature of flowcharting, any changes or modifications in the program logic will usually require a completely new flowchart.
4. There are no standards determining the amount of detail that should be included in a flowchart.

Because of these disadvantages, many organisations are now reducing the amount of flowcharting used.

CamScanner

Exercise

2.1. What is an algorithm?

2.2. What are the three control structures used in coding?

2.3. Explain the difference between the two loop (iteration) structures, WHILE DO-WHILE.

2.4. Write algorithm showing the logic required to calculate and print the sale on a purchase of four items.

2.5. Write the algorithm for each of the following tasks listed sequentially:

(a) Write algorithm to generate numbers from 1 to 10. (using WHILE loop

1. Set X = 0.
2. WHILE X < 10
   Set X = X + 1.
   Write X.
   [End of WHILE loop.]
3. Exit.

(b) Write algorithm to generate numbers from 1 to 10. (using IF-E structure).

1. Set X = 0.
2. Set X = X + 1.
3. Write X.
4. IF X = 10, then
   Exit.
   ELSE
   Goto step 2.
   [End of IF structure.]

(c) Write algorithm to write any number 10 times.

1. Read N.
2. FOR K = 1 to 10 by 1
   Write N.
   [End of FOR loop.]
3. Exit.

(d) Write algorithm to add numbers till input zero is given.

1. Set S = 0.
2. WHILE N $\neq$ 0
   Read N.
   Set S = S + N.
   [End of WHILE loop.]
3. Write S.
4. Exit.

CamScanner

(e) Write algorithm to write odd numbers from 1 to 11.

1. FOR K = 1 to 11 by 2
   Write K.
   [End of FOR loop.]
2. Exit.

(f) Write algorithm to take sum of array of 10-elements (first take input and then add).

1. FOR K = 1 to 10
   Read N[K].
   [End of FOR loop.]
2. Set S = 0.
3. FOR K = 1 to 10
   Set S = S + N[K].
   [End of FOR loop.]
4. Write S.
5. Exit.

2.6. What is a flowchart and how may types of flowcharts are there?

2.7. What are the various symbols used in flowcharting? Give their pictorial representation.

2.8. Describe the function of the various flowcharting symbols.

2.9. Determine the appropriate flowchart symbol in which each of the following instructions or questions should be placed:

(a) Read A, B, C
(b) Is A = 0?
(c) Add B to C giving A
(d) Start
(e) Is A is less than B?

Ans.

(a) Parallelogram, since it is an input instruction.
(b) Diamond, since it asks a question.
(c) Rectangle, since it is a process (calculation).
(d) Oval.
(e) Diamond, since it asks a question.

2.10. Determine the minimum number of arrows connected to a diamond in a flowchart.

Ans. There must be at least one arrow going to the diamond, and at least two arrows leaving the diamond; hence there are at least three arrows connected to a diamond.

2.11. Why are there standards for the symbols used in drawing flowcharts?

2.12. How does a flowchart help a programmer in program development?

2.13. What are the various guidelines to be followed while drawing a flowchart?

CamScanner

2.14. What are the advantages of flowcharts when they are used to solve a problem?

2.15. What are the disadvantages of flowcharts?

2.16. How can a counter be used to keep track of the number of times a loop has been executed?

2.17. Is it possible to have more than one flowchart for a given problem? Give reasons for your answer.

2.18. Three numbers, denoted by the variables A, B, and C are supplied as input data. Draw a flowchart for the logic to pick and print the smallest of the three numbers.

2.19. Analyse the flowchart shown in Fig. 2.18 and list the final output.

2.20. Consider the flowchart in Fig. 2.19. Find the output if the input is (a) A = 5, B = 10, C = 15 and (b) A = 15, B = 10, C = 5.

Ans.

(a) Since A is less than B, BONUS = 100 is executed, and 100 is outputted.

(b) A is greater than B, and B is greater than C; so BONUS = 300 is executed, and 300 is outputted.

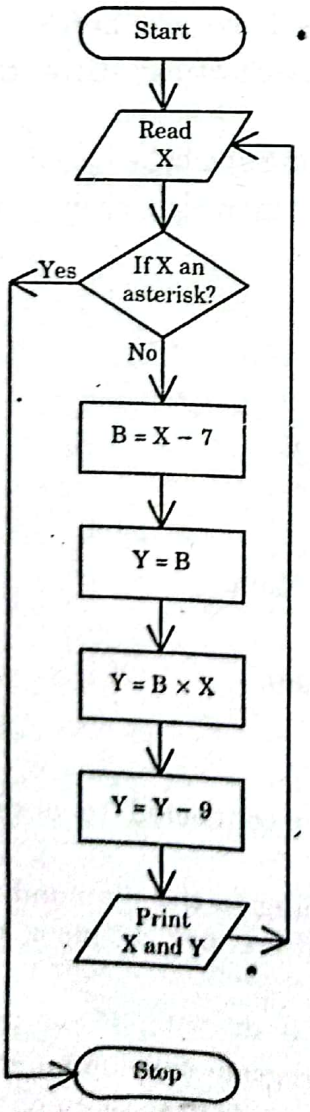


Fig. 2.18

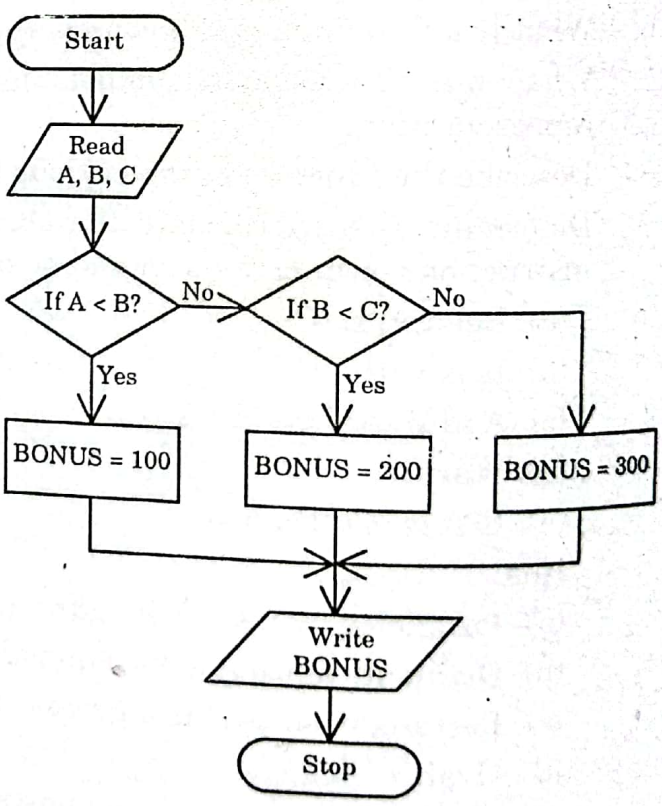


Fig. 2.19

2.21. The first 20 records in a data set are to be read and printed. Draw a flowchart to do this job. Make sure that the processing stops after the twentieth record.

2.22. Draw a flowchart for the logic to find out whether a given triangle ABC is a right-angled triangle. Assume that the sides of the triangle are supplied as input data. Print the answer as yes or no.

2.23. Consider Fig. 2.20. Find the output if the input is (a) A = 5, B = 10; (b) A = 20, B = 10.

Ans.

(a) 40; (b) 40

2.24. Consider the flowchart in Fig. 2.21. Find the output if the input is (a) A = 5, B = 10, C = 20; (b) A = 30, B = 10, C = 20.

Ans.

(a) 40; (b) 70

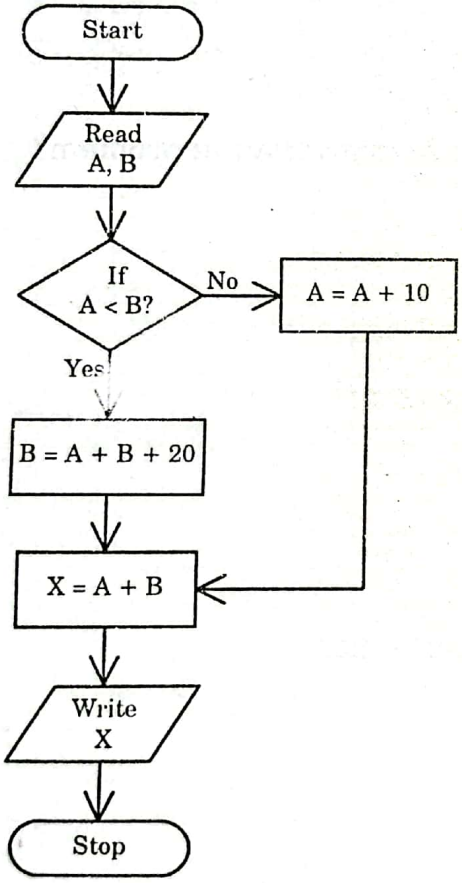


Fig. 2.20

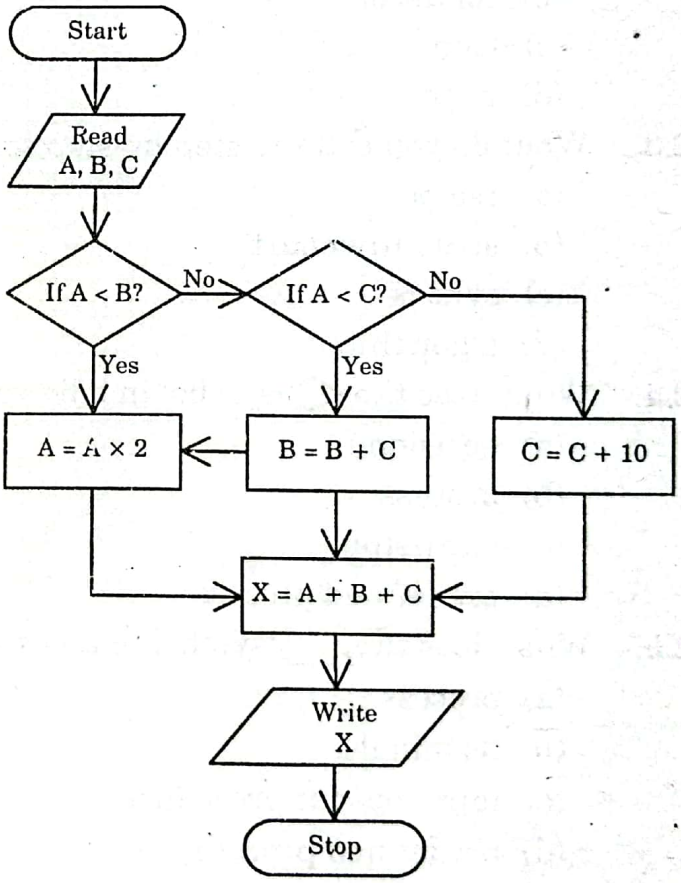


Fig. 2.21

2.25. Draw a flowchart that calculates:

$$\text{SUM} = \frac{1}{1} + \frac{1}{3} + \frac{1}{5} + \ \ldots \ + \frac{1}{25}$$

2.26. Consider the equation $y = x^4 + 2x^3 - 4x^2 - 16x + 32$. Draw a flowchart that calculates y for values of x from -4 to 4 in step of 0.25, and write each x and its corresponding y.

CamScanner

Objectives and MCQs

Choose the answer that best completes the statement.

2.1. A pictorial representation of a problem is called
- (a) a flowchart
- (b) an algorithm
- (c) a pseudocode
- (d) None of the above

2.2. Which of the following is *not* one of the three program logic constructs?
- (a) simple sequence
- (b) condition
- (c) loop
- (d) go to

2.3. What do you call the step-by-step solution to a programming problem?
- (a) recipe
- (b) structure chart
- (c) syntax
- (d) algorithm

2.4. What does the symbol in a flowchart represent?
- (a) decision
- (b) process
- (c) start/stop
- (d) predefined process

2.5. What does the symbol in a flowchart represent?
- (a) process
- (b) terminal
- (c) input/output procedure
- (d) predefined process

2.6. Which of the following control structures use DO-WHILE?
- (a) sequence
- (b) selection
- (c) iteration
- (d) None of the above

CamScanner

Objectives

2.1. A pictorial representation of a problem is made by a flowchart.

2.2. In micro flowchart only major events are shown.

2.3. In macro flowchart all the events (major as well as minor) are shown.

2.4. The oval symbol is used to represent the beginning of a program.

2.5. The rectangle symbol in a flowchart represents process.

2.6. At least two arrows leave the diamond in a flowchart.

2.7. The selection control structure is frequently shown in algorithm as an IF-ELSE statement.

2.8. Match the following symbols with their functions.

(a) Oval	(a) Decision
(b) Parallelogram	(b) Connector
(c) Circle	(c) Terminal
(d) Rectangle	(d) Input/output of data
(e) Diamond	(e) Process

CamScanner

CHAPTER 3

AN OVERVIEW OF 'C' LANGUAGE

By: Sir Ahad

Introduction to C

C is a powerful programming language developed by Dennis Ritchie at Bell (now AT & T's Bell) Laboratories of USA in 1972.

C evolved from two earlier languages: BCPL (Basic Combined Programming Language) and B. BCPL was developed in 1967 by Martin Richards at the Cambridge University as a language for writing operating systems software and compilers. In 1970 Ken Thompson at Bell Laboratories developed a language based on BCPL and called it B, the first letter of BCPL. Dennis Ritchie developed B further into a new language and gave it the name C, the next letter of BCPL.

The C language was a result of the way software developers used the basic ideas of structured programming. These ideas include explicit data typing, which means that all variables must be given a specific data type from a set of predefined choices; control structures without GOTOs, such as the IF-ELSE and the WHILE loop and block-structured programming allowing easier program deigns and debugging.

By the late 1970s, C had evolved into what is now referred to as "traditional C". The publication in 1978 of Brian Kernighan and Dennis Ritchie's book, *The C Programming Language,* brought wide attention to the language. This publication become one of the most successful computer science books of all time.

The rapid expansion of C for various *platforms* (computer systems) led to many

CamScanner

similar but often incompatible variations. This was a serious problem for software developers who needed to create *portable code* that could be used on different machines. In 1983 a subcommittee was formed by the American National Standards Institute (ANSI) Committee on Computers and Information Processing to "provide an unambiguous and machine-independent definition of the language". Consequently, the ANSI Committee 1989 developed a standardised definition of the C language. Virtually all commercial C compilers and interpreters now adhere to the ANSI standard. Many also provide additional features of their own.

C is traditionally called the development language of the UNIX operating system. Today, however, C is becoming a popular choice for developing other operating systems and virtually all new major operating systems are written in C and/or C++. It is also becoming a common application software programming language and many software packages from major software vendors are written in C. These include word processing, database management and graphics software. Over the past two decades, C has become available for most computers. C is hardware independent. With careful design, it is possible to write C programs that are *portable* to most computers.

In 1983, C++ was developed by Bjarne Stroustrup at the Bell Laboratories, and is basically a superset of C. Hence all standard C features are available within C++. However, C++ is not merely an extension of C. Rather it incorporates several new fundamental concepts that form a basis for object-oriented programming (OOP)—a new programming paradigm that is of interest to professional programmers. C++ mainly developed to provide a programming tool, for designing complex software projects, having the following characteristics:

1. Be helpful in reducing the complexity while solving problems.
2. Ensure correctness of results.
3. Be affordable in terms of hardware and resources.
4. Be open—easy and cheap to integrate existing software libraries and facilities.
5. Be portable.

The rapid expansion of C++ over various types of computers prompted many software companies to release their own complier of C++. *Borland International Inc., USA* released version 3 of C++ in 1992 and called its complier as "Turbo C++". Borland's Turbo C++ provides the best platform yet devised for learning C. Its Integrated Development Environment (IDE) combines all the features needed to develop a C program—editor, compiler, linker, debugger and help system—into a single, easy-to-use screen display. This enables you to develop programs far more simple and conveniently than you can using the traditional command-line compiler system. Turbo C++ is also very fast and surprisingly inexpensive and in this book we will use Turbo C++ complier for learning C language.

Why Learn C?

C has many advantages over other programming languages. Among them are:

1. It is an efficient language, suitable for applications that run on personal computers, microcomputers, minicomputers, workstations, and other small machines.

CamScanner

2. Its large variety of operators and commands can be used to write anything from operating systems to accounting packages. In fact, many of the C compilers on the market today were written in C.
3. It is capable of handling a wide variety of systems applications.
4. C is a portable language. A C program written on one computer (*e.g.* IBM-PC) can be run with little or no modification on any other computer (*e.g.* Apple Macintosh) with a C compiler.
5. Like assembly language, C enables programmers to have extensive control over computer hardware, but because C uses English like statements, which are easy to read, it is often classified as a high-level language.
6. It is a block-structured language, in which you can replace elementary units with entire blocks, thus allowing-modular development and easy expansion of programs. This block approach makes program debugging and maintenance easier and makes C compatible with structured methods and software engineering.
7. It is flexible. By creating your own function modules, you can make C do just about anything you want. If you are so inclined, you can create your own language in C!
8. It has high execution speed. C is said to be "closer" to assembly language than other high-level languages because some C commands directly address the physical hardware of the computer. Situations arise where execution speed is critical; sorting is a common example. Coding the program in assembly language was the usual solution. The problem was that, as new Central Processing Unit (CPU) chips were introduced, the programmer was forced to learn a new instruction set for each CPU. Retraining costs were so high that many commercial software houses switched to C language. With a good C compiler, the difference in execution speed between C and an assembler will go unnoticed in all but the most demanding cases.
9. Many companies such as *Lotus* and *Microsoft* use it for developing their software packages. Many of popular applications, such as Word Perfect and Lotus 1-2-3 which were originally written in assembly language have been converted to C.
10. C is the standard programming language in many industries and applications. If you read job.advertisements in a major newspapers, you will find that there are more openings for C programmers than for FORTRAN or COBOL programmers, with languages as Pascal and PL/1 rarely find being mentioned.

The IDE of Turbo C++ Compiler

To start *Turbo C++*, just type

```
C:\TC\BIN\TC
```

and press **Enter**↵.

The blank screen of Integrated Development Environment (IDE) is displayed. Turbo C++ IDE consists of menu bar, windows, status line, and quick reference keys. Fig. 3.1 shows the 10 options in the menu bar. You can activate the menu bar by pressing F10 or using mouse. Each of the main menu bar

CamScanner

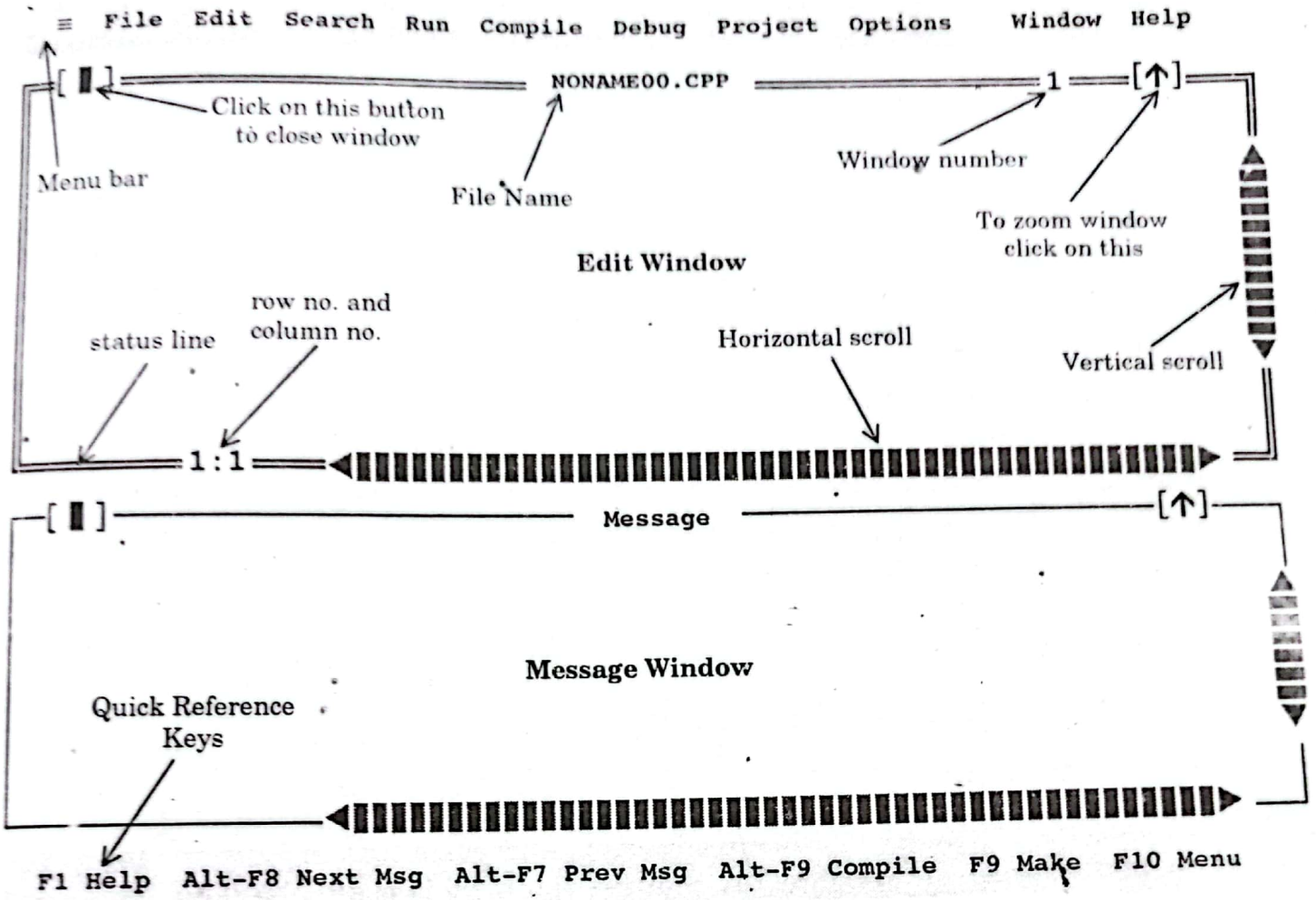


Fig. 3.1. Turbo C++ version 3.0 Integrated Development Environment (IDE).

You can invoke pull-down menu one by one by moving the cursor to it and pressing Enter or by just typing its first letter (for example, F for File or R for Run). The cursor is initially on *File* at the far left. You can also invoke each of the pull-down menus by pressing *Alt+<highlighted character>* keys.

Look at the bottom of the screen to know what function keys do. Borland calls it the *Quick-Reference* line. Pressing F1 invokes help, and F10 switches between the main screen and the menu bar.

Immediately below the menu bar is a status line. It indicates the current cursor position in lines (rows) and columns. The starting (or home) position is row 1, column 1 in the top left corner. The status line also contains mode information and the name of the current file at the top.

Initially, the name will be "NONAME00.CPP" which is *Turbo C++*'s default. The "CPP" (which means C plus plus) extension indicates a C++ language (source) program.

Options of File Menu

The File menu contains commands for manipulating files. It contains the following commands:

CamScanner

New	New opens a new Edit window with the default name NONAMExx.CPP (the xx stands for a number from 00 to 31) and automatically makes the new Edit window active.
Open (F3)	Open lets you load an existing file into an edit window.
Save (F2)	The Save command saves the file that is in the active edit window to disk.
Save as	Save as lets you save the current file under a different file name, in a different directory, or on a different drive.
Save all	The Save all command works just like the Save command except that it saves the contents of all modified files, not just the file in the active edit window.
Change dir	Change dir lets you choose the current directory and drive.
Print	The Print command prints the contents of the active edit, output, or message window.
DOS shell	DOS shell temporarily exits Turbo C++ and displays DOS prompt. To return from the DOS prompt back to Turbo C++, type *exit* and press ↵.
Quit	The Quit command exits Turbo C++.

Options of Edit Menu

The Edit menu provides commands to cut, copy, and paste text in Edit windows. You can also undo changes and reverse the changes you have just undone. You can open a Clipboard window to view or edit its contents, and copy text from Message, Output, and Help windows.

Undo (Alt+BkSp)	The Undo command takes back the last editing command you performed on a line.
Redo (Shift+Alt +BkSp)	The Redo command reverses the effect of the most recent Undo command. Redo has an effect only immediately after an Undo command or another Redo command.
Cut (Shift+Del)	The Cut command removes the selected text block from your document and places it on the Clipboard. Once you have Cut a text block, you can use *paste* command to paste that text somewhere else in the same or a different window.
Copy (Ctrl+Ins)	The Copy command copies a selected text block and puts a copy on the Clipboard. Once you have copied a text block, you can use *paste* command to copy that text somewhere else in the same or a different window.
Paste (Shift+Ins)	The Paste command puts text from the Clipboard into the current window. Before you can use the Paste command, you must first put text on the Clipboard using the Cut or Copy command.
Clear (Ctrl+Del)	It works like the Cut command except that it does not place the selected text on the Clipboard.

CamScanner

Copy example	The Copy example command copies the text of the pre-selected example from the current Help window to the Clipboard. You can then paste the example into an edit window and run it to see how the function being demonstrated in that example works.
Show clipboard	The Show clipboard command opens the Clipboard window, so you can see the current Clipboard contents.

Options of Search Menu

The Search menu provides commands to search for text, function declarations, and error locations in your files.

Find	The Find command displays a dialog box, where you type in the text you want to search for. You can set several options in this dialog box that affect the search.
Replace	The Replace command displays a dialog box, where you type in the text you want to search for and text you want to replace it with.
Search again (Ctrl+L)	The Search again command repeats the last Find or Replace command, using any settings made in the last Find or Replace dialog box.
Go to Line Number	Go to line number command displays a specific line number that you want to see. Turbo C++ displays the current line number and column number in the lower left corner of every edit window.
Previous error (Alt+F7)	The Previous error command moves the cursor to the location of the previous error or warning message. This command only works if the Message window contains messages.
Next Error (Alt+F8)	The Next error command moves the cursor to the location of the next error or warning message. This command only works if the Message window contains messages.
Locate function	The Locate function command displays a dialog box, where you enter the name of a function to search for. This command is available only during a debugging session.

Options of Run Menu

The Run menu provides commands to run your program and to start and end debugging sessions.

Run (Ctrl+F9)	The Run command lets you execute a program that appears in the current window. This command will compile, link, and run your program.
Program reset (Ctrl+F2)	The Program reset command stops the program from continuing while you are using the Turbo C++ integrated debugger. It also closes any open files that your program was using.
Go to cursor (F4)	The Go to cursor command works while you are using the Turbo C++ integrated debugger. This command lets you run your program from the current highlighted line to the location of the cursor.

CamScanner

Trace into (F7)	The Trace into command runs your program line-by-line. When the debugger reaches a function call, the debugger displays the lines of the called function. Upon exiting the function, the debugger displays the main program.
Step over (F8)	The Step over command runs your program line-by-line, like the Trace into command. The difference is when the debugger reaches a function call, it does not displays the lines of the called function.
Arguments	Arguments lets you define any command-line arguments for your program to use as if you had typed them from the DOS command line.

Options of Compile Menu

You use the commands on the Compile menu to compile the program in the active Edit window, or to make or build your project.

Compile (Alt+F9)	The Compile [Compile] command compiles your Turbo C++ programs and projects from .CPP file to an .OBJ file. A project is simply a collection of related programs that work together.
Make (F9)	The Make command creates an executable .EXE file.
Link	The Link command takes the current .OBJ or .LIB file and links them to produce an executable .EXE file.
Build all	The Build all command rebuilds all the files in your project regardless of their date.
Information	The Information command displays the Information box, which lists statistics about the current program.
Remove messages	The Remove messages command clears all messages from the Message window.

Options of Debug Menu

The Debug menu commands let you debug or remove errors in Turbo C++ programs.

Inspect (Alt+F4)	Inspect lets you examine and modify the contents of a data element, such as a char, pointer, array, function, class, or structure.
Evaluate/ modify (Ctrl+F4)	The Evaluate/modify command leads to the Evaluate and Modify dialog box where you examine and modify the value of a variable or other data item.
Call stack (Ctrl+F3)	The Call stack command lists the sequence of functions your program called.
Watches	The Watches command displays a menu for four watch point commands: (1)*Add watch* lets you examine the contents of a variable while running your program with Turbo C++ integrated debugger. (2)*Delete watch* removes any displayed watch points created with the Add watch command. (3)*Edit watch* lets you edit a watch point created with the Add watch command. (4)*Remove all watches* deletes all watch points created with the Add watch command.

CamScanner

Toggle breakpoint (Ctrl+F8) Toggle breakpoint lets you set a breakpoint or remove an existing breakpoint. A breakpoint is a line in your program where your program stops running until you give the command to continue by pressing the F7 key (Trace into), F8 key (Step over), or both Ctrl and F9 keys together (Run).

Breakpoints The Breakpoints command displays a list of all breakpoints, their line numbers, and their conditions. This command lets you quickly view and delete any breakpoints you have set in your program.

Options of Project Menu

The Project menu commands let you manage your Turbo C++ projects. A *project* is simply a collection of separate files that work together to create a single program.

Open project Open Project lets you create a new project or load in an existing project. Project files contain information needed to create an .EXE program file.

Close project Close project removes the current project and return to the default project, TCDEF.PRJ.

Add item The Add item command lets you add a file to a project.

Delete item The Delete item command lets you delete files from a Project.

Local options The Local options command displays an Override Options dialog box that defines options for a project. The three available options are:

1. include command-line options for a specific project-file module.
2. give a specific name and path for the object file.
3. choose a translator for the module.

Include files The Include files command lists the Include files the current project file uses.

Options of *Options* Menu

The Options menu commands lets you view and change various default settings in Turbo C++ IDE.

Application The Application command lets you select initial settings that affect the compilation and linking of DOS executables.

Compiler The Compiler command displays a menu that lets you specify how Turbo C++ creates optimised code from your program. Until you are familiar with Turbo C++, you should use the default settings.

Transfer The Transfer command lets you add or delete programs to the System menu.

Make The Make command lets you specify conditions that will stop the making of a project file.

Linker The Linker command lets you define how Turbo C++ links your objec files with extension .OBJ together.

CamScanner

Librarian The Librarian command lets you choose options that affect the built-in Librarian. The built-in Librarian combines the .OBJ files in your project into a .LIB file.

Debugger The Debugger command let you define how the integrated debugger works.

Directories This command specifies in which directory Turbo C++ can find *include* and *library* files. This command also lets you specify the directory in which to save program files.

Environment The Environment command lets you specify the appearance of the Turbo C++ integrated environment, how the integrated debugger behaves, when Turbo C++ will automatically save your files, how the editor works, and how a mouse works with Turbo C++.

Save The Save command saves any options you set so that Turbo C++ will automatically use them the next time you run Turbo C++.

Options of Window Menu

The Window menu commands let you define the appearance of the Turbo C++ integrated environment windows. It contains commands for changing the appearance of the Turbo C++ integrated development environment windows.

Size/Move (Ctrl+F5) The Size/Move command lets you change the size and position of a window.

Size: To change the size of the active window:

1. Choose Window [Size/Move].
2. Press and hold [⇧ Shift].
3. Press an arrow key.
4. When it is the size you want, press [Enter ↵].

If a window has a Resize corner, you can drag that corner, or any other corner, to resize the window.

Move: To move the active window:

1. Choose Window [Size/Move].
2. Press an arrow key.
3. When the window is where you want it, press [Enter ↵].

You can also move a window by dragging its title bar.

Zoom (F5) The Zoom command resizes the current window to the maximum size. If the window is already zoomed, this command restores it to its previous size.

Tile The Tile command tiles all open windows so they fit on the screen. With tiled windows, no window will overlap another window.

Cascade The Cascade command displays windows so they may overlap one another.

CamScanner

Next (F6)	The Next command makes the next window active.
Close (Alt+F3)	The Close command closes the active window. You can also click the Close box in the window's upper left corner to close it.
Close all	The Close all command closes all windows and clears all history lists. This command is useful when you are starting a new project.
Message	The Message opens the Message window. If a Message window already appears, this command will make the Message window the active window. The Message window displays status, warning, and error messages when Turbo C++ compiles and links.
Output	Output opens a window that shows the output of your program.
Watch	Watch opens a Watch window and makes it active. The Watch window displays variables and their values.
User screen (Alt+F5)	User screen command displays a full screen output of your program (the IDE becomes temporarily invisible). You can view both text and graphics output on the User screen.
Register	Register displays the CPU registers and their contents to help you debug your program.
Project	Project lists all the files in your current project.
Project notes	Project notes command lets you write ideas, lists, or notes to yourself about each project.
List all (Alt+0)	The List all command displays all the files currently open.

Options of Help Menu

This command brings up the Help menu, which lets you get more information about using the Turbo C++ integrated environment or in writing Turbo C++ programs.

Contents	Contents contains a complete list of topics available for more help.
Index (Shift+F1)	Index lists key words so that you can get help on a specific topic or Turbo C++ command.
Topic search (Ctrl+F1)	When you are in the edit window, you can place the cursor on a word and choose Help [Topic search] (or press Ctrl+F1) to get language help on that particular word.
Previous Topic (Alt+F1)	When you are in the Help system, Previous topic displays the last Help window you opened. You can back up through the last 20 help screens.
Help on Help	Once you are in the Help system, pressing F1 again pop up the Welcome to Online Help screen. You can use Help on Help which explains how to use the Turbo C++ help system.
About	It displays Turbo C++ copyright and version information.

CamScanner

Preparing to Program

C, perhaps more than other languages, demands that the programmer design the program before writing it. Complex problems, however, such as the ones professional programmers are challenged with every day, do require design, and the more thorough the design, the more likely it is that the program will solve the problems it is designed to solve, on time and on budget. A good design also makes a program relatively bug-free and easy to maintain. It has been estimated that nearly 90 percent of the cost of software is the combined cost of debugging and maintenance. To the extent that good design can reduce those costs, it can have a significant impact on the bottom-line cost of the project.

The *first question* you need to ask when preparing to design any program is, "What is the problem I am trying to solve?" Every program should have a clear, well-articulated goal, and you will find that even the simplest programs in this book do so.

The *second question* every good programmer asks is, "Can this be accomplished without writing a custom software?" Reusing an old program, or buying software from store is often a better solution to a problem than writing something new. The programmer who can offer these alternatives will never suffer from lack of work; finding less-expensive solutions to today's problems will always generate new opportunities later.

Assuming you understand the problem, and it requires writing a new program, you are ready to begin your design.

Your Development Environment

Your compiler of Turbo C++ has its own built-in text editor. The files you create with your editor are called *source files,* and for Turbo C++ they typically are named with the extension `.CPP`.

Compiling the Source Code

Although the source code in your file is somewhat strange, and anyone who doesn't know C will struggle to understand what it is for, it is still in what we call human-readable form. Your source code file is not a program, and it can't be executed, or run, as a program can.

To turn your source code into a program, you compile it. To compile a program using Turbo C++ use Compile submenu or hold Alt and press F9.

After your source code is compiled, an *object file* is produced. This file is often named with the extension `.OBJ`. This is still not an executable program, however. To turn this into an executable program, you must run your linker.

Creating an Executable File with the Linker

C programs are typically created by linking together one or more OBJ files with one or more libraries. A library is a collection of linkable files that were supplied with compiler. Turbo C++ compiler comes with a library of useful functions (or procedures)

CamScanner

and classes that you can include in your program. A *function* is a block of code that performs a service, such as adding two numbers or printing to the screen. A class is a collection of data and related functions.

The steps to create an executable file are

1. Create a source code file, with a `.CPP` extension.
2. Compile the source code into a file with the `.OBJ` extension.
3. Link your OBJ file with any needed libraries to produce an executable program.

The Development Cycle

If every program worked the first time you tried it, that would be the complete development cycle: Write the program, compile the source code, link the program, and run it. Unfortunately, almost every program, no matter how trivial, can and will have errors, or bugs, in the program. Some bugs will cause the compile to fail, some will cause the link to fail, and some will only show up when you run the program.

Whatever type of bug you find, you must fix it, and that involves editing your source code, recompiling and re-linking, and then rerunning the program. This cycle is represented in Figure 3.2, which shows the steps in the development cycle.

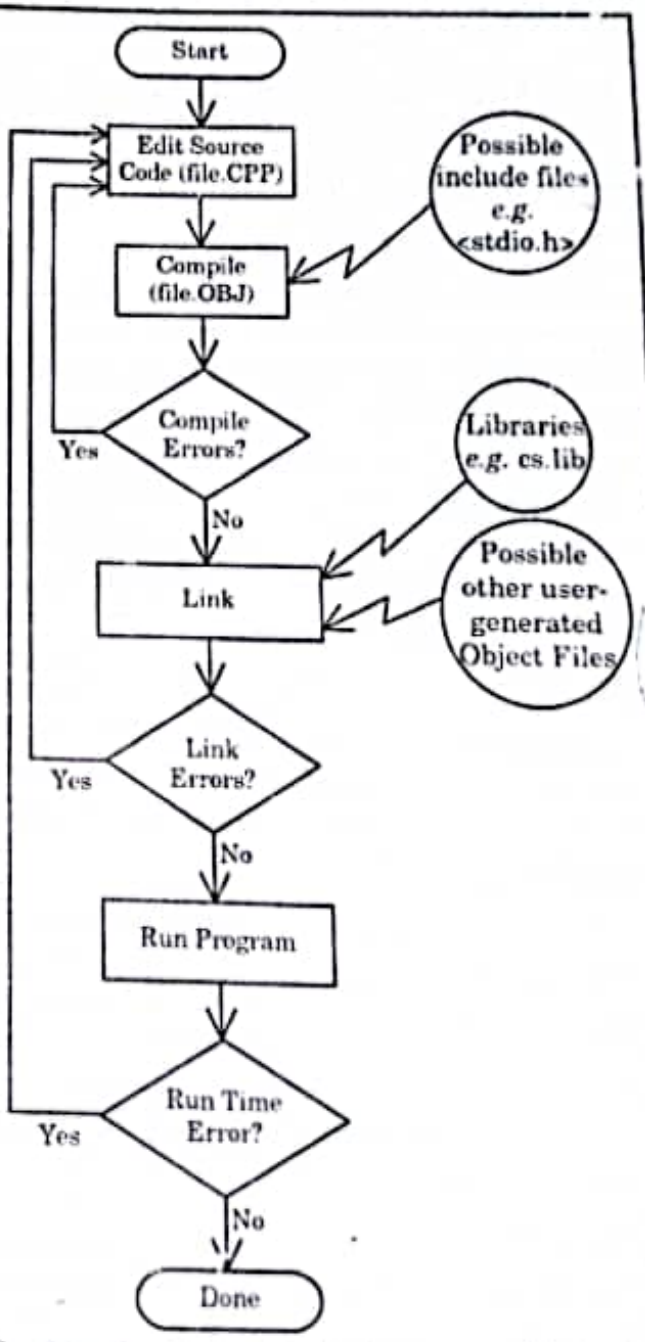


Fig. 3.2. The steps in the development of a C program using Turbo C++ compiler

The Basic Structure of a C Program

All C programs consist of one or more *functions*. Remember, a function is a group of one or more C statements designed to accomplish a specific task. Many functions that you will be using are already written and compiled for you in the function library supplied with your compiler. Instead of writing the individual instructions yourself, you just tell the compiler to use one of its standard function. Only when you want to perform a task that is not in the library do you need to write your own function.

We begin by considering the basic structure of a C program which may look like:

CamScanner

```
#include<...>
#include<...>
void main(void)
{
  statement 1;
  statement 2;
  .........
}
```

Header files
Function name
Start of function
Function body
End of function

The first task of anyone learning programming is to get computer to print on the screen. Our first example prints the text "This is my first C program" on the screen. The program 3.1 shows the structure of complete C program;

Program 3.1

```
#include<stdio.h>
void main(void)
{
  printf("This is my first C program");
}
```

Commentary on Program 3.1

1. `#include<stdio.h>`

   The lines that begin with a # are called *pre-processing directives*. This `#include` directive tells the compiler to use the information in the *header* file called `stdio.h`. The initial `stdio` stand for *standard input/output* and `h` stand for header file. The `stdio.h` file contains all the instructions the compiler needs to work with disk files and send information to the screen. This header file is provided by the Turbo C++ compiler. The angle brackets around `<stdio.h>` indicate that this file is to be found in the "usual place", which is system-dependent. Another example of header file is `<conio.h>` in which `conio` stand for *console input/output*.

2. `void main (void)`

   C programs are collections of *functions* designed to perform a specific task and return a computed result. At a minimum, all C programs must specify one function called `main`. The parentheses following `main` indicate to the compiler that it is a function.

   The `void` before `main` specifies that the function `main()` will not return a value. The second `void` in the parentheses, specifies that the function takes no arguments.

3. `{ }`

   The left brace begins the body of each function. A corresponding right brace must end the function.

4. `printf()`

   The C compiler contains a standard library of functions that can be used in programs. This is a function from the library that prints on the screen and is accessible through the standard input/output header file (`stdio.h`).

CamScanner

5. `"This is my first C program"`

A string constant in C is a series of characters marked by the quotation marks. This string is an *argument* to the function `printf()`, which controls what is to be printed. So in the instruction

```
printf("")
```

the string *This is my first C program* is an argument which will be printed.

6. `;`

Every statement must end with semicolon (also known as the *statement terminator*). You can put several statements on a line. For clarity's sake usually one statement is put on a line.

After you have entered your source code, pull down the *File* (Press Alt+F) menu and press S for *Save*. Type in the name of your program when the dialog box appears. The default *Turbo C++*'s "NONAME00.CPP" disappears as soon as you start typing. You don't have to type "CPP" extension. *Turbo C++* automatically assigns CPP extension.

After you enter the filename, say FIRST and press ↵, the filename, FIRST.CPP, appears in the caption bar instead of NONAME00.CPP.

Understanding Arguments

When you use a function in a statement, such as `printf()`, you are said to be *calling the function*. This means that you are called on C to perform the function.

The parentheses after a function name are designed to hold zero or more arguments. An argument is an item of information the function needs in order to complete its task (arguments are also referred to as *parameters*). For example, `printf()` is a function in the library. The function contains instructions that tell the computer to print a string of characters on the screen. But what string does it display? You have to tell it by placing what you want to display in the parentheses. We call this *passing an argument*. So in the instruction:

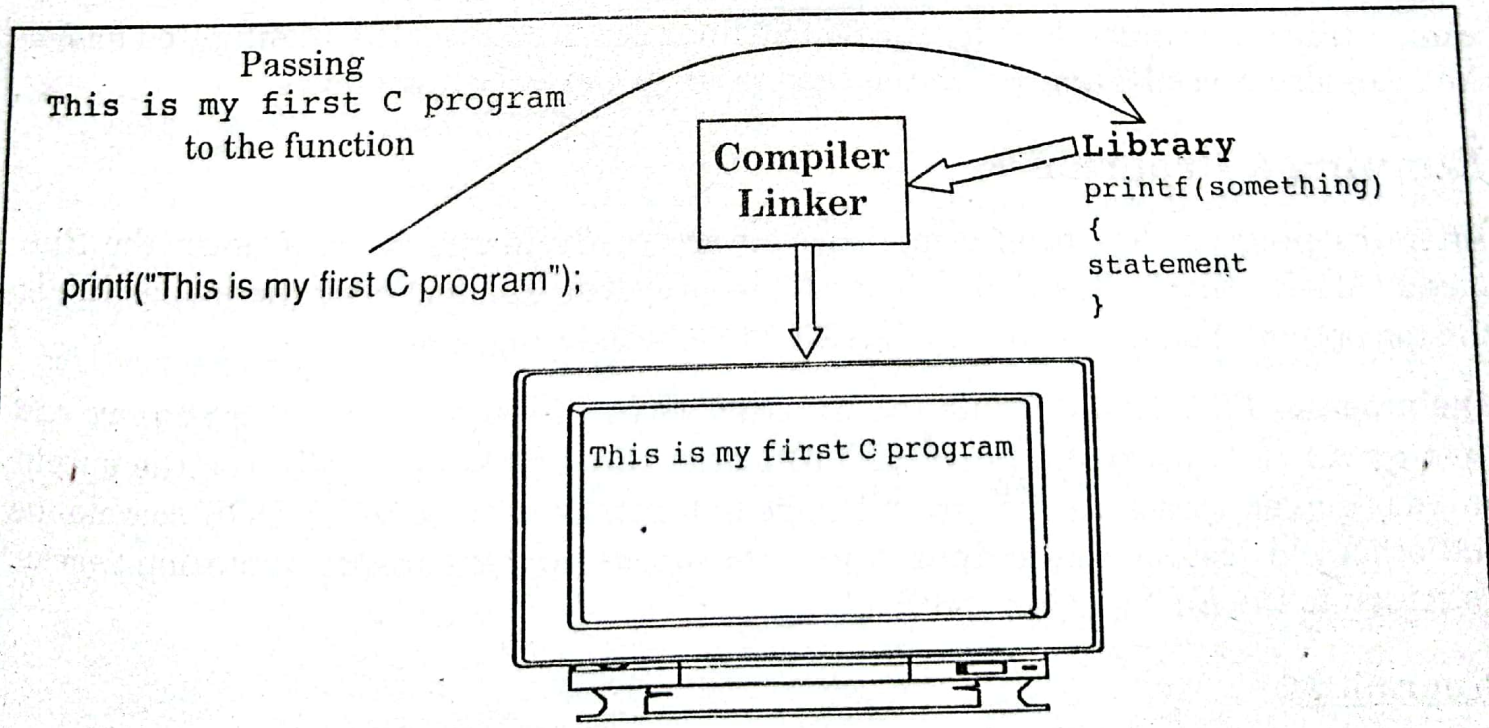


Fig. 3.3. Passing an argument to a library function.

```
printf("This is my first C program");
```

the string *This is my first C program* is the argument we are passing to the function. As illustrated in Fig. 3.3, we are telling the compiler to perform the `printf()` function and to use the string *This is my first C program* when it does.

Compiling a Program

After you have typed and saved your source code, the next step in program development is compilation. To compile a program, press Alt+F9. Alternatively, you can pull down the *Compile menu* (press Alt+C), then press C or ↵ for *Compile to OBJ*.

The compilation takes place instantaneously. If the code is correctly compiled, the dialog box will display, *"Success". Press any key.* While compilation goes on you can see the running line number and error total in the dialog box.

Any typing or spelling errors are reported by the *Turbo C++* compiler at the bottom of the screen in the *Message* window. The error messages are not very helpful, as the descriptions in the Reference Guide are necessarily quite vague. You can however, examine the program for common mistakes such as misspellings and missing parentheses, braces, slashes, backslashes, or semicolons. Be sure to save the corrected program with the *S (Save)* command from the *File* menu.

Turbo C++ provides several error messages. Each error may produce several of them. You may even find correct lines flagged as errors because of errors that precede them. If you get one or more errors, *Turbo C++* will display them all in the *Message* window. Press an arrow key to move to another message and the F6 key to move the cursor to the actual error in the *Edit* window. Press F6 again to return to the *Message* window.

Sometimes if more than one window is open, you may not be sure about the currently active window. You can ascertain which is the currently active window by first looking for the cursor or highlight, which is always in the active window. If you cannot trace the cursor, look for the double lines at the top and the highlighted name. You can also press F6 and watch the change when you switch windows.

Running a Program

After the program has been compiled, you are ready to run it. Pull down the *Run* menu (Alt+R), then press ↵ or *R* to start the program, ↵ alone does the job, as run is the top option. You can also press Ctrl+F9 to run the program.

The program FIRST.CPP prints the message `This is my first C program` and then quickly (almost instantaneously) return to the *Edit* window. To see the output on your screen, press Alt-F5. It will appear together with leftover DOS commands and other old results. The output is just the words you put inside quotation marks. To return to the *Edit* window, press ↵.

Program 3.2

Here is another simple C program.

```
#include<stdio.h>
void main(void)
{
  printf("This is");
  printf(" another");
  printf(" C program");
}
```

This program display *This is another C program* on the screen. The key point to this program is that statements are executed sequentially, beginning with the opening brace and ending with the closing brace.

Using Comments

A *comment* is a message to anyone reading the program. Adding comments makes your programs easier to understand. Comments can appear anywhere in a program. They are ignored by the compiler and linker. In fact, they are never added to the object code or to the executable program.

You begin a comment with the characters /* and end it with */, like this:

```
/*This program displays a word on the monitor*/
#include<stdio.h>
void main(void)
{
  printf("OK");
}
```

The /* indicates the start of the comment, and */ indicates the end of the comment. *Turbo C++* will ignore everything between the two symbols. Programmers usually place a comment at the start of each program to explain its overall purpose. Comments within a program explain a particular instruction or point of logic, and you can even add a comment after the semicolon of an instruction:

```
/*This program displays a word on the monitor*/
#include<stdio.h>
void main(void)    /*Start of a function*/
{
  printf("OK");    /*This instruction prints a word*/
}
```

When you include an instruction and comment on the same line, it is conventional to leave some blank space between the two. This simply makes the instruction and comment easier to read.

If you have lengthy comments, type as many lines as you want between the codes:

```
/*This program displays a word OK on the monitor and includes a
* void main (void) function to make it compatible with Turbo C++
* compiler that require it.
*/
```

The additional asterisks are only for appearance. They, like everything else between /* and */, are ignored.

CamScanner

A common mistake programmers make is to leave out the closing */ resulting in compiler error messages. *Turbo C++* makes using comment a little easier by allowing another comment marker, the symbols //. This type of comment stops at the carriage return ↵ that ends the line, so you do not need to type an end-of-comment code:

```
//This program displays a word on the monitor
#include<stdio.h>
void main(void)    //Start of a function
{
  printf("OK");    //This instruction prints a word
}
```

If your comment runs over one line, however, each line must start with //:

```
// This program displays a word OK on the monitor and includes a
// void main (void) function to make it compatible with Turbo C++
// compiler that require it.
```

Escape Sequences

You can control the way the cursor moves on your monitor, and some other computer functions, by using special codes, called escape sequences. An escape sequence consists of the escape character (the backslash '\') and a second character. The backslash causes an escape from the normal interpretation of a string, so that the next character is recognised as one having a special meaning.

Though there are two characters in an escape sequence, they are considered as a single character. Some of them are shown in the following table.

Table 3.1. Escape sequence Characters

Escape Sequence	Meaning	Result at Execution Time
\n	New line	Move the cursor at the beginning of the next line
\t	Tab	Move the cursor to the next tab stop
\a	Alert	Produces a beep
\r	Carriage return	Moves the active position to the initial position of the next paragraph
\b	Backspace	Moves the active position to previous position on the current line.
\f	Form feed	Moves the active position to the initial position of the next logical page
\0	End of string	Null
\\	Backslash	Presents with a backslash \
\'	Single quote	Presents with a single quote
\"	Double quote	Presents with a double quote
\?	Question mark	Presents with a question mark
\xhh	Hexadecimal escape sequence	ASCII code in hexadecimal notation (each h represents a digit)
\ddd	Octal escape sequence	ASCII code in octal notation (each d represents a digit)

CamScanner

The escape sequence \n performs a new line command and is most commonly used at the end of an instruction. It can appear anywhere inside the double quotes and moves the cursor to the start of the next line. These commands

```
printf("This is\n");
printf("my first\n");
printf("C program");
```

or this command

```
printf("This is\nmy first\nC program");
```

displays three lines of text:

```
This is
my first
C program
```

The \t tab command moves the cursor to the next preset tab stop. To see how the tab code works, use this command

```
printf("This is\n\tmy first\n\t\tC program");
```

The command displays the output

```
This is
        my first
                C program
```

The \r code performs a carriage return, moving the cursor to the start of the same line, without moving down to the next line.

Fig. 3.4 shows the difference between a carriage return and a new line command.

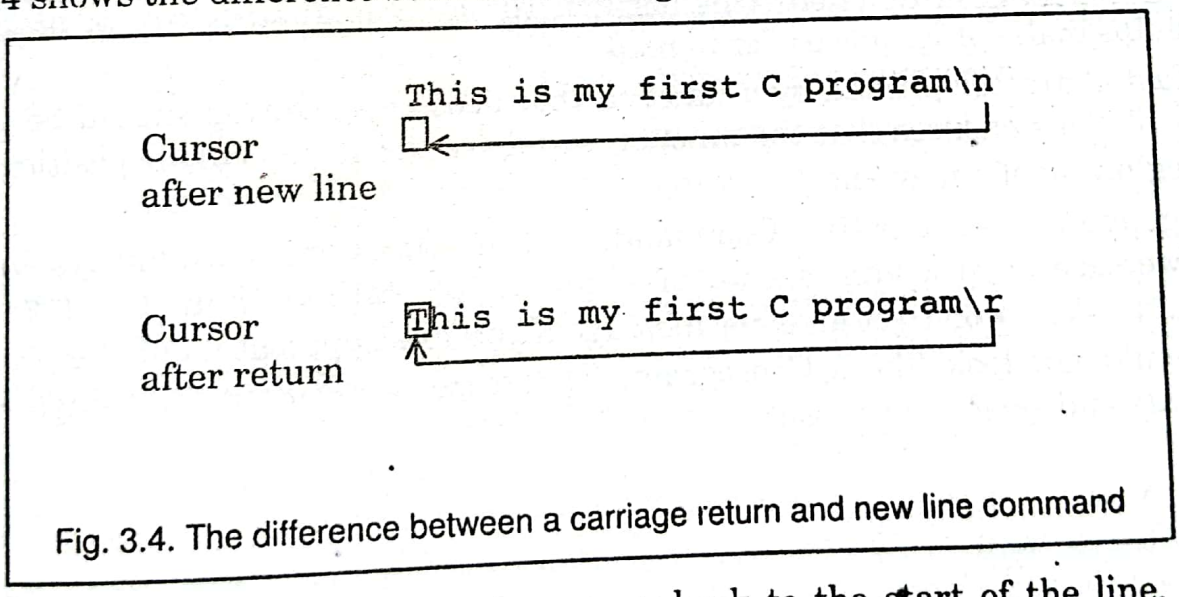


Fig. 3.4. The difference between a carriage return and new line command

If your display something, move the cursor back to the start of the line, and then display something else, the first text will be overtyped. The effect of using the \r code can be seen in this instruction:

```
printf("Left\rRight");
```

The screen displays only the word Right. Here is why: After displaying the *Left*, the \r code moves the cursor back to the start of the line. The word *Right* is displayed, replacing the characters of the word *Left*.

Unlike the \r code, which backs the cursor to the start of the line, the \b code moves

the cursor just one position to the left. It, too, is non-destructive in that it does not erase a character it encounters.

The `\0` code represents the *null* character, is used to indicate the end of a string. The compiler automatically places a null character at the end of every string, as the last character within the string. This character is not visible when the string is displayed.

Example 3.3

Name the escape sequences used in the following string constant.

```
"\tTo continue, press the \"ENTER\" key\n"
```

Solution

The escape sequence `\t` for horizontal tab
`\"` for double quotation marks, which appears twice
and
`\n` for new line.

Good Programming Practice

1. The compiler will be able to perform the program either way. But to make a program more readable, put `void main (void)` function on a line by itself and opening and closing braces on their own lines.
2. Every program or function should be preceded by a comment describing the purpose of the program.
3. Indent the entire body of function one level of indentation within the braces that defines the body of function. This displays the functional structure of programs and helps make programs easier to read.
4. The last character printed by a function that does any printing should be a new line (`\n`). This ensures that the function will leave the screen cursor positioned at the beginning of a new line.
5. C language is case sensitive. Commands and function names are always written in lowercase characters, so we use `printf()` rather than `PRINTF()` or `Printf()`. You won't get an error message if you use uppercase, but the program just would not look like a C program. Uppercase is reserved for things called *constants* and *symbolic names*.

CamScanner

Exercise

3.1. Briefly explain the history of C language.

3.2. Where was C originally developed and by whom? What has been done to standardise the language?

3.3. What is C++? What is the relationship between C and C++?

3.4. What are the general characteristics of C?

3.5. What is the purpose of the statement `#include<stdio.h>` in a C program?

3.6. What is a statement terminator?

3.7. What is the purpose of comments in a C program?

3.8. Describe the composition of a function in C. Explain the `main()` function.

3.9. Why do you use a parameter when calling a function?

3.10. What is the general structure of a C program?

3.11. Why are some of the statements within a C program indented?

3.12. Write a program that displays the following on the screen:

(a) Welcome to Computer Lab.

(b) Board of Intermediate Education, Karachi.

3.13. Write the `printf()` instructions for printing your name, address, and telephone number, centred on the screen.

3.14. Explain what is wrong with the following program:

```
#include<stdio.h>
void main(void)
(
  printf("I am a College Student");
}
```

3.15. ANSI C does not provide for the nesting of comments, although many compilers provide an option for this. Try the following line on your compiler and see what happens.

```
/*This is an attempt/*to nest*/a comment.*/
```

3.16. What are escape sequences?

3.17. What is the difference between `\n` and `\r` escape sequence?

3.18. Are C programs required to be typed in lowercase? Is uppercase ever used in C program? Explain.

3.19. State some rules of a good programming practice.

CamScanner

Objectives and MCQs

Choose the answer that best completes the statement.

3.1. C is traditionally called the development language of the
- (a) Windows 98 operating system
- (b) Windows NT operating system
- (c) UNIX operating system
- (d) MS Word

3.2. In 1983, C++ was developed by Bjarne Stroustrup at the
- (a) Bell Laboratories.
- (b) Microsoft
- (c) IBM
- (d) U.S. Defence Department

3.3. Borland International Inc., USA released its own version of C++ in 1992 and called its complier as
- (a) Turbo C++
- (b) R&K C++
- (c) SuperSoft C++
- (d) DBS C++

3.4. All C programs consist of one or more
- (a) functions
- (b) `scanf()` statements
- (c) `printf()` statements
- (d) pointers

3.5. At a minimum, all C programs must specify one function called
- (a) basic function
- (b) main function
- (c) primary function
- (d) elementary function

3.6. You begin a comment with the characters
- (a) */
- (b) /*
- (c) *
- (d) *\

CamScanner

Objectives

3.1. C language has been developed by Dennis Ritchie.
3.2. C language has been developed at AT & T's Bell Laboratories of USA in 1972.
3.3. C++ is basically a superset of C.
3.4. C is known as the development language of the UNIX operating system.
3.5. The ANSI C standard was approved in 1989.
3.6. All C programs consist of one or more functions.
3.7. All C statements must end with a semicolon.
3.8. The statements within a C function must always be enclosed within a pair of {}.
3.9. Every C program must contain a `main()` function.
3.10. It is possible to write programs in C that are portable to most computers.
3.11. A compiler translates a C program into machine language code (or object code).
3.12. The C pre-processor obeys pre-processor directives which typically indicate that other files are to be included in the file to be compiled and special symbols are to be replaced with program text.
3.13. A linker links the object code with the code for missing function to produce an executable image.
3.14. Comments can appear anywhere in a program.
3.15. Comments are ignored by the compiler and linker.
3.16. The `void` before `main` specifies that the function `main()` will not return a value.
3.17. The `void` in the parentheses after `main`, specifies that the function takes no arguments.
3.18. C++ provides capabilities to do object-oriented programming.

CamScanner

CHAPTER 4

C FUNDAMENTALS

By: Sir Ahad

We refer to input given to the computer as *data*. You feed data into the computer; the computer processes it following your instructions and gives you information as output. But before you input data into your program, you have to tell it what type of data it is dealing with.

In this chapter we will discuss the basic elements used to construct simple C programs. These basic elements include the C character set, identifier and keywords, data types, constants or literals, variables and arrays and their declarations.

The C Character Set

A character denotes any alphabet, digit or special symbol used to represent information. The valid alphabets, numbers and special characters allowed in C are listed below:

Alphabets	A, B, C, ..., Z
	a, b, c, ..., z
Digits	0, 1, 2, ..., 9
Special characters	+ − * / = % & # @ $! ? ^ " ' ~ \ ¦
	< > ( ) [ ] { } : ; . , _ blank space

Identifiers and Keywords

Identifiers are names given to various program elements, such as variables, functions and arrays. Following rules must be observed while constructing identifiers:

1. An identifier must start with a letter or (rarely) an underscore (_).
2. An identifier can contain letters, digits, or underscores (_).
3. An identifier cannot contain blanks, commas, or ay other special characters.
4. An identifier cannot consist of more than 31 characters.
5. An identifier must conform to case sensitivity (C differentiates between uppercase and lowercase alphabetical characters). For example, the identifier `MAX` is not the same as the identifier `max`.

CamScanner

There are several conventions for selecting variable names. Variable names are usually written in lowercase letters, with uppercase names reserved for symbolic constants. The underscore may be used to improve the readability of a variable name. The following names are valid identifiers.

```
x          y1         y2          sum_1       sum_2
TABLE      table      tax_rate    _auto       _temperature
```

The following names are *not* valid identifiers for the reasons stated.

`triangle area`	illegal character (blank space).
`tax-rate`	illegal character (-).
`"x"`	illegal character (").
est#west	special character # not allowed.
5^{th}	the first character must be a letter.
−temperature	do not mistake − for _.

Convention suggests using the underscore as the first letter for variable and function names in the standard library (*e.g.* `_auto`). The underscore minimizes the chances of a "collision" between function and variable names in your library routines and those in a program.

Good programming style requires the programmer to choose names that are meaningful. If you were to write a program to figure out various taxes, you might have identifiers such as tax_rate, price, and tax, so that the statement

```
tax = price * tax_rate
```

would have an obvious meaning. The underscore is used to create a single identifier from what would normally be a string of words separated by spaces. Meaningfulness, avoiding confusion and readability constitute the main guidelines for a good programming style.

There are certain reserved words, called *keywords*, that have standard, predefined meanings in C. These keywords can be used only for their intended purpose; they cannot be used as variable names. The standard keyword for *Turbo C++* are:

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

Compared to other major languages, C has only a small number of keyword. It is a characteristic of C that it does a lot with relatively few special symbols and keywords. Although C does not permit the use of keywords as variable names, a variable named `auto_type`, for example, is permissible because C treats it as one word, and therefore, does not consider it to be a keyword.

Data Types

To prevent data values from being manipulated with the wrong operations and to

prevent their representations from being misinterpreted, the data items are classified into *data type*. Values belonging to the same type are subjected to the same operations and are represented in the same way.

C supports several different types of data, each of which may be represented differently within the computer's memory. The basic data types, their description and the memory requirements for each data type is listed in Table 4.1.

Table 4.1. The Basic Data Types

Data Type	Description	Typical Memory Requirements
`int`	integer quantity	2 bytes
`char`	single character	1 bytes
`float`	floating-point number (*i.e.*, a number containing a decimal point and/or an exponent)	4 bytes
`double`	double-precision floating-point number (*i.e.*, more significant figures, and an exponent which may be larger in magnitude)	8 bytes

Integer and floating-point data types represent numbers. They are often referred to collectively as *numeric data type*.

These four basic types can be modified using C's *type modifiers* to more precisely fit your specific need. The type modifiers are

1. short
2. long
3. singed
4. unsigned

These type modifier precedes the type name. For example, this declares a `long` integer.

```
long int i;
```

There is also a data type called 'void', which is used to specify an empty set of values. The use of such a data type is important when we study *functions* and *pointers*, which is out of the scope of this book.

Now we will discuss each of these four data types in detail in the next subsection.

`int` Data Type

The `int` data type represents *integer* values, meaning any positive or negative number, or zero, without a decimal point. When such values appear in a program they are called integer. Example of valid integers are

4 −87 +9345 −18239

An integer consists of an optional plus or minus sign followed by a series of digits and cannot contain any other character. Commas, decimal points, or special symbols such as $ or e for exponential notation are not allowed.

Declaration of an integer can be preceded by type modifiers short, long, signed or unsigned. So examples are:

CamScanner

```
short int num_1, num_2;        define short integer variables
long int avg;                  define long integer variable
unsigned int total;            define unsigned integer variable
long unsigned int c;           define unsigned long integer variable
int a, b;                      define integer variables
int a=5;                       initialise integer variable
```

In fact, a `short int` is nothing but ordinary `int`. C allows the abbreviation of `short int` to `int` and of `long int` to `long`. So the first two declarations made above can be written as,

```
int num_1, num_2;      define short integer variables
long avg;              define long integer variable
```

Also the `unsigned int` is nothing but a `short unsigned int`. Thus, all the following declarations are same:

```
short unsigned int total;
unsigned int total;
unsigned total;
```

The way there exists a `short unsigned int`, there also exists a `long unsigned int`. Naturally, most C programmers prefer this short-cut.

Usually the number of bytes reserved by the compiler for the above integer type modifiers are as given below.

Data Type	Range	Bytes width	Format
short singed int	−32768 to +32767	2	%d
short unsigned int	0 to 65535	2	%u
long singed int	−2147483648 to +2147483647	4	%ld
long unsigned int	0 to 4294967295	4	%lu

Example 4.1

Which of the following are not valid integers?

15 +983.8709 $13,789 78,975 875

Solution

+983.8709	decimal point
$13,789	$ and comma
78,975	comma

`float` Data Type

The `float` data type represents *floating-point* values, meaning any singed or unsigned number having a decimal point. Examples of valid `float` variables are

2. −78.294 395.20587 +95.0 −439.

A `float` consists of an optional plus or minus sign followed by a series of digits and must include a decimal point. It cannot contain commas or special characters such as the dollar sign but can include the letter `e`. `float` can be single- or double-precision.

CamScanner

The *default*—the assigned data type when one is not specified—is `double` (double-precision), which occupy twice the number of bytes as `float`.

`float` and `double` can be written in exponential or scientific notation if needed. The number is written as a value (the mantissa), which is multiplied by 10 (represented by letter `e` (or `E`) which stands for 'exponent') to the stated power (the characteristic). The exponent specifies the number of places the decimal point is to be moved to the left or right. If the exponent is positive, the decimal point is moved to the right; if the exponent is negative, the decimal point is moved to the left. The

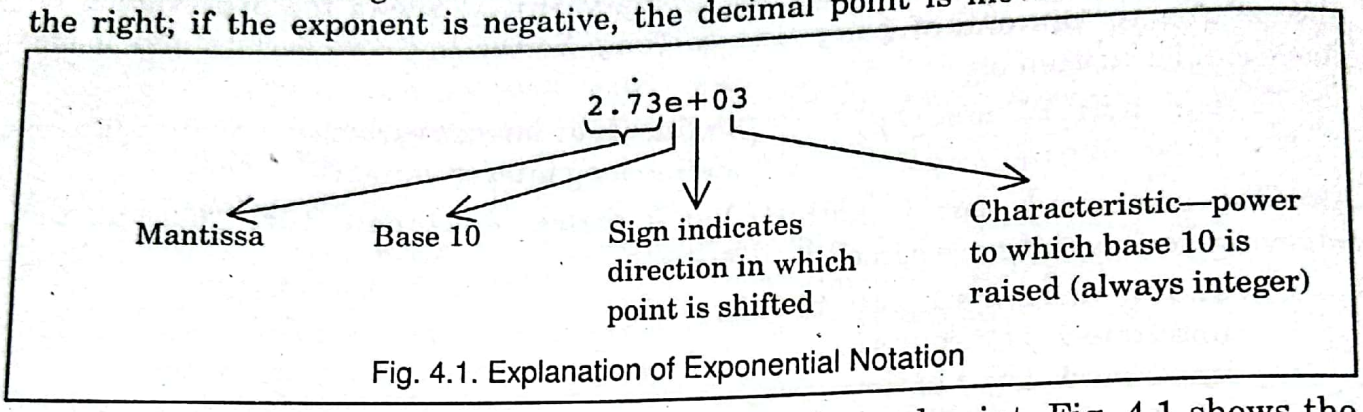


Fig. 4.1. Explanation of Exponential Notation

mantissa always has one non-zero digit before the decimal point. Fig. 4.1 shows the parts of a number represented in exponential notation. The number in Fig. 4.1 would be 2730.0 in decimal notation. Below are some numbers in exponential notation along with their equivalents in floating-point notation:

Exponential Notation	*Float Notation*
7.262e+4	72620.0
−2.1613e+2	−216.13
3.14E−3	0.00314
−729.0e−2	−7.29

The `long double` may refer to a separate "extra-large" double-precision data type. It stores an 80-bit floating point number.

Some examples of the declaration of a float are shown below:

```
float flnum;          define floating point number
float fna=93.9;       initialising floating point number
double verybig;       define double precision floating point number
double b=98.74563;    initialising double precision number
long double verybig;  define long double
long double b=9.74563; initialising long double
```

Usually the number of bytes reserved by the compiler for the above float type variables are as given below.

Data Type	Range	Bytes width	Format
float	3.4e−38 to 3.4e+38	4	%f
double	1.7e−308 to 1.7e+308	8	%lf
long double	3.4e−4932 to 1.1e+4932	10	%Lf

CamScanner

Example 4.2

Which of the following are not valid doubles?

15 983.8709 $13,789 −7897I +4.1 487.OO 673.0e03 8475e−3

Solution

15	no decimal point
$13,789	$ and comma and no decimal point
−7897I	no decimal point, letter "I" instead of number "1"
487.OO	"OO" are two characters not the two number "00"
8475e−3	no decimal point

char Data Type

C also recognises nonnumeric character data. A `char` represent individual characters. Hence, the `char` type will generally require only one byte of memory. A `char` is any valid ASCII character enclosed in single quotes. Examples of `char` are

```
'm'   '>'   'H'   '2'   '['   '!'   'F'
```

Note that a `char` data item can also be a single numeric digit, such as 1, 2, or 3. But C makes a distinction between the character '1' and the number 1. As a *character* '1' cannot be used in a mathematical operation because it does not have a numeric value. As a *number*, 1 can be used in math.

Declaration of a character can be preceded by type modifiers *signed* or *unsigned*. The `singed char` is same as ordinary `char`. Some examples of the declaration of a character are shown below:

```
char ch;                define character variable
unsigned char p;        define unsigned character variable
char c1='a';            initialise character variable
char c2=432;            initialise using decimal value
char c3=0x61;           initialise using hexadecimal value
```

Usually the number of bytes reserved by the compiler for the above character type variables are as given below.

Data Type	Range	Bytes width	Format
signed char	−128 to 127	1	%c
unsigned char	0 to 255	1	%c

Example 4.3

Which of the following are not valid `char`?

'z' 'zz' "z" '0' z ' '

Solution

'zz'	2 characters
"z"	double quotation marks
z	no quotation marks

CamScanner

Choosing the Data Type

When choosing the data type to represent something in a program, think about the likely data range of the variable. For example, if a variable is to hold student examination marks out of 100, then an `int` type will be more than sufficient. However if the variable is to hold, for example, the population of Pakistan, then the integer type will be too small, and a `long unsigned int` type will be required.

Unsigned variables are useful if we know in advance that there will never be a negative value for that variable. For example, if we are considering number of students in a class, then the value can never be less than zero, so an unsigned variable may be used.

Any variable which is likely to contain decimal fractions will have to be at least a `float`. If we are considering a variable which is required to handle very large numbers, *i.e.*, for programs dealing with astronomy for example, the `long` data type would be necessary to meet the accuracy requirements.

The `char` data type can hold numbers, but is used to represent characters. A `char` can be assigned either a numeric value or the character itself, surrounded by apostrophes. For example, using ASCII code, the number 65 and the character 'A' are equivalent values for a `char` data type, since 65 is the ASCII code for a capital 'A'.

Example 4.4

What would be the appropriate data types to store the following:

1. Some one's height in metres
2. An exclamation mark (!)
3. The number of students in a university

Solution

The selected data types should be as follows:

1. A *float,* because the height may contain a decimal fraction.
2. A *char,* because it is an ASCII character.
3. An *int,* or more appropriately, an *unsigned int,* because there will never be a negative number of students.

Constants

Constants are quantities whose values do not change during program execution. In C, however, some authors refer them as *literals* or *literal constants*. In any case, there are four basic types of constants in C. They are *integer constants, floating-point constants, character constants* and *string constants*. Integer and floating-point constants represent numbers. They are often referred to collectively as *numeric-type constants*. Let us see the details of each of these constants.

Integer Constants

An *integer constant* is an integer-valued number. Thus it consists of a sequence of digits.

CamScanner

Following rules must be observed while constructing integer constants:

1. An integer constant must have at least one digit.
2. It must not have a decimal point.
3. It could be either positive or negative.
4. If no sign precedes an integer constant it is assumed to be positive.
5. No commas or blanks are allowed within an integer constant.
6. The value of a constant cannot exceed specified minimum and maximum bonds. The allowable range for integer constants is −32768 to +32767.

Integer constants include decimal, octal and hexadecimal constants. A *decimal* integer constant can consist of any combination of digits from 0 to 9. If the decimal integer constant contains two or more digits, the first digit must not be a 0.

Some examples of valid decimal integer constants are:

```
273      −597      0      3      666
```

On the other hand, the following decimal integer constants are written incorrectly for the reason stated.

`08`	the first digit cannot be a zero
`574-26-28`	illegal character (-)
`−36.2`	contain fractional point
`67 29`	blank space is not allowed
`17,269`	illegal character (,)

An *octal* integer constant can consist of any combination of digits from 0 to 7. However the first digit must be 0, in order to identify the constants as an octal number.

Some examples of valid octal integer constants are:

```
0      01      −0573      0666
```

On the other hand, the following are not valid octal integer constants.

`05927`	illegal digit (9)
`056.23`	contain fractional point
`264`	does not begin with 0

A *hexadecimal* integer constant can consist of any combination of digits from 0 to 9 and letters `a` to `f` (either upper- or lowercase). Hexadecimal integer constants must begin with `0x` or `0X` in order to identify it.

Some examples of valid hexadecimal integer constants are:

```
0x      0x2      −0X27FA      0xabcd
```

On the other hand, the following are not valid hexadecimal integer constants

`0XDFG`	illegal character (G)
`0x14.9A`	contain fractional point
`9AC5`	does not begin with `0x` or `0X`

CamScanner

Floating-Point Constants

Integers are inadequate for representing very large or very small numbers and fractions. For this you need floating-point types.

Floating-point constants are also called *real constants*. These are base-10 numbers and could be written in two forms, factional form or exponential form (or both).

Some examples of valid floating-point constants are:

```
0.      1.      -0.5      6070      0.0001      3E-7      3e-7
0.002e+3      .2191129e19
```

On the other hand, the following floating-point constants are written incorrectly for the reason stated.

`3`	no decimal point or exponent
`7,00.0`	illegal character (,)
`2e+5.6`	the exponent value must be an integer
`5e 2`	illegal character (blank space in the exponent)
`2.7e`	the exponent sign must be followed by a number

Character Constants

A *character constant* is a single character, enclosed in apostrophes (*i.e.*, single quotation marks). They include the letters of alphabet (lowercase and uppercase), the numerals 0 to 9, punctuation marks such as :, ; and ?, and special characters such as &, +, −, # and $.

Some examples of valid character constants are:

```
'B'      'm'      '5'      '*'      '?'      ' '
```

Notice that the last constant consists of a blank space, enclosed in apostrophises.

On the other hand, the following character constants are written incorrectly for the reason stated.

`'3`	missing right quotation mark
`'73'`	more than one character inside quotation marks
`#`	quotation marks missing

Notice that it is possible for the computer to store a number as a character value. However, because the computer stores character values differently than numeric values, the computer can no longer perform arithmetic operations on these numbers.

Virtually all the personal computers make use of the ASCII character set to encode a character. The standard ASCII code uses 7-digit binary numbers; *i.e.*, number consisting of various sequences of 0s and 1s. The code can represent 128 (*i.e.* $2^7 = 128$) different characters, since there are 128 different possible combinations of seven 0s and 1s. Appendix A contains the ASCII character set, showing the decimal equivalent of the 7 bits that represent each character. For example the ASCII code for 'A' is 65. Therefore, after making declaration

```
char ch;
```

you can make either of the flowing assignment statements;

CamScanner

```
ch = 'A';
ch = 65;
```

Note that character constants are enclosed in single quotes. The quotes tell the compiler to get the numeric code (ASCII code value) of the character.

Program 4.1

This Program prints ASCII code value of the inputted character.

```
/* program 4-1.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  char key;
  clrscr();
  printf("\nPress any Key: ");
  scanf("%c", &key);
  printf("\nASCII code of %c = %d",key,key);
}
```

String Constants

A sequence of zero or more characters enclosed by double quotes is called a *string constant*. This can also be defined as an array of character constant. Some examples of valid character constants are:

"The quick brown fox jumps over the lazy dog."
"514-86-112109"
"12-Qasim Rd."

Below are two incorrect examples:

"The city	(missing right quotation mark)
'A brown cat.'	(must use double quotation marks)

A blank string constant can be represented by " ". In this case, a blank space is placed between two quotes. Also a character constant (*e.g.*, 'A') and the corresponding single-character string constant ("A") are not equivalent. Also remember that a character constant has an equivalent integer value, whereas a single-character string constant does not have an equivalent integer value and, in fact, consists of two characters—the specified character followed by the escape sequence (\0).

Symbolic Constants

A *symbolic constant* is a programmer-defined identifier that is replaced with a sequence of characters called *text*. The replacement *text* may represent a numeric constant, a character constant or a string constant. Thus, a symbolic constant allows an identifier to appear in place of a numeric constant, a character constant or a string constant. When a program is compiled, each occurrence of a symbolic constant is replaced by its corresponding replacement text.

Symbolic constants are usually defined at the beginning of a program, often before [illegible] also allowed to appear after the opening brace { and main.

CamScanner

The symbolic constants may then appear later in the program in place of the numeric constants, character constants, etc. that the symbolic constant represent.

Consider a program that makes extensive use of π. We could use the constant 3.14159 throughout, but there are several compelling reasons not to. First, there is a possibility of a typographical error on one of the occurrences of the number. Second, if it later becomes important to change the degree of precision, we have to find all occurrence of 3.14159 and replace them with, say, 3.1415926535897 93. Finally, it is usually more informative to use a name rather than a "mystery" number whose value may have no meaning to the reader and may get confused with other unrelated numbers.

C provides two techniques for creating symbolic constants. The first uses `const`. We specify the keyword const, then the data type of the constant, then its symbolic name, and finally an assignment of its given value. Examples using `const` are as follows:

```
const float PI = 3.14159;
const int YEAR_LENGTH = 12;
const char ANSWER = 'Y';
```

For stylistic reasons, we use the convention of writing constants in uppercase. This distinguishes them from variables, which are usually written in lowercase.

The second technique for creating constants, the only method that was available before ANSI C, is through the `#define` pre-processor directive. The `#define` directive has an effect similar to `const`, although the syntax is slightly different. The general syntax for defining symbolic constant is:

```
# define NAME TEXT
```

where `NAME` represents a symbolic name, typically written in uppercase letters, and `TEXT` represents the sequence of characters that is associated with symbolic name. Note that the directive does not end with a semicolon or contain = sign. The exact syntax is based on the type of data you are defining—numeric, character, or string (see Fig. 4.2).

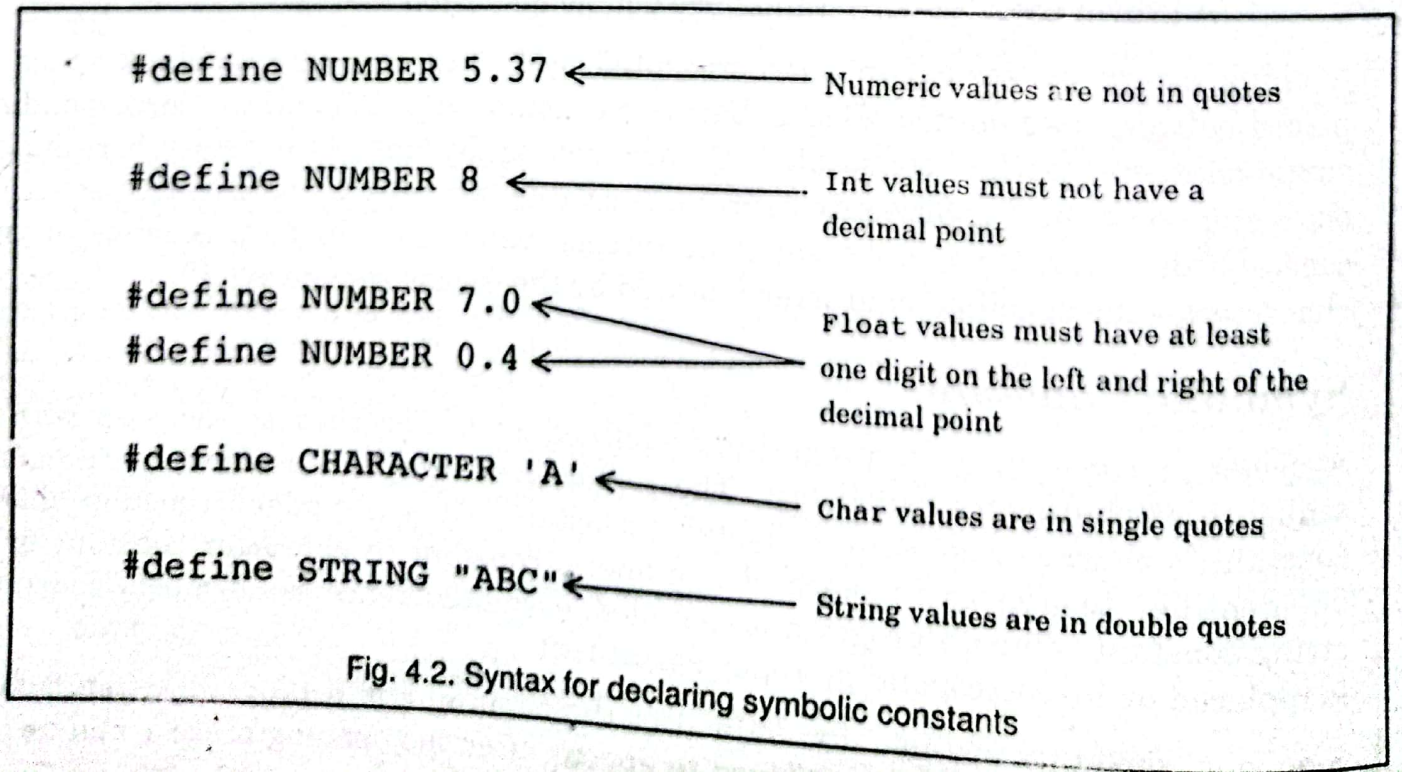


Fig. 4.2. Syntax for declaring symbolic constants

CamScanner

You declare a numeric constant by specifying the constant's name and value. For example, this directive:

```
# define PI 3.14
```

creates a constant called PI and gives it the value 3.14 (see Fig. 4.3).

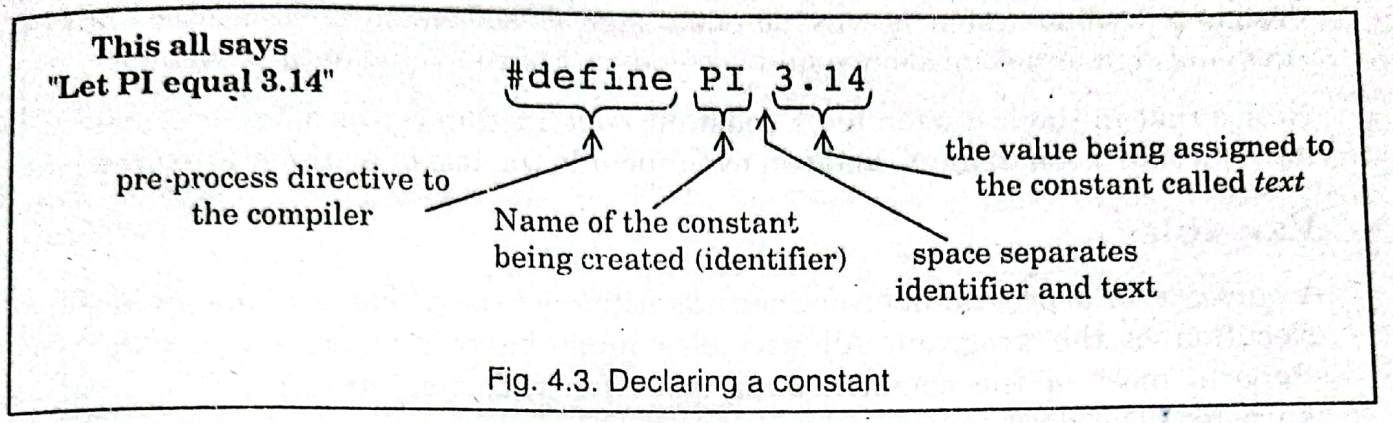


Fig. 4.3. Declaring a constant

Whenever the compiler sees PI in the source code (for example, in a formula), it substitutes the value 3.14 (see Fig. 4.4).

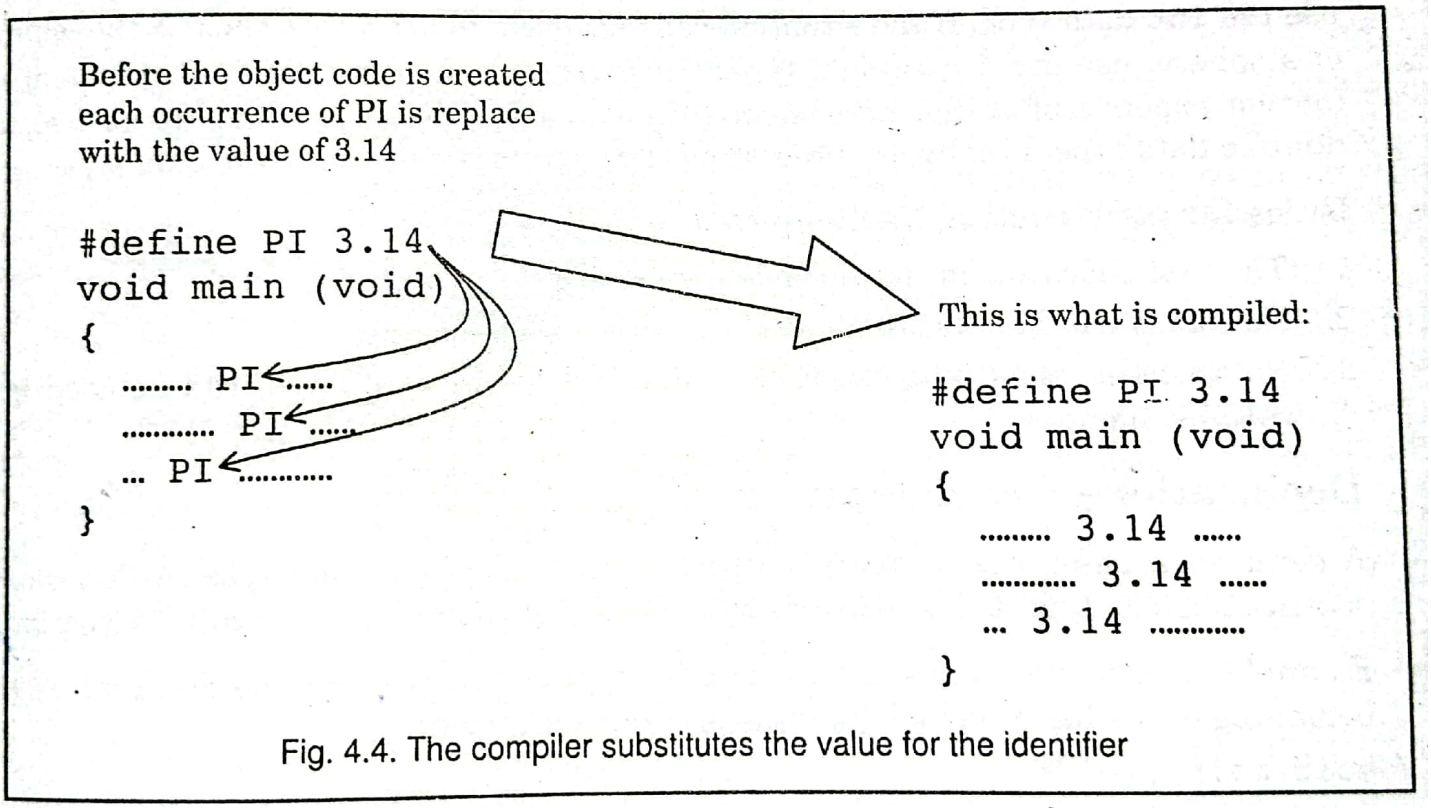


Fig. 4.4. The compiler substitutes the value for the identifier

Some other examples of pre-processed defined constants are as follows:

```
#define ANSWER              'Y'
#define PI                  3.14159
#define RADIUS              27
#define DAYS_OF_WEEK        "Mo Tu We Th Fr Sa Su"
#define FIRST_PROGRAMMER    "Ada Lovelace"
#define TRUE                1
#define FALSE               0
#define POLLUTION           FALSE
#define CIRCUMFERENCE       2*PI*RADIUS
```

CamScanner

Unlike `const`, which is a true C-language element, `#define` is a pre-processor directive and is handled differently. Before compilation, the pre-processor removes all references to the symbols created by `#define` and replaces them with their designated values. That is why no equal sign or semicolon is associated with them: An equal sign or a semicolon would become part of the value of the constant!

Notice that in the last example a constant is defined in terms of other constants that have already been defined, and can even include the use of arithmetic expressions.

Variables

A *variable* is a programmer-defined identifier whose value can change during the execution of the program. All variables must have a name and a data type. C supports most of the common data types for variables—integer, float (real), and character. The data type should be chosen carefully because it determines how the computer internally represents the associated data. If the situation calls for integer data, such as the number of pages in a book or the number of characters in a word, use the `int` data type. If the situation calls for decimal precision, such as the volume of a sphere, use the `float` data type. For extra precision in floating-point numbers (at the expense of using more memory and increased computation time), use the `double` data type. Finally, for processing character data, use the *char* data type.

Rules for constructing Variable Names

1. The first character in the variable name must be an alphabet.
2. No commas or blanks are allowed within a variable name.
3. No special symbol other then an underscore (as in `tax_rate`) can be used in a variable name.

Declaration of Variables

A *declaration* associates a group of variables with a specific data type. A declaration consists of a data type, followed by one or more variable names, ending with a semicolon.

Example of variable names and their data types as they might be declared at the beginning of the function look like this program statement:

```
void main (void)
{
  int num;
  float tax_rate;
```

This creates an `int` variables named num, and a `float` variable called `tax_rate` (see Fig. 4.5).

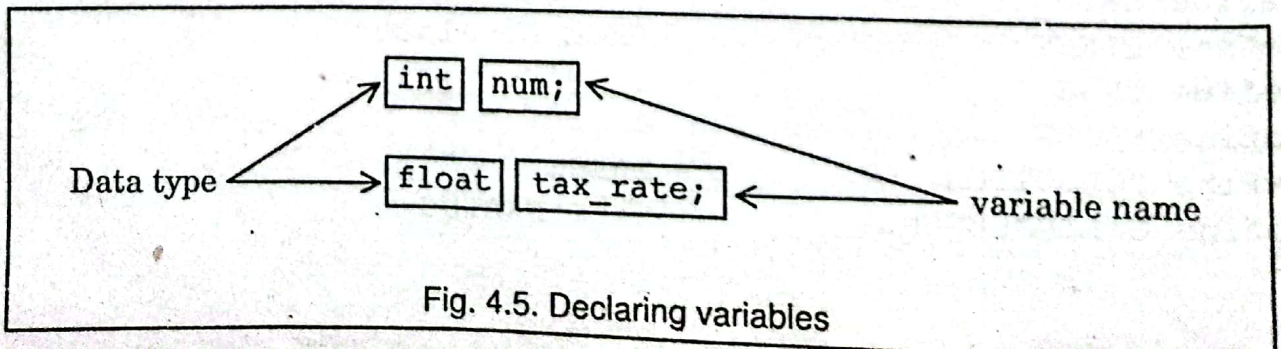


Fig. 4.5. Declaring variables

CamScanner

If you have several variables of the same type, you can declare them in one instruction. Separate each variable name with a comma and end the declaration with a semicolon, as in

```
void main (void)
{
  int count, students, year;
  float tax_rate, discount;
}
```

This creates five variables—three `int` values and two `float` values. Declare variables in a group within the `main( )` function, after its opening brace and before any other instructions. These declarations could also have been written as follows.

```
void main (void)
{
  int count;
  int students;
  int year;
  float tax_rate;
  float discount;
}
```

This form may be useful if each variable is to be accompanied by a comment explaining its purpose. In small programs, however, items of the same type are usually combined in a single declaration.

Some variables have an *initial* value—the value you want the variable to have when you start the program. To do so, the declaration must consist of a data type, followed by a variable name, an equal sign (=) and a constant of the appropriate type. A semicolon must appear at the end, as usual. Unlike a constant, this value can change as the program runs. You assign an initial value either in the declaration or as a separate instruction.

With numeric and `char` data, you can assign the value as part of the declaration:

```
void main (void)
{
  int count = 5;
  char initial = 'A';
  float rate = 0.55;
```

This instruction declares a variable named *count* and assigns it the initial value of 5. It also declares a `char` variable named *initial* and assigns it the letter A, and a `float` variable named *rate* with the value of 0.55. The value of the `char` variable must be in single quotes, and `float` values must have at least one digit to the left and to the right of the decimal point.

Once you declare a variable, you can also assign it a value in a separate instruction using the = symbol, like this:

```
count = 5;
initial = 'A';
rate = o.55;
```

CamScanner

The assigned value is only the initial value. As variables, the initial values can be changed as the program runs.

Arrays

An *array* is an identifier that refers to a collection of data items that all have the same name. The data items must all be of the same type (*e.g.*, all integers, all characters, etc.). C does not contain a string type and it does not have any built-in string functions. Fortunately, C does allow us to work with string-type data through the use of arrays. The characters of the string are stored together, in consecutive memory storage locations.

To declare a string variable, you use the `char` variable type and specify the maximum number of characters that the variable can hold. The syntax is this:

```
char VARNAME[N];
```

VARNAME is the name of the variable, and `N` is the maximum number of characters. The number is always in square brackets, [and].

The number in the brackets should actually be one more than the maximum number of characters you want to use. C needs the extra character to store the `\0` code.

You cannot assign more characters to the variable than you declare. That's because C sets aside just enough space to store the maximum number of characters. So think carefully before declaring the variable.

For example, suppose you need a variable to store a student's name. You declare it this way:

```
char student[10];
```

Your program might run correctly until the name of student is Mr. Mohd_Asif. When you try to assign this student's name Mr. Kamran_Bilgrami, your program will stop with a run-time error.

You could declare the variable with some large number, such as

```
char student[80];
```

But if you have several variables like this, you will be wasting quite a bit of your computer's resources.

A string variable can also be initialised within a declaration. To do so, the string is usually written without an explicit size specification (the square brackets are empty). The array name is then followed by an equal sign, the string (enclosed in quotes), and a semicolon, such as

```
char student[ ] = "Kamran_Bilgrami";
```

This declaration will cause `student` to be an 16-element character array. The first 15 elements will represent the 15 characters within the word `Kamran_Bilgrami`, and the 16th element will represent the null character (`\0`) which is automatically added at the end of the string.

CamScanner

Exercise

4.1. List five keywords and explain their use.

4.2. State the rules which must be observed while constructing identifiers.

4.3. Which of the following are not valid identifiers and why?

`2y` `yes` `ono_o_no` `00_go` `star*it` `me_to-2` `xYint` `int`

4.4. What are the keywords in C? What restrictions apply to their use?

4.5. Which data type is each of the following?

(a) TRUE
(b) 3958
(c) "&"
(d) 9
(e) "9"
(f) "This is my home."

4.6. Are these integers valid or invalid?

(a) 485
(b) 000
(c) 3854&
(d) −935
(e) 96,109

4.7. Are these valid floating-point numbers? If not, change them so they will be valid?

(a) +953.0
(b) −0.3874
(c) 6,209.85
(d) %385.86
(e) 395e+3

4.8. Determine which of the following are valid character constants?

(a) 't'
(b) "U'
(c) '#'
(d) '\n'
(e) '\876'
(f) '\\'

4.9. Determine which of the following are valid string constants?

(a) "1.7E+23"
(b) '4:40 A.M.'
(c) "STUDENT:
(d) "Chap-3 (cont\'d)"

CamScanner

4.10. What is a char data type?
4.11. How is the data type string different from the data type char?
4.12. How does the character '3' differ from the number 3?
4.13. What data type would you use for a variable storing area of a triangle?
4.14. Why would a program need to use the long int data type?
4.15. Why would you use a double float type?
4.16. Name and describe the four basic data types in C.
4.17. What is a variable?
4.18. What is the difference between a constant and a variable?
4.19. How do you declare a constant?
4.20. Does C have a string data type?
4.21. Describe two different ways that floating-point constants can be written. What special rules apply in each case?
4.22. Can a variable have the same value throughout the entire program?
4.23. How do you change the value of a constant?
4.24. Distinguish between the following:
  (a) float and double data type
  (b) constant and variable
  (c) integer and character constant
4.25. Decide what types of data you need, and write their declarations, for a program that computes the sum and average of four numbers.
4.26. Write C statement to declare a variable to represent the answer to a true/false question. Assume that true has a value of 1 and false has a value of 0.
4.27. Explain what is wrong in the following program segment:

```
char student[3] = "Rizwan"
void main (void)
float tax_rate;
char name (10);
int num [10]
tax_due = "Rs.1,284.00";
```

4.28. Write appropriate declaration for following variables and arrays.
  (a) Integer variables: a, b
  (b) Unsigned integer variable: students
  (c) Floating-point variables: x, y, z
  (d) Character variables: l, m, n
  (e) 60-element character array: message
4.29. Write appropriate declaration and assign the given initial values to following variables and arrays.
  (a) Integer variables: a = 485, b = −278
  (b) Unsigned integer variable: students = 96854
  (c) Floating-point variables: x = 0.4, y = 0.0005, z = 45.7
  (d) Double-precision variables: d1 = 1.68×10^{-6}

CamScanner

Objectives and MCQs

Choose the answer that best completes the statement.

4.1. `Char` constant values
- (a) must be enclosed in double quotes
- (b) must be enclosed in single quotes
- (c) can be enclosed in either single or double quotes
- (d) does not need to be enclosed in quotes

4.2. `int` constant values
- (a) can include an optional comma to improve readability
- (b) can include an optional dollar sign when representing money
- (c) can include an optional plus or minus sign
- (d) can consist only of digits

4.3. Symbolic constants
- (a) must appear in capital letters
- (b) are disliked by serious programmers
- (c) enhance the readability of a program
- (d) can include keywords

4.4. Variables
- (a) are created using the `#define` directive
- (b) are floating-point by default
- (c) should have simple names
- (d) can be defined as having any C data type

4.5. A variable definition differs from a declaration in that
- (a) the declaration sets aside storage space.
- (b) the declaration specifies the name and type of the variable.
- (c) the definition sets aside storage space.
- (d) the definition specifies the name and type of the variable.

Objectives

4.1. On a personal computer an integer is usually represented with two bytes.

4.2. Integer and floating-point data types represent numbers. They are often referred to collectively as numeric data type.

4.3. Constants are quantities whose values do not change during program execution.

4.4. In C some authors refer constants as literals or literal constants.

4.5. C allows us to work with string-type data through the use of arrays.

4.6. The characters of a string are stored in consecutive memory storage locations.

4.7. Names cannot contain special characters.

4.8. `#define` is called a pre-processor directive.

CamScanner

CHAPTER 5

OPERATORS AND EXPRESSIONS

By: Sir Ahad

The processing phase of a program converts input *data* into *information* that is output. This process of converting data into information takes many forms. When working with numbers, the process often involves some form of mathematical operation. *Operators* are needed to perform such type of calculations that transform data into information. An operator defines what is done with data. Therefore we can say that 'operators' are used to perform some 'operation' on the 'operands'. For example:

$$a = b + c$$

Here, b and c are operands, and the operation is being performed is 'addition' by using '+' operator.

Expressions

An expression is a collection of operands (one or more) and operators (zero or more) that can be evaluated to a single value. An operand is a constant, variable, or another expression. The most general form is

```
operand-1 operation operand-2
```

The result is a new value. For example:

```
2 + 6 evaluates to 8
4 * 5 evaluates to 20
48/8 evaluates to 6
10 - 3 evaluates to 7
18.4 - 4 evaluates to 14.4
```

CamScanner

Note that the use of whitespace (that is, blanks) has no effect on the evaluation of the expression.

```
6 + (7 + 6)(i) / (3*j/-k) - 84.27
```

The use of parentheses also improves the readability and guarantees the order of evolution according to precedence rules.

Classifications of Operators

Operators fall into three classes:

1. *Unary Operator:* Those that requires one operand.
2. *Binary Operator:* Thos that requires two operands.
3. *Ternary Operator:* This one is unique. It requires three operands.

C is very rich in operators and is sometimes called "the language of operators". It has as many as 45 different operators. Some commonly used operators are:

1. Arithmetic operators
2. Assignment operators
3. Relational operators
4. Logical/Boolean operators
5. Increment and Decrement operators

Arithmetic Operators

Arithmetic operators are used to perform mathematical operations. An arithmetic expression is made up of constants, variable, a combination of both or a function call, connected by arithmetic operators. Following list provide the detail about arithmetic operators.

Operator	Operation	Example
+	addition	a + b
–	subtraction	a – b
*	multiplication	a * b
/	division	a / b
%	remainder after integer division	a % b

The operands acted upon by arithmetic operators must represent numeric values. Thus, the operands can be integer quantities, floating-point quantities or characters (remember that character constants represent integer values, as determined by the computer's character set). The remainder operator (%) requires that both operands be integers and the second operand be nonzero. Similarly, the division operator (/) requires that the second operand be nonzero.

Division of one integer quantity by another is referred to as *integer division*. This operation always results in a truncated quotient (*i.e.*, the decimal portion of the quotient will be dropped). For example:

```
7/2     evaluates to    3
11/5    evaluates to    2
```

CamScanner

On the other hand, if a division operation is carried out with two floating-point numbers, or with one floating-point number and one integer, the result will be a floating-point quotient.

Suppose that `i` and `j` are integer variables whose values are `7` and `2`, respectively. Several arithmetic expressions involving these variables are shown below, together with their resulting values.

Expression	*Value*
`i + j`	`= 9`
`i - j`	`= 5`
`i * j`	`= 14`
`i / j`	`= 3`
`i % j`	`= 1`

Notice the truncated quotient resulting from the division operation, since both operands represent integer quantities. Also, notice the integer remainder resulting from the use of the remainder operator in the second expression.

Now suppose that `f` and `g` are floating-point variables whose values are `18.5` and `2.0`, respectively. Several arithmetic expressions involving these variables are shown below, together with their resulting values.

Expression	*Value*
`f + g`	`= 20.5`
`f - g`	`= 16.5`
`f * g`	`= 37.0`
`f / g`	`= 9.25`

Now again suppose that `c` and `d` are character-type variables that represent the characters `F` and `S`, respectively. Several arithmetic expressions involving these variables are shown below, together with their resulting values (based upon the ASCII character set).

Expression	*Value*
`c`	`= 70`
`c + d`	`= 153`
`c + d + 9`	`= 162`
`c + d + '9'`	`= 210`

Note that `F` is encoded as (decimal) `70`, `S` is encoded as `83`, and `9` is encoded as `57` in the ASCII character set, as shown in Appendix A.

If one or both operands represent negative values, then the addition, subtraction, multiplication and division operations will result in values whose signs are determined by the usual rules of algebra. Integer division will result in truncation toward zero; *i.e.*, the resultant will always be smaller in magnitude than the true quotient.

Unlike the other arithmetic operators, which accept both integer and floating point operands, the remainder operator (sometime called the modulus operator) accepts

CamScanner

only integer operands. The resulting value is the remainder of the first operand divided by the second operands. For example:

```
9 % 5 = 4
```

because 5 goes into 9 once with a remainder of 4. The expression:

```
10 % 5 = 0
```

because 5 goes into 10 two times with no remainder.

Order of Precedence

Order of precedence means the order in which the multiple arithmetic operations take place. The order of operation of arithmetic operators is as follows:

Operator	Operation	Order
*, /, %	multiplication/division/remainder	higher
+, −	addition/subtraction	lower

Another important consideration is the *order* in which consecutive operations within the same precedence group are carried out. This is known as *associativity*. Within each of the precedence groups described above, the associativity is left to right. In other words, consecutive addition and subtraction operations are carried out from left to right, as are consecutive multiplication, division and remainder operations.

Several arithmetic expressions are shown below whose results may have different values if the associativity is changed.

```
8 − 2 + 3   = 9
8*3 / 2     = 12
9/3 + 2 * 4 = 11
```

The natural precedence of operations can be altered through the use of parentheses. Whatever operations occur in parentheses is performed first following the order of precedence given in table above. In the event of nested parentheses (*i.e.* parentheses inside the parentheses), the expression enclosed by the innermost parentheses is performed first. For example, the expression 1 + ((3 + 1)/(8 − 4) − 5) is evaluated as shown in Fig. 5.1:

1 + ((3 + 1) / (8 − 4) − 5)	The innermost parentheses are evaluated first. The expressions (3+1) and (8−4) are at the same depth, so they can be evaluated in either order.
1 + (4 / (8 − 4) − 5)	
1 + (4 / 4 − 5)	Division has a higher precedence than subtraction.
1 + (1 − 5)	
1 + −4	
−3	Final result

Fig. 5.1. Evaluation of an expression enclosed by parentheses

CamScanner

Several arithmetic expressions are shown below where parentheses override the order of precedence.

```
3 * (4 + 2)                  = 18
(6 + 3)/(1 + 2)              = 3
2 * (1 + 6/2) / (3 + 1)      = 2
```

Sometimes it is a good idea to use parentheses to clarify an expression, even though the parentheses may not be required. On the other hand, the use of overlay complex expressions, such as

```
2 * ((5 % i) * (4 + (j - 3)/(k+2)))
```

should be avoided if at all possible. Such expressions are difficult to read, and they are often written incorrectly because of unbalanced parentheses.

Example 5.1

Given the following declarations:

Integer variables	`i = 3, j = 4`
Floating-point variables	`f = 2.5, g = 1.0`

Evaluate the values for the following expressions:

1. *i + j + f + g*
2. *i + f * j + g*
3. *f / g + i / j*
4. *f − g * i + g / j*
5. *f / 0*

Solution

1. The expression *i + j + f + g* is equal to 3 + 4 + 2.5 + 1.0 = 10.5
2. The expression *i + f * j + g* is equivalent to (*i* + (*f* * *j*) + *g*) because the precedence of * is higher than that of +. Substituting the values of the variables we get: (3 + (2.5 * 4) + 1.0) = 14.0
3. The expression *f*/*g* + *i*/*j* is equivalent to (*f*/*g*) + (*i*/*j*). Thus substituting the values of *f*, *g*, *i* and *j*, we get:

   (2.5/1.0) + (3/4) = 2.5

   Note that the value of 3/4 is calculated to be 0. Because the division of two integer values 3/4 does not give any integer quotient. Hence it is taken to be zero.
4. The expression *f* − *g* * *i* + *g*/*j* is equivalent to (*f* − (*g* * *i*)) + (*g*/*j*). Thus substituting the values of *f*, *g*, *i* and *j*, we get:

   (2.5 − (1.0*3)) + (1.0/4) = 2.5 − 3.0 + 0.25 = −0.25
5. *f*/0 is undefined. Any number divided by zero is infinite and hence the value is not defined.

Assignment Operators

The basic assignment operator is =. This is used to assign value of an expression to an identifier. It has the general form

```
identifier = expression
```

CamScanner

where identifier generally represents a variable, and expression represents a constant, a variable or a more complex expression.

Some examples of expressions that make use of the = operator are:

Expression	*Meaning*
`a = 5`	assign integer value 5 to variable a
`x = y`	assign the value of y to x
`sum = 25.23`	assign floating point value 25.23 to sum
`a = b + c`	assign the sum value of b and c to a
`area = length * breath`	assign the multiplicative value of length and breath to area

The assignment operator has right-to-left associativity, so it causes the value of the right-hand operand to be written into the memory location of the left-hand operand. Hence the expression

```
a = b = c = d = 1;
```

is interpreted as

```
(a = (b = (c = (d = 1))));
```

First the value 1 is assigned to d, then d is assigned to c, then c is assigned to b, and finally b is assigned to a. The value of the entire expression is 1. This is a convenient syntax for assigning the same value to more than one variable. Note, however, that each assignment may cause quiet conversions, so

```
int i;
double f;
f = i = 3.5;
```

assigns the truncated value 3 to both f and i. On the other hand,

```
i = f = 3.5;
```

assigns 3.5 to f and 3 to i.

In addition to =, there are a number of assignment operators in C. Some of them are

Operator	Operation	Example	Explanation
+=	add-assign	a += b	same as a = a + b
−=	subtract-assign	a −= b	same as a = a − b
*=	multiply-assign	a * =b	same as a = a * b
/=	divide-assign	a / =b	same as a = a / b
%=	remainder-assign	a % =b	same as a = a % b

For example, the expression

```
i = i * 5;
```

can be written as

```
i *= 5;
```

One of the main reasons for using the arithmetic assignment operators is to avoid spelling mistakes and make code more readable. For example, the expression

```
number_of_students_in_class = number_of_students_in_class * 2;
```

CamScanner

can be written as

```
number_of_students_in_class *= 2;
```

The second version is easier to read and to write and contains fewer opportunities for spelling errors.

Now suppose that `i` and `j` are integer variables whose values are 5 and 7, and `f` and `g` are floating-point variables whose values are 5.5 and −3.25 respectively. Several arithmetic expressions involving these variables are shown below, together with their resulting values. Each expression utilises the original values of `i`, `j`, `f` and `g`.

Expression	*Equivalent Expression*	*Final Value*
`i += 5`	`i = i + 5`	`= 10`
`f -= g`	`f = f - g`	`= 8.75`
`j *= (i - 3)`	`j = j * (i - 3)`	`= 14`
`f /= 3`	`f = f / 3`	`= 1.833333`
`i %= (j - 2)`	`i = i % (j - 2)`	`= 0`

The assign operators have relatively low precedence. This leads to interesting consequences. For example, the following two expression are not the same:

```
i = i * 3 + 4;
i *= 3 + 4;
```

The addition operator has higher precedence than the assign operator, and the multiplication operator has higher precedence than the addition operator, so the two expressions are interpreted as follows:

Expression 1	*Expression 2*
`i = i * 3 + 4;`	`i *= 3 + 4;`
`or i = ((i * 3) + 4);`	`or i *= (3 + 4);`
	`or i = (i * (3 + 4));`

Example 5.2

Given the following declarations:

Integer variables `i = 3, j = 4`

Floating-point variables `f = 2.5, g = 1.0`

Evaluate the values for the following expressions:

1. $i \mathrel{+}= j + f - g$
2. $i \mathrel{/}= f * j + g$
3. $j \mathrel{\%}= g + i$
4. $f \mathrel{+}= g \mathrel{-}= i$

Solution

1. The expression $i \mathrel{+}= j + f - g$ is equivalent to $i = i + ((j + f) - g)$. Substituting the values of the variables, we get the result to be 8.
2. The expression $i \mathrel{/}= f * j + g$ is equivalent to $i = i / ((f * j) + g)$. Substituting the values of the variables we get the result to be 0.
3. The expression $j \mathrel{\%}= g + i$ is equivalent to $j = j \% (g + i)$. Substituting the values of the variables, we get the result to be 0.

CamScanner

4. The expression $f \mathrel{+}= g \mathrel{-}= i$ is equivalent to $f = f + (g = (g - i))$. Substituting the values of the variables, we get the result to be 0.5.

Relational Operators

Relational operators are used to compare two numeric operands. The operands can be variables, constants or expressions that ultimately get evaluated to a numerical value. Since characters can be represented as integers using ASCII code, therefore, operands to be compared can be characters as well. All C relational operators are binary operators and hence require *two* operands. The relational operators are summarised as:

Operator	Operation	Example
<	less than	a < b
>	greater than	a > b
<=	less than or equal to	a <= b
>=	greater than or equal to	a >= b
==	equal to	a == b
!=	not equal to	a != b

A relational expression is made up of two arithmetic expressions connected by a relational operator. It returns `0` when the relation is *false* and `1` when it is *true*.

These relational operators all fall within the same precedence group, which is lower than the arithmetic operators. For example, the expression:

```
a + b * c < d / f
```

is evaluated as if it had been written as follows:

```
(a + (b * c)) < (d / f)
```

The associativity of these operators is left to right.

Suppose the `i`, `j` and `k` are integer variables whose values are `1`, `2` and `3`, respectively. Several expressions using relational operators involving these variables are shown below:

Expression	*Interpretation*	*Value*
`i < j`	true	1
`(i + j) >= k`	true	1
`(j + k) > (i + 5)`	false	0
`k != 3`	false	0
`j == 2`	true	1

Now suppose that `i` is an integer variable whose value is `7`, `f` is a floating-point variable whose value is `5.5`, and `c` is a character variable that represents the character `'w'`. Several logical expressions that make use of these variables are shown below. Each expression involves two different type of operands. (Assume that the ASCII character set applies.)

Expression	*Interpretation*	*Value*
`f > 5`	true	1
`(i + f) <= 10`	false	0
`c == 119`	true	1
`c != 'p'`	true	1
`c >= 10 * (i + f)`	false	0

CamScanner

Example 5.3

Given the following declarations:

Integer variables `i = 0, j = 1, k = -1`

Floating-point variables `f = 2.5, g = 0.0`

Evaluate the values for the following expressions:

1. $i > j$
2. $j / k < f$
3. $i <= j >= k$
4. $i <= f == j$
5. $-f + i == g > k >= j$
6. $f += (g >= k)$
7. $i + 1 == j != g * 2$

Solution

1. $i > j$ is not true because the value of $i = 0$ and value of $j = 1$. Hence the result is 0 (false).
2. The expression $j / k < f$ is equivalent to $(j / k) < f$ because the precedence of / is higher than that of <. Substituting the values of the variables, we get the result to be 1 (true).
3. The expression $i <= j >= k$ is equivalent to $(i <= j) >= k$. Substituting the values of the variables we get the result to be 1 (true).
4. The expression $i <= f == j$ is equivalent to $(i <= f) == j$. Substituting the values of the variables, we get the result to be 1 (true).
5. The expression $-f + i == g > k >= j$ is equivalent to $((-f) + i) == ((g > k) >= j)$. Substituting the values of the variables we get the result to be 0 (false).
6. The expression $f += (g >= k)$ is equivalent to $f = (f + (g >= k))$. Substituting the values of the variables we get the result 2.5 = 3.5. Thus the value of the expression is 0 (false).
7. The expression $i + 1 == j != g * 2$ is equivalent to $((i + 1) == j) != (g * 2)$. Substituting the values of the variables we get the result to be true (1). The term $(i + 1 == j)$ is evaluated to be 1 because the value of $i + 1$ is 1, and the value of $j = 1$. Therefore, $(i + 1 == j)$ is true or the value of the term is 1. The value of the term $(g * 2)$ is 0 * 2 is zero. Hence the resultant expression amounts to (1 != 0) and this is true. Thus the value of the expression is 1.

Confusing Equality (==) and Assignment (=) Operators

There is one very common type of error that C programmers make. The error is the swapping of operators == (equality) and = (assignment). The more damaging thing is that with this error program may run correctly but probably generate incorrect result.

There are two aspects of this problem. One is that any expression in C that produces a value can be used for decision-making. If the value is 0, it is treated as false, and if

the value is nonzero, it is treated as true. The second is that assignments in C produce a value, namely the value that is assigned to the variable on the left side of the assignment operator. For example, suppose we intend to write

```
if (pay_code == 4)
    printf("Your get a bonus!");
```

but we accidentally write

```
if (pay_code = 4)
    printf("Your get a bonus!");
```

The first `if` statement properly awards a bonus to the person whose pay code is equal to 4. The second `if` statement—the one with the error—evaluates the assignment expression in the `if` condition. This expression is a simple assignment whose value is the constant 4. Because any nonzero value is interpreted as "true", the condition in this `if` statement is always true, and the person always receives a bonus regardless of what the actual pay code is!

Logical/Boolean Operators

A logical operator also called Boolean operator to honour the Irish mathematician George Boole (1815-1864) combines the results of one or more expressions and the resultant expression is called the *logical expression*. After testing the conditions, they return either true (non-zero value) or false (0) as net result.

Fig. 5.1 George Boole

The logical operators are unary or binary operators. The operands may be constants, variables or expressions. The basic logical operators are:

Operator	Operation	Example
&&	Logical AND	1 if a = b && c = d; else 0
||	Logical OR	1 if a = b || c > d; else 0
!	Logical NOT	1 if !a; else 0

Logical AND operator (&&) takes two operands. The result of operation will be *true* (nonzero) only if both operands are *true* (nonzero) otherwise *false* (zero). The table 5.1 shows all the four possible combinations of *false* (zero) and *true* (nonzero) values of operands. Such tables are often called *truth tables*.

Table 5.1. Truth Table for logical AND (&&) operator

First Operand	Operator	Second Operand	Result
false (0)	&&	false (0)	0
true (nonzero)	&&	false (0)	0
false (0)	&&	true (nonzero)	0
true (nonzero)	&&	true (nonzero)	1(nonzero)

Logical OR operator (||) also takes two operands. The result of operation will be *true* (nonzero) if either operand is *true* or if both operands are *true* (nonzero) otherwise *false* (zero). In other words, the result of a logical OR operation will be *false* only if

both operands are *false* (zero). The table 5.2 shows all the four possible combinations of *false* (zero) and *true* (nonzero) values of operands.

Table 5.2. Truth Table for logical OR (||) operator

First Operand	Operator	Second Operand	Result
false (0)	||	false (0)	0
true (nonzero)	||	false (0)	1(nonzero)
false (0)	||	true (nonzero)	1(nonzero)
true (nonzero)	||	true (nonzero)	1(nonzero)

Logical NOT operator (!) takes only one operand. If the operand is *true* (nonzero), the result is *false* (zero); if the operand is *false*, the result is *true*. The table 5.3 shows these two possibilities.

Table 5.3. Truth Table for logical NOT (!) operator

Operand	Operator	Result
false (0)	!	1(nonzero)
true (nonzero)	!	0

Each of the logical operators falls into its own precedence group. The logical NOT (!) operator has a higher precedence than the other two logical operators. Logical AND (&&) has a higher precedence than logical OR (||). The precedence of logical operators is lower than the relational and arithmetic operators. The associativity is left to right.

The operands to the logical operators may be integers or floating-point variables. Suppose that `i` is an integer variable whose value is `7`, `f` is a floating-point variable whose value is `5.5`, and `c` is a character variable that represents the character `'w'`. Several complex logical expressions that make use of these variables are shown below.

Expression	*Interpretation*	*Value*
`(i >= 6) && (c == 'w')`	true	1
`(i >=6) || (c == 119)`	true	1
`(f < 11) && (i > 100)`	false	0
`(c != 'p') || ((i + f) <= 10)`	true	1

The first expression is true because both operands are true. In the second expression, both operands are again true; hence the overall expression is true. The third expression is false because the second operand is false. And finally, the fourth expression is true because the first operand is true.

Now suppose that `i` is an integer variable whose value is `7`, and `f` is a floating-point variable whose value is `5.5`. Several logical expressions which make use of these variables and the logical NOT (!) operator are shown below:

Expression	*Interpretation*	*Value*
`f > 5`	true	1
`!(f > 5)`	false	0
`i <= 3`	false	0
`!(i <= 3)`	true	1
`i > (f +1)`	true	1
`!(i > (f +1))`	false	0

CamScanner

Example 5.4

Given the following declarations:

Integer variables `i = 0, j = 1, k = -1`

Floating-point variables `f = 2.5, g = 0.0`

Evaluate the values for the following expressions:

1. *i && j*
2. *i < j && k < j*
3. *j + k = 1 || !i*
4. *f*5 && 5 || j/k*
5. *i <= 10 && f >= 1 && j*
6. *!f || !k || j + k*
7. *f*g < i + j || k*

Solution

1. *i && j* is not true because the value of *i = 0* and the value of *j = 1*. Hence the result is 0 (false).
2. The expression *i < j && k < j* is equivalent to *(i < j) && (k < j)* because the precedence of relational operators is higher than that of logical operators. Substituting the values of the variables we get the result to be 1 (true).
3. The expression *j + k || !i* is equivalent to *(j+k) || (!i)*. Substituting the values of the variables we get the result to be 1 (true).
4. The expression *f*5 && 5 || j/k* is equivalent to *((f*5) && 5) || (j/k)*. Substituting the values of the variables, we get the result to be 1 (true).
5. The expression *i <= 10 && f >= 1 && j* is equivalent to *((i<=10) && (f >= 1)) && j*. Substituting the values of variables, we get the result to be 1 (true).
6. The expression *!f || !k || j + k* is equivalent to *((!f) || (!k)) || (j + k)*. Substituting the values of the variables, we get the result to be 0 (false).
7. The expression *f*g < i + j || k* is equivalent to *((f*g) < (i + j)) || k*. Substituting the values of variables, we get the result to be 1 (true).

Increment and Decrement Operators

The increment operator `++` and decrement operator `--` are very useful operators and are not found in other languages. If a variable a is incremented by 1, the increment operator `++` can be used rather than the expression `a = a + 1` or `a += 1`. The increment operator causes its operand to be increased by 1, whereas the decrement operator causes its operand to be decreased by 1.

These are 'unary' operators since they operate on only one operand. The operand has to be a variable and not a constant. Thus, the expression `a++` is valid whereas `6++` is invalid.

The increment and decrement operators can be of two types. These are:

1. First increment or decrement the value of the variable and then take this new value for processing (*prefix*).

CamScanner

2. First take the value of the variable for processing and then only increment or decrement the variable (*pos'fix*).

The increment and decrement operators (postfix and prefix) are summarised as:

Operator	Example	Operation	Explanation
++	++a	prefix increment	Increment a by 1 then use the new value of a in the expression in which a resides.
++	a++	postfix increment	Use the current value of a in the expression in which a resides, then increment a by 1.
--	--a	prefix decrement	Decrement a by 1 then use the new value of a in the expression in which a resides.
--	a--	postfix decrement	Use the current value of a in the expression in which a resides, then decrement a by 1.

To illustrate these operations, suppose that i and j are integer variables whose values are 3 and 5 respectively. Several expressions that make use of these variables are shown below.

```
int i, j;          /* declare */
i = 3, j = 5;      /* initialise */

j = i++;           /* j now = 3 and i = 4 postfix increment */
j = ++i;           /* j now = 5 and i = 5 prefix increment */
j = i--;           /* j now = 5 and i = 4 postfix decrement */
j = --i;           /* j now = 3 and i = 3 prefix decrement */
```

If you just want to increment or decrement without any assignment, the postfix and prefix methods are effectively equivalent.

```
i++;        /* set i to i+1 */
++i;        /* set i to i+1 */
i--;        /* set i to i-1 */
--i;        /* set i to i-1 */
```

Example 5.5

Subtract 1 from a variable x in four different ways.

Solution

```
assignment:                x = x - 1;
compound assignment:       x -= 1;
decrement (postfix):       x--;
decrement (prefix)         --x;
```

Example 5.6

Given the following declarations:

Integer variables `i = 0, j = 1, k = -1`

Floating-point variables `f = 2.5, g = 0.0`

Evaluate the values for the following expressions:

CamScanner

1. `++i == j != g*2`
2. `(f > g) + !i || k++`
3. `(i || j) + (f || ++k)`

Solution

1. The expression `++i == j != g*2` is equivalent to `((++i) == j) != (g*2)`. Substituting the values of the variables we get the result to be true (1). The term `(++i == j)` is evaluated to be 1 because the value of `++i` is 1, and the value of `j = 1`. Therefore, `(++i == j)` is true or the value of the term is 1. The value of the term `(g*2)` is zero because `0*2` is zero. Hence the resultant expression amounts to `(1 != 0)` and this is true.
2. The expression `(f > g) + !i || k++` is equivalent to `((f > g) + (!i)) || (k++)`. Substituting the values of the variables, we get the result to be true (1).
3. The expression `(i || j) + (f || ++k)` is equivalent to `(i || j) + (f || (++k))`. Substituting the values of the variables, we get the result to be 2. Note here that `++k` represents the prefix increment value of `k`.

Conversions of Data Type

Suppose that an arithmetic expression such as `i + j` has both a value and a data type. For example, if both `i` and `j` have type `int`, then the expression `i + j` also has type `int`. But if `i` has type `int` and `j` has type `float`, the `int` would be automatically converted to a `float` before carrying out the operation. Some other conversion rules are shown in following tables:

First Operand	Second Operand	Result
char	char	char
char	int	int
char	long int	long int
char	float	float
char	double	double

First Operand	Second Operand	Result
int	char	int
int	int	int
int	long int	long int
int	float	float
int	double	double

First Operand	Second Operand	Result
long int	char	long int
long int	int	long int
long int	long int	long int
long int	float	float
long int	double	double

First Operand	Second Operand	Result
float	char	float
float	int	float
float	long int	float
float	float	float
float	double	double

First Operand	Second Operand	Result
double	char	double
double	int	double
double	long int	double
double	float	double
double	double	double

CamScanner

To illustrate the idea of automatic conversion, we take the following examples:

Operation	Result	Operation	Result
5/2	2	2/5	0
5.0/2	2.5	2.0/5	0.4
5/2.0	2.5	2/5.0	0.4
5.0/2.0	2.5	2.0/5.0	0.4

Casts

Sometimes we are required to convert the value of an expression to a particular data type. To do so, the expression must be preceded by the name of the desired data type, enclosed in parentheses *i.e.*

```
(data type) expression
```

This type of construction is known as *cast*. This would be clear from the following segment of program:

```
float f;
int i = 6, j = 4;
f = i/j;
```

The output of this segment of program is `f = 1.000000` and not 1.5. This is because, 6 and 4 are both integers and hence 6/4 yields an integer, 1. This 1 when stored in `f` is converted to 1.000000. But what if we do not want the quotient to be truncated. One solution is to make either `i` and `j` as `float`. Let us say that other requirements of the program does not permit us to do this. In such a case what do we do? Use cast. The following segment of program illustrate this:

```
float f;
int i = 6, j = 4;
f = (float) i/j;
```

and we will get the output `f = 1.500000`. In this program `(float)` causes the variable `i` to be converted from type `int` to type `float` before being used in the division operation.

Some other examples of casts are:

```
(long)('A' + 1.0)
i = (float) ((int) j + 1)
(double) (x = 45)
```

but not

```
(double) x = 45  /*equivalent to ((double) x) = 77, error*/
```

The cast operator is a unary operator having the same precedence and right-to-left associativity as other unary operators. Thus the expression

`(float) i + 3` is equivalent to `((float) i) + 3`

because the cast operator has higher precedence than +.

Table 5.4. gives the summary of the precedence, for highest to lowest, for various operators including their associativity.

CamScanner

Table 5.4. Precedence of Various Operators

Operators	Type	Associativity
++ -- ! casts	unary	right to left
* / %	multiplicative	left to right
+ -	additive	left to right
< > <= >=	relational	left to right
== !=	equality	left to right
&&	logical AND	left to right
||	logical OR	left to right
= *= /= %= += -=	assignment	right to left

Library Functions

The standard library contains many useful functions called *library functions* that carry out various commonly used operations or calculations. The ANSI C standard specifies a minimum number of library functions to be supplied by all C compilers, which your program may use.

The standard library contains library functions to perform standard input/output operations (*e.g.*, read and write characters, read and write numbers, open and close files, test for end of file, etc.), functions that perform operations on characters (*e.g.*, convert from lower- to uppercase, test to see if a character is uppercase, etc.), functions that perform operations on strings (*e.g.*, copy a string, compare strings, concatenate strings, etc.), functions that carry out various mathematical calculations (*e.g.*, evaluate trigonometric, logarithmic and exponential functions, compute absolute values, square roots, etc.), and much more.

When your program is compiled, the code for library functions is automatically added to your program. In most other computer languages, like BASIC or Pascal, these functions are built into the language. The advantage of having them as library functions is increased flexibility. As you will see, virtually all C programs you create will use functions from the C standard library.

The most common library functions provided with most C compilers are listed in table 5.5. The column labelled "type" in this table refers to the data type of the quantity that is returned by the function.

Table 5.5. Some Commonly used Library Functions

Function	Type	Purpose	include File
`abs(i)`	int	Return the absolute value of `i`.	stdlib.h
`cos(d)`	double	Return the cosine of `d`.	math.h
`cosh(d)`	double	Return the hyperbolic cosine of `d`.	math.h
`exp(d)`	double	Raise `e` to the power `d` (`e` = 2.7182818 ... is the base of the natural logarithm).	math.h
`fabs(d)`	double	Return the absolute value of `d`.	math.h
`floor(d)`	double	Round down to the next integer value (the largest integer that does not exceed `d`).	math.h

CamScanner

`fmod(d1,d2)`	double	Return the remainder (*i.e.*, the non-integer part of the quotient) of `d1/d2`, with same sign as `d1`.	math.h
`getchar()`	int	Enter a character from the standard input device.	stdio.h
`log(d)`	double	Return the natural logarithm of `d`.	math.h
`log10(d)`	double	Return the logarithm (base 10) of `d`.	math.h
`pow(d1,d2)`	double	Return `d1` raised to the `d2` power.	math.h
`printf(...)`	int	Send data items to the standard output device	stdio.h
`putchar(c)`	int	Send a character to the standard output device.	stdio.h
`rand()`	int	Return a random positive integer.	stdlib.h
`sin(d)`	double	Return the sine of `d`.	math.h
`sqrt(d)`	double	Return the square root of `d`.	math.h
`srand(u)`	void	Initialize the random number generator.	stdlib.h
`scanf(...)`	int	Enter data items from the standard input device.	stdio.h
`tan(d)`	double	Return the tangent of `d`.	math.h
`toascii(c)`	int	Convert value of argument to ASCII.	ctype.h
`tolower(c)`	int	Convert letter to lowercase.	ctype.h
`toupper(c)`	int	Convert letter to uppercase.	ctype.h

Note:

Type refers to the data type of the quantity that is returned by the function.

- `c` denotes a character-type argument
- `i` denotes an integer argument
- `d` denotes a double-precision argument
- `u` denotes an unsigned integer argument

To call a function, you specify its name followed by a parenthesised list of arguments that you will be passing to it. If the function does not require any arguments, no arguments will be specified—and the parenthesised list will be empty. If there is more than one argument, the arguments must be separated by commas.

Another component common to most C programs is the *header file*. In C, information about the standard library functions is found in various files supplied with your compiler. The files all end with a `.h` extension. The C compiler uses the information in these files to handle the library functions properly. You add these files to your program using the `#include` compiler *pre-processor directive*. All C compilers used as their first phase of compilation a pre-processor, which performs various manipulations on your source file before it is actually compiled. Pre-processor directives are not actually part of the C language, but rather instructions from you to the complier. The `#include` directive tells the pre-processor to read in another file and include it with your program. The most commonly used header files are `stdio.h`, `conio.h`, `math.h` and `stdlib.h`.

CamScanner

Exercise

5.1. What is an operator? Describe several different types of operators.

5.2. What is an operand? What is the relationship between operators and operands?

5.3. Describe the five arithmetic operators. Summarise the rules associated with their use.

5.4. What is the difference between the operators / and %?

5.5. What is meant by operator precedence? What are the relative precedences of the arithmetic operators?

5.6. What is meant by associativity? What is the associativity of the arithmetic operators?

5.7. When should parentheses be included within an expression? When should the use of parentheses be avoided?

5.8. In what order are the operations carried out within an expression that contains nested parentheses?

5.9. Suppose a, b and c are integer variables that have been assigned the values $a = 8$, $b = 3$ and $c = -5$. Determine the value of each of the following arithmetic expressions.

1. $a + b + c$ (Ans. 6)
2. $2 * b + 3 * (a - c)$ (Ans. 45)
3. a / b (Ans. 2)
4. a / c (Ans. −1)
5. $a \% b$ (Ans. 2)
6. $a * b / c$ (Ans. −4)
7. $a * (b / c)$ (Ans. 0, because b / c is zero)

5.10. Suppose a, b and c are floating-point variables that have been assigned the values $a = 8.8$, $b = 3.5$ and $c = -5.2$. Determine the value of each of the following arithmetic expressions.

1. $a + b + c$ (Ans. 7.1)
2. $2 * b + 3 * (a - c)$ (Ans. 49)
3. a / b (Ans. 2.51429)
4. a / c (Ans. −1.69231)
5. $a \% b$ (Ans. The remainder operator is not defined for floating-point operands.)
6. $a / (b + c)$ (Ans. −5.17647)
7. $(a / b) + c$ (Ans. −2.68571)
8. $2 * a / 3 * b$ (Ans. 20.53333)
9. $2 * a / (3 * b)$ (Ans. 1.67619)

5.11. Describe the six relational operators. With what type of operands can they be used? What type of expression is obtained?

CamScanner

5.12. Describe the three logical operators. What is the purpose of these operators? With what type of operands can they be used? What type of expression is obtained?

5.13. Write the logical statements for the following:
(a) basic_pay is more than 2200 for all employees whose category is P.
(b) stock of all items whose item_no is 100 and in the shop A.

5.14. What are the relative precedences of the relational and logical operators with respect to one another and with respect to the arithmetic operators?

5.15. Describe the logical NOT operator. What is its purpose? Within which precedence group is it included? How many operands does it require?

5.16. Explain the difference between prefix and postfix increment and how does it affect the result of an expression?

5.17. Explain the difference between `count=count+1` and `count++`.

5.18. What is the difference between `--count` and `count--`.

5.19. Can you mix different types of data in the same operation? If so, how will it affect the results of the operations.

5.20. What is the result after the execution of the following program segment.

```
a = 7;
b = ++a + 5;
c = b-- + 10;
```

5.21. What is an expression?

5.22. Evaluate following expressions independent of each other. Assume that `i, j, k` are integers with values 3, 4, and 2 respectively.

(a). `i++ - j--`
(b) `i-- % j++`
(c) `j++ / i--`
(d) `k++ * --i`
(e) `i - 1 % j + 1`

5.23. Write C expressions corresponding to the following. Assume all quantities are of the type integer.

(a) $\dfrac{(ax+b)}{(ax-b)}$

(b) $\dfrac{(2x+3y)}{(x^3-6)}$

(c) $x^5+10x^4+8x^3+4x+2$

(d) $\dfrac{(4a+3)}{(2y^2+2z+2)}$

5.24. What is the final value of `f` in the following program segment:

```
float f;
int i;
f = 2.5;
f = (f + 0.04) * 10;
```

5.25. What does the cast operator do? Why is it useful?

CamScanner

Objectives and MCQs

Choose the answer that best completes the statement.

5.1. Which of the following is an example of initialising a variable?
- (a) num = 2;
- (b) int num;
- (c) num < 2;
- (d) int num = 2;

5.2. Floating point variables are used instead of integers to
- (a) avoid being too specific about what value a number has
- (b) permit the use of decimal points in numbers
- (c) make possible the use of even numbers
- (d) conceal the true value of the numbers

5.3. Which of the following is an arithmetic operator?
- (a) +
- (b) &
- (c) &&
- (d) <

5.4. Are the following expressions true or false?
- (a) `1 > 2`
- (b) `'a' < 'b'`
- (c) `1 == 2`
- (d) `'2' == '2'`

5.5. Precedence determines which operator
- (a) is most important
- (b) is used first
- (c) is fastest
- (d) operates on the largest number

5.6. A relational operator is used to
- (a) combine values
- (b) compare values
- (c) distinguish different types of variables
- (d) change variables to logical values

5.7. The standard library contains many useful functions called
- (a) initial functions
- (b) basic functions
- (c) library commands
- (d) library functions

CamScanner

Objectives

5.1. An expression is a collection of operands and operators.

5.2. Operators are used to perform some operation on the operands.

5.3. Unary operator requires one operand.

5.4. Arithmetic operators are used to perform mathematical operations.

5.5. Order of precedence means the order in which the multiple operations t
place.

5.6. After testing the conditions, logical operators return true or false as net resu

5.7. Logical NOT operator takes only one operand.

5.8. The precedence of logical operators is lower than the relational operators.

5.9. C is very rich in operators and is sometimes called the language of operator

CamScanner

By:Sir Ahad

CHAPTER 6

INPUT AND OUTPUT STATEMENTS

By Sir Ahad

Input and output statements of computer languages are used to provide communications between the user and the program. In *Turbo C++*, the input and output operations are carried out through built-in functions. Through these functions data can be read from keyboard or displayed on monitor. The execution of the program can be interrupted by input/output calls. Hence data can be entered or output can be taken during execution of program.

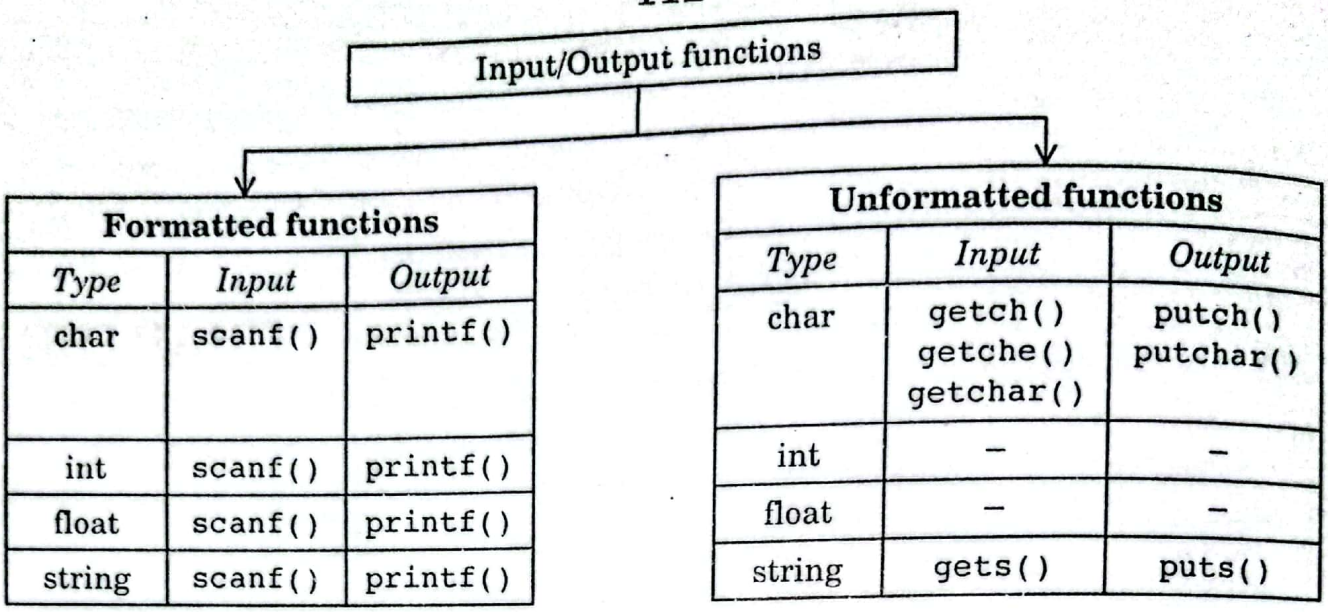
C makes use of a collection of libraries. Library functions and commands are only included in the program if actually needed. The result is fast, compact code. The standard input/output library is included via a header file called `stdio.h`. The header file is not actually the input/output library, but a set of declarations and definitions that add features to the I/O library. Almost every C program begins with the statement

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

This statement instructs the compiler to include the header file for compilation. The use of angle (*i.e.*, <>) brackets means to either look the header file in the root directory or to begin looking for the header file in the user's directory.

Input/output functions can be classified as:

CamScanner



Formatted functions

Type	*Input*	*Output*
char	`scanf()`	`printf()`
int	`scanf()`	`printf()`
float	`scanf()`	`printf()`
string	`scanf()`	`printf()`

Unformatted functions

Type	*Input*	*Output*
char	`getch()` `getche()` `getchar()`	`putch()` `putchar()`
int	–	–
float	–	–
string	`gets()`	`puts()`

The basic difference between formatted and unformatted I/O functions is that the formatted functions allow the input read from keyboard or the output displayed on the monitor to be formatted as per out requirements (the `f` in the `scanf` or `printf` stands for formatted). For example, if values of total marks and percentage marks are to be displayed on the screen, then the details like where this output would appear on the screen of the monitor, how many spaces would be present between the two values, the number of places after the decimal points etc. can be controlled using formatted function. Now we will discuss I/O functions in detail.

Unformatted I/O Functions

There are several standard library functions available for unformatted input and output. Some of them deal with a single character and some can deal with a string of characters. Few more famous input/output functions are discussed below.

Single Character Input

There are some functions in C that handle characters only. These functions are useful in some applications when the user is prompted to answer yes/no, or ask you to select from a menu of items, such as the number of the choice or the first letter of the choice name.

Three most common input functions are: the original `getchar()` function which was introduced by Ritchie and Kernighan, and the new functions `getch()` and `getche()` which were added later to suit the microcomputer interactive environment. The three functions are used without arguments, but you must keep the empty parentheses.

The `getchar()` Function

The function `getchar()` stands for "get character" and inputs a single character from the keyboard. When called, the function waits for a key to be pressed. You can input the character as either a `char` or `int` type because `getchar()` function treat characters as integers and accept the ASCII value of character as a variable of type `int`.

Therefore, to input a character, use either of these formats:

CamScanner

```
int letter;
letter = getchar();
```

or

```
char letter;
letter = getchar();
```

Notice that the function does not require any arguments, through a pair of empty parentheses must follow the word `getchar`.

When the user presses a key, `getchar()` echoes (displays) the character that you typed to the screen and further it requires Enter key ↵ to be typed following the input character.

The following program demonstrates the use of the function `getchar()`. It begins with declaring a character variable "ch", then the function itself is assigned to this variable. This means that the character received by the function will be contained in the variable "ch". The first `printf()` function displays the contents of the variable "ch" in the character format (%c) preceded by the string "You typed:". The second `printf()` function displays the same variable in the decimal format (%d) preceded by the string "which corresponds to the ASCII:".

```
/* program 6-1.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  char ch;
  ch = getchar();          /* read a character */
  printf("You typed: %c",ch);
  printf("\nwhich corresponds to the ASCII: %d",ch);

  getch();
}
```

A sample run of this program gives the output:

```
w
You typed: w
which corresponds to the ASCII: 119
```

The function `getchar()` can also be assigned to a variable of the type `int`. Just replace the word `char` by the word `int` in the above program and the program still gives the same results.

```
/* program 6-2.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  int ch;
```

CamScanner

```
  ch = getchar();          /* read a character */
  printf("You typed: %c",ch);
  printf("\nwhich corresponds to the ASCII: %d",ch);

  getch();
}
```

Notice that if you enter more than one character, the function `getchar()` still reads the first one only. Also, if you press Enter key ↵ only, you get the number "10", which is the ASCII code for the new-line character.

The `getch()` and `getche()` Functions

We often want a function that inputs a single character the instant it is typed without waiting for the Enter key ↵ to be hit. For instance, in a game we might want an aeroplane to move each time we pressed one of the cursor-control (arrow) keys; it would be awkward to type Enter key ↵ each time we pressed an arrow key.

The `getch()` and `getche()` are two very similar function which serve this purpose. The "get" means it gets some input and "ch" means it gets a character. The difference between two functions is that the function `getche()` echoes (displays) the character that you typed to the screen (the letter "e" in `getche()` stands for "echo") whereas `getch()` just returns the character that you typed without echoing it on the screen. Following program demonstrates the difference between two functions:

```
/* program 6-3.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  char ch;
  /* The character is displayed when pressed */
  printf("\nMake a choice and press a character:");
  ch=getche();

  /* The character is not displayed when pressed */
  printf("\nMake a choice and press a second character:");
  ch=getch();
}
```

If you run this program, you will notice that the phrase "Make a choice and press a second character" is printed immediately when you press any character key; you don't have to press Enter key ↵.

When you use the function `getch()` or `getche()` in a program, you must include the header file `conio.h` (conio stands for *console input/output* and h stands for *header* file).

One of the most common uses of `getche()` is to select from a menu of items. For example, this program allows the user to add, subtract, multiply, or divide two numbers.

CamScanner

```
/* program 6-4.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
clrscr();
float a,b;
char ch;
printf("Do you want to:\n");
printf("add, subtract, multiply, or divide?\n\n\n");
printf("Enter first letter: ");
ch = getche();              /* read a character */
printf("\n");
printf("Enter first number: ");
scanf("%f", &a);
printf("Enter second number: ");
scanf("%f", &b);

if(ch=='a')
   printf("%.2f", a+b);
if(ch=='s')
   printf("%.2f", a-b);
if(ch=='m')
   printf("%.2f", a*b);
if(ch=='d' && b!=0)
   printf("%.2f", a/b);

getch();
}
```

Another common reason that your program will need to read a character from the keyboard is to obtain a yes/no response from the user. For example, this segment of program determines if the user wants to proceed.

```
printf("Do you wish to continue? (Y/N): ");
ch=getche();
if(ch=='Y')
{
  /* continue with something */
  . . .
  }
```

Single Character Output

The command you use to output depends on the type of the data being output and how you want it to appear. The most straightforward forms of output work with character and string data.

C contain many functions that handle single character output. Two most popular output functions are `putchar()` and `putch()`.

CamScanner

The putchar() and putch() Functions

The function putchar() stands for "put character" and displays only a single char value on the monitor. The char can be a character variable or the character itself contained in single quotes. This function is complement of input function getchar().

The argument (the information in parentheses, which you want to display) must be one of these:

1. A char constant:

```
putchar('T');
```

or

```
#define TRIANGLE 'T'
void main(void)
{
  putchar(TRIANGLE);
}
```

2. A char variable:

```
void main(void)
{
  char letter;
  letter = 'L';
  putchar(letter);
}
```

Only one character is allowed. The instruction

```
putchar('Hi');
```

would generate a compiler error.

Now consider a simple program.

```
/* program 6-5.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  int ch;
  putchar('?');          /* display a prompt */
  ch = getchar ();       /* read a keyboard char */
  putchar(ch);           /* echo to screen */

  getch();
}
```

The program is not very useful, but illustrates the simplest form of putchar(). The printf() function is much more useful in such type of output. Note that putchar() and getchar() both require the inclusion of <stdio.h>.

The following program illustrates the fact that you can use C's backslash character constants with putchar().

CamScanner

```
/* program 6-6.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  putchar('A');
  putchar('\n');    /* blank line */
  putchar('B');

  getch();
}
```

This program displays

```
A

B
```

on the screen.

Turbo C++ also include the function `putch()` which has the same effect as `putchar()`. The following program illustrates this.

```
/* program 6-7.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  char ch='P';
  putch(ch);
  putchar(ch);
  putch('K');
  putchar('K');

  getch();
}
```

And the output will be

```
PPKK
```

The limitation of `putchar()` and `putch()` is that they can output only one character at a time.

String Input

C contains a number of library functions to input text strings. The most common method is the use of `gets()` function which is smaller and faster as compared to the formatted input functions.

The `gets()` Function

The function `gets()` stands for "get string". It reads a string from the keyboard, adds the null character '\0' to it, and assigns it to the required variable, which comes

as an argument of the function. The newline character '\n' (created by pressing the enter key) signals the end of the string. Consider the following program:

```
/* program 6-8.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  char name[15];          /* assume 14 columns screen */
  gets(name);             /* read a line */
  puts(name);             /* echo */

  getch();
}
```

The `gets()` command will assign up to first 14 characters typed on the keyboard to the string variable called name and one space is reserved for the null terminator '\0'. You can enter one character less than the specified number. If you want to enter a 15-character name, for example, declare the variable with a maximum of 16 characters length, as in

```
char name[16];
```

or you can declare it explicitly as

```
char name[15+1];
```

Now look at another example:

```
/* program 6-9.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  char name[25];          /* assume 24 columns screen */
  printf("Please enter your name: ");
  gets(name);
  printf("Confirming, your name is: %s", name);

  getch();
}
```

When you run the program, you will see this prompt:

```
Please enter your name:
```

Since a `printf()` command is used, the cursor remains immediately after the prompt. It will be one space way from the colon because the space was inserted between the colon and the closing quote in the `printf()` argument. The program will wait for some sort of input—for you to type your name.

As you type your name, the characters are echoed onto the screen. If you make a mistake, before you press ↵, you can press the backspace key to delete characters

and retype them. *Turbo C++* let you press the Esc key to ignore all of your characters at one time and start over.

When you press ↵, the character you typed are assigned to the variable name and the null terminator is inserted. The second `printf()` command is then executed. The screen will look like this (if your name is Imtiaz Arif):

```
Please enter your name: Imtiaz Arif
Confirming, your name is: Imtiaz Arif
```

The `gets()` function is an excellent ways to enter strings into your program.

String Output

C contains a number of library functions to output text strings. The most common method is the use of `puts()` function which is smaller and faster as compared to the formatted input functions.

The `puts()` Function

The function `puts()` stands for "put string" and displays a string on the monitor. This function is complement of input function `gets()`.

The argument must be one of these:

1. A string constant:

```
puts("Best Wishes");
```

or

```
#define GREETING "Best Wishes"
void main(void)
{
  puts(GREETING);
}
```

2. A string variable:

```
void main(void)
{
  char greeting[30];
  puts("Enter greeting");
  gets(greeting);
  puts("Hi");
  puts(greeting);
}
```

Turbo C++ perform a new-line command after the `puts()` function. This means that after the data is displayed, the cursor will move to the start of the next line.

Formatted I/O Functions

The `scanf()` and `printf()` fall under the category of formatted input and output functions respectively. These functions instruct the compiler how to interpret the various values of input data and let us obtain the output in the specified form. Let us discuss these functions one by one.

CamScanner

The scanf() Function

The scanf() is the most versatile input function because it can handle all of the different variables and control their formatting. Think of scanf() as meaning *SCAN Formatted* characters from the keyboard. The function scans (monitors) the keyboard for keystrokes to be entered, then interprets the input based on the format specifiers. This function can be used to enter any combination of numeric, character, and string variables all at the same time. Function scanf() has the following input formatting capabilities:

1. Inputting all types of data.
2. Inputting specific characters from an *input stream*. A stream is a series of characters being input from some source. In the case of scanf(), the source is the keyboard.
3. Skipping specific characters in the input stream.

In general terms, scanf() function consists of two parts: a control string enclosed in double quotation marks followed by a comma, and then a list of variable names. The general form is

```
scanf("control string", variable list);
```

The control string contains *format specifiers* indicating how the data input will be interpreted. The variable list indicates the variables that will hold the input values.

Each format specifier starts with the percent symbol (%) followed by a letter indicating the data type. Table 6.1 summaries the scanf() format specifiers:

Table 6.1. scanf() format specifiers

Specifier	Meaning
%d	inputs an integer number
%u	inputs an unsigned integer
%f	inputs a float
%e	inputs number in scientific notation (float)
%c	inputs a character
%s	inputs a string
%o	inputs an octal number (unsigned)
%x	inputs a hexadecimal number (unsigned)

The l prefix is used before d, o, u, and x to specify type long int. Before e and f, it specifies type double. The L prefix before e and f specifies type long double.

The variable list is actually a list of variable addresses. The & (ampersand) operator is the address operator. So to input an integer or character data (but not the strings), you must use the address operator with the variable and not just the variable name itself:

```
/* program 6-10.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
```

CamScanner

```
{
  clrscr();
  float cost;
  printf("Please enter the cost of product:");
  scanf("%f", &cost);
  printf("Cost of the product is %f", cost);

  getch();
}
```

The `scanf()` instruction will input a `float` value and insert it in memory where the variable `cost` is stored. When a value is placed in the variable's address, it is automatically assigned to the variable.

Here is what happens when you run the program:

1. The `printf()` instruction displays a prompt on the screen:
   `Please enter the cost of product:`
   Because a new-line code is not included, the cursor remains at the end of the prompt.
2. The program waits for the user to enter a value.
3. You type an entry, then press ↵.
4. `scanf()` looks at the input stream and determines which characters to accept and which to ignore. If it encounters numeric characters before any non-numeric characters, it continues scanning the line up to the first non-numeric or white-space character. The numeric characters are converted to a number and stored in the address of the variable. If it encounters any non-numeric characters before a numeric character, the `scanf()` stops and the characters are ignore.
5. The cursor moves to the beginning of the next screen line.
6. The `printf()` instruction displays the value of the variable `cost` following the message *Cost of the product is.*

`scanf()` automatically ignores all whitespace characters—spaces, tabs, or the new-line command—except when the next data type is a `char`. Consider this segment of program:

```
int count;
puts("Please enter a number: ");
scanf("%d", &count);
print("The number is %d", count);
```

You can press the spacebar as many times as you want before typing the number. C ignores the spaces and looks for first non-whitespace character. It then tries to fit the characters into the format expected by the specifier. It the characters match—in this case if they are numbers—`scanf()` assigns the numbers to the variable. The assignment stops at the first non-matching, non-numeric character. So if you type *123ABC*, the value *123* will be assigned to the variable and the letters *ABC* will be ignored, as shown in Fig. 6.1. The assignment will also stop at the first whitespace character—if you enter *12 3*, the value *12* will be assigned to the variable and the number *3* ignored.

CamScanner

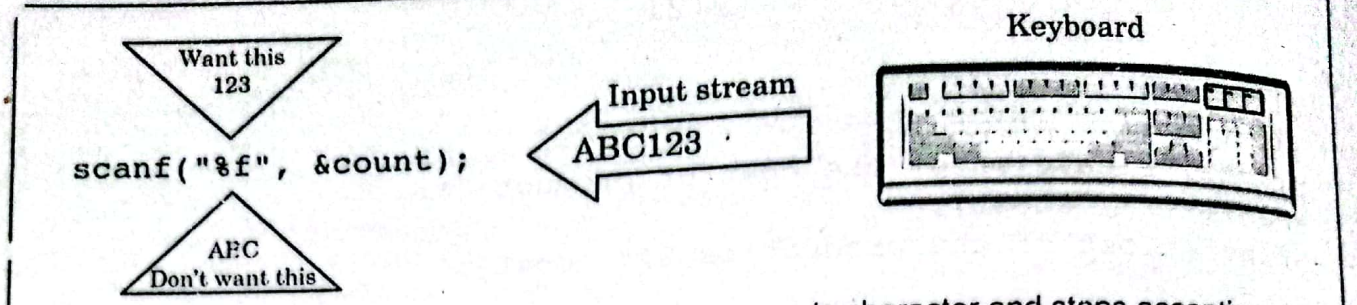


Fig. 6.1. scanf() starts accepting data at the first numeric character and stops accepting data at the first non-numeric character

The first non-whitespace character in the input stream must match the format specifier. If you entered *ABC123*, the entire entry would be ignored and you would have no idea what value the variable will hold.

The characters considered a "match" depend on the specifier. If the specifier is `%d`, for example, only the number characters and the hyphen (minus sign) are acceptable. If the specifier is `%x`, the characters *0123456789ABCDEF* are acceptable because they are used as hexadecimal numbers. If the specifier is `%c`, any character is allowed, even a space. Because a `char` type can be any character, `scanf()` does not ignore whitespace characters in the input stream. If the instruction is

```
char letter;
scanf("%c", &letter);
```

and you press the spacebar before typing the character, `scanf()` would assign the space to the variable, ignoring the character that followed. You cannot type spaces before entering a `char` value.

When you enter a string, `scanf()` begins the assignment at the first non-whitespace character. It stops assigning input to the variable at the next whitespace character. Look at this program:

```
/* program 6-11.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  char name[25];
  clrscr();
  puts("Please enter your name: ");
  scanf("%s", name);
  puts(name);

  getch();
}
```

Notice that the ampersand (&) is not used with a string variable name. If you enter *Iqbal* and press ↵, the characters are assigned to the variable name. But if you typed *Iqbal Hussain* (as illustrated in Figure 6.2), `scanf()` would begin assigning characters at the first non-whitespace character and stop at the next whitespace

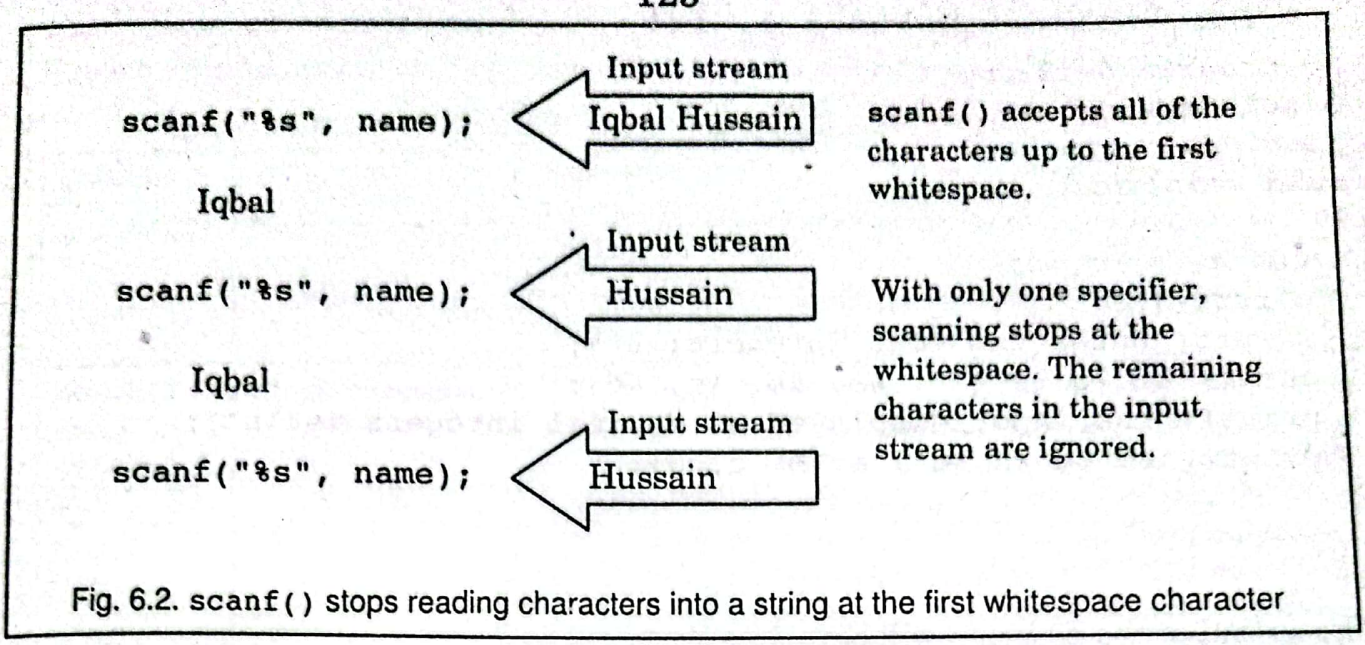


Fig. 6.2. scanf() stops reading characters into a string at the first whitespace character

character. Only the first name *Iqbal* is assigned, and the last name *Hussain* is ignored. Because of the ways `scanf()` operates, it is not recommended for entering strings and if you want to enter a string particularly that contains a space, you must use the `gets()` function.

You can use as many `scanf()` function as needed to input your data. Program 6-12, for example, show a program that inputs four variables—an `int`, a `float`, a `char`, and a string. Notice that the address operator is used for all the variables except the string.

```
/* program 6-12.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  int num;
  float fnum;
  char letter, name[15];
  clrscr();
  printf("Type \n\tinteger, \n\tfloat, \n\tcharacter and \n\t
  string\n");
  scanf("%d %f %c %s", &num, &fnum, &letter, name);

  printf("\n\tInteger   =       %d", num);
  printf("\n\tFloat     =       %.2f", fnum);
  printf("\n\tLetter    =       %c", letter);
  printf("\n\tName      =       %s", name);

  getch();
}
```

Program 6.13 reads integers with the various integer conversion specifiers an displays the integers as decimal numbers.

CamScanner

```
/* program 6-13.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  int a, b, c, d;
  clrscr();
  printf("Enter the four integers:\n");
  scanf("%d %o %u %x", &a, &b, &c, &d);
  printf("The input displayed as decimal integers is:\n");
  printf("%d %d %d %d", a, b, c, d);

  getch();
}
```

The output of the program will be:

```
Enter the four integers:
-14 14 14 14
The input displayed as decimal integers is:
-14 12 14 20
```

A field width can be used in a `scanf()` conversion specification to read a specific number of characters from the input stream. Program 6-14 inputs a series of consecutive digits as a two-digit integer and an integer consisting of the remaining digits in the input stream.

```
/* program 6-14.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  int a, b;
  clrscr();
  printf("Enter a six digit integer: ");
  scanf("%2d %d", &a, &b);
  printf("The integers input were %d and %d\n", a, b);

  getch();
}
```

The output of the program will be:

```
Enter a six digit integer: 456789
The integers input were 45 and 6789
```

Caution: On many systems, characters being typed into the input stream are stored in an area of memory called the *buffer*. If `scanf()` terminates early, some characters will remain in the buffer instead of being assigned to variables. It is possible that the next input operation could start reading the characters still in the buffer from the last attempted input, rather than those you type in direct response to the next input.

Use `scanf()` with care. Input one piece of data at a time and include prompts thoroughly explaining the type of input desired.

CamScanner

Table 6.2 summarises the input commands you have learned in this chapter.

Table 6.2. Some Input Commands

Function	Data Type	Comments
`getchar()`	Single characters	Use to input a character declared as either an `int` or `char` type. You have to press ↵ after typing the character. Use without an argument, as a variable, for example, `letter = getchar()`.
`getch()` and `getche()`	Single characters	Use to input a character declared as either an `int` or `char` type. You do not have to press ↵ after typing the character. Use without an argument, as a variable, for example, `letter = getch()` or `letter = getche()`, or use by itself to pause a program. The difference between two functions is that the function `getche()` echoes the character that you typed to the screen whereas `getch()` just returns the character that you typed without echoing it on the screen.
`gets()`	String type only	Use for inputting strings, including those with spaces. You must press ↵ after typing the string.
`scanf()`	All data types	Needs format specifiers for each data item to input. Do not use to input strings that contain spaces. You must be careful to insure that the data is input and assigned correctly. `scanf()` can be used to input multiple arguments in one instruction. Do not use a space before a `char` variable.

The `printf()` Function

The `puts()` and `putchar()` functions are quite useful but they have several limitations. Neither can output numeric data, and each can only display one argument. Both `puts()` and `putchar()` can only display one thing.

The `rintf()` is a more versatile function because it can display data of all types and can work with multiple arguments. This function can be used to output any combination of numerical values, single characters and strings. In addition, `printf()` can format how the data appears. It is similar to the input function `scanf()`, except that its purpose is to display data rather than to enter it into the computer. Function `printf()` can perform the following formatting capabilities:

1. *Rounding* floating-point values to an indicated number of decimal places.
2. *Aligning* a column of numbers with decimal points appearing one above the other.
3. *Right-justification* and *left-justification* of outputs.
4. *Inserting characters* (`char`) at precise locations in a line of output.
5. Representing floating-point numbers in exponential format.
6. Representing unsigned integers in octal and hexadecimal format.
7. Displaying all types of data with fixed-size field widths and precisions.

CamScanner

Like `scanf()` function, the `printf()` function consists of two parts: a control string enclosed in double quotation marks followed by a comma, and then a list of variable names. The general form is

```
printf("control string", variable list);
```

The control string contains *format specifiers* indicating how the data output will be interpreted. The variable list indicates the variables that will hold the output values.

Each format specifier starts with the percent symbol (%) followed by a letter indicating the data type. Table 6.3 summaries the `printf()` format specifiers:

Table 6.3. printf() format specifiers

Specifier	Meaning
`%d`	displays an integer number
`%u`	displays an unsigned integer
`%f`	displays a `float`
`%e`	displays number in scientific notation (`float`)
`%c`	displays a character
`%s`	displays a string
`%o`	displays an octal number (`unsigned`)
`%x`	displays a hexadecimal number (`unsigned`)

The `l` prefix is used before `d`, `o`, `u`, and `x` to specify type `long int`. Before `e` and `f`, it specifies type `double`. The `L` prefix before `e` and `f` specifies type `long double`.

A complete, but very simple, `printf()` instruction looks like this:

```
printf("%d", 17);
```

When these instructions are performed, the value 17 will be inserted at the position of the placeholder, the `%d` format specifier (see Fig. 6.3).

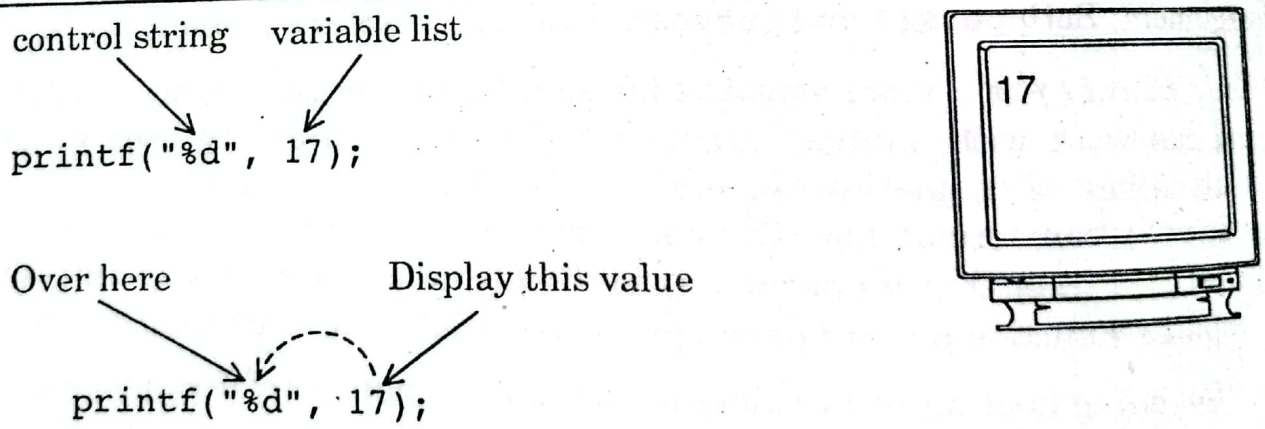


Fig. 6.3. The value replaces the format specifier

The control string can also include text along with the format specifiers displaying data. As an example, look at this line:

```
printf("My age is %d years", 17);
```

The control string here is

```
"My age is %d years"
```

The format specifier, `%d`, shows that you want to display a number between

words *My age is* and *years* (see Fig. 6.4). The compiler substitutes the number 17 for the format specifier to output:

```
My age is 17 years
```

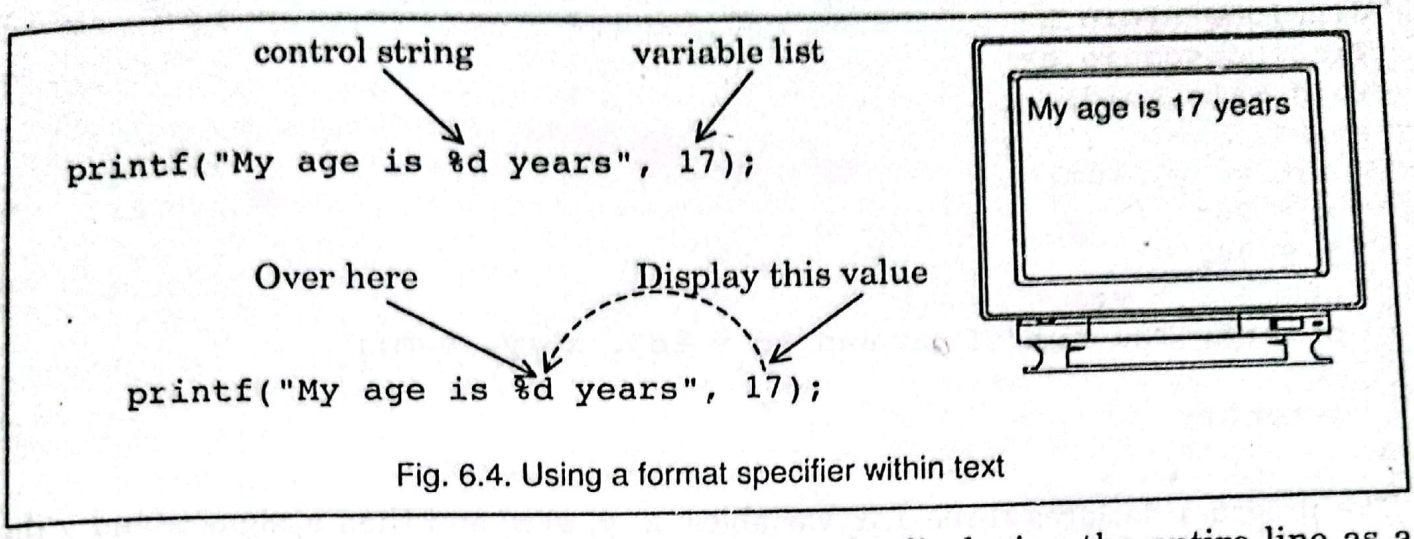


Fig. 6.4. Using a format specifier within text

Of course, you could have achieved the same result, displaying the entire line as a string, with either of these instruction:

```
printf("My age is 17 years");
puts("My age is 17 years");
```

But to combine text with a numeric constant or variable, you have to use `printf()` and a format specifier, as in this program:

```
/* program 6-15.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  int age;
  age = 17;
  printf("My age is %d years", age);

  getch();
}
```

Here, the program displays a string and an `int` variable in one instruction (as shown in Fig. 6.5).

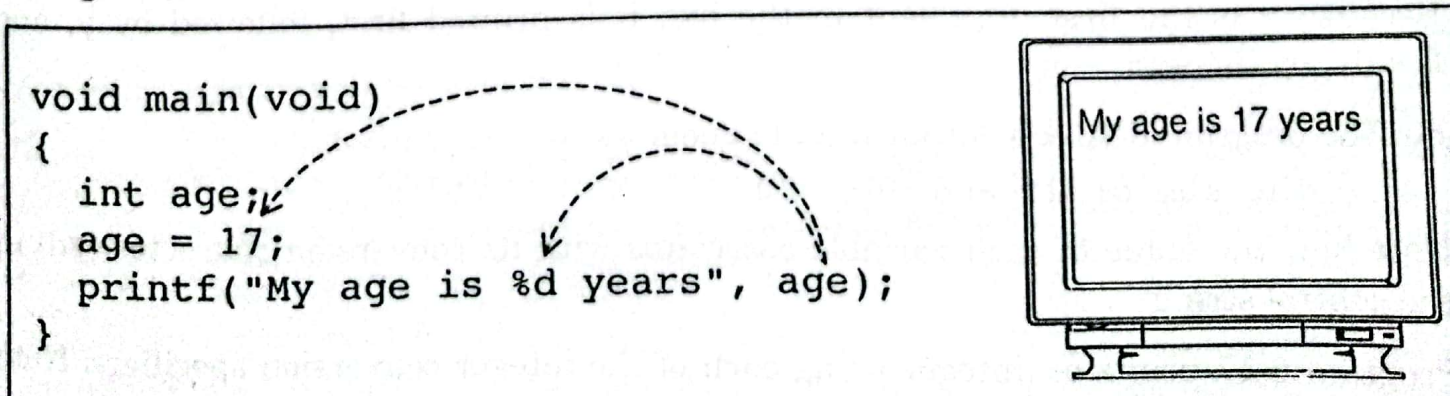


Fig. 6.5. The value of a variable replaces the format specifier

CamScanner

You can display multiple arguments, but you must include a format specifier for each argument. The values in the data list must appear in the same order as their specifiers. The first item in the list is substituted for the first format specifier, the second item for the second specifier, and so on. Look at this program:

```
/* program 6-16.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  int x, y, sum;
  x = 20;
  y = 30;
  sum = x + y;
  printf("The sum of %d and %d = %d", x, y, sum);

  getch();
}
```

The program declares three `int` variables, `x`, `y`, `sum`, and then assigns `x` and `y` the initial values. The data list in the `printf()` instruction contains three variables (arguments) that must be displayed, so the control string must have three format specifiers. Since all the three variables are `int` type, the three format specifiers must be `%d`, as shown in Fig. 6.6.

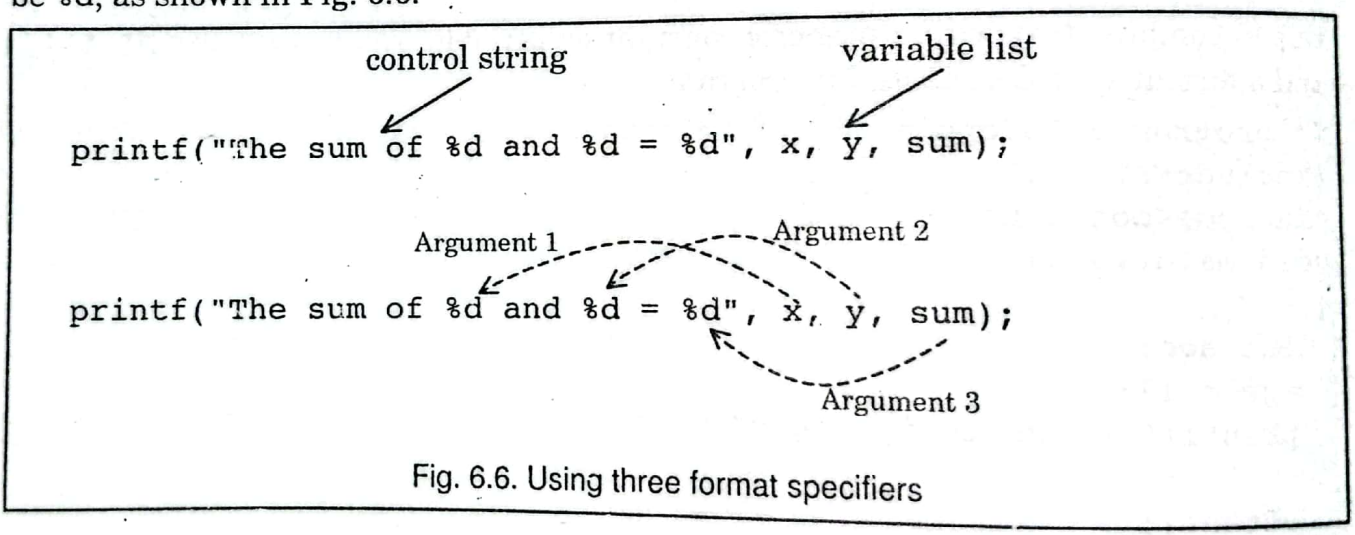


Fig. 6.6. Using three format specifiers

The *control string* appears between quotation marks and contains three format specifiers indicated by the three `%d`'s in the string. These `%d`'s signal that three arguments will follow the end of the control string, each separated by a comma. Because `x` is the first argument in the list, it is printed first, followed by `y`, and finally, by the `sum`.

For the program 6-16 the output would appear as

```
The sum of 20 and 30 = 50
```

Note how the value of each variable associates with its conversion character (`%d`) in the control string.

Program 6-17 prints an integer using each of the integer conversion specifiers. Note

CamScanner

that only the minus sign prints; plus signs are suppressed. Also note that the value -455 is read by %u and converted to unsigned value 65081.

```
/* program 6-17.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  printf("%d\n", 455);
  printf("%3d\n", 455);
  printf("%d\n", +455);
  printf("%d\n", -455);
  printf("%10d\n", 455);
  printf("%-8d\n", 455);
  printf("%ld\n", 2000000000);
  printf("%o\n", 455);
  printf("%u\n", 455);
  printf("%u\n", -455);
  printf("%x\n", 455);
  printf("%x\n", 455);

  getch();
}
```

The output of the program will be:

```
455
455
455
-455
        455
455
2000000000
707
455
65081
1c7
1c7
```

Function printf() also provides the ability to specify the *precision* with which data is printed. The default conversion for %f is rounded to six decimal places. You can vary this precision using a decimal point and a precision specifier. To specify the number of decimal places, use the format

```
%.nf
```

where n is the number of places desired (see Fig. 6.7). For example,

```
float height = 2500.35;
printf("The height is %.2f", height);
```

will display the float number with only two decimal places:

```
The height is 2500.35
```

CamScanner

Similarly, the instruction

```
printf("The height is %.3f", height);
```

will display the height with three decimal places:

```
The height is 2500.350
```

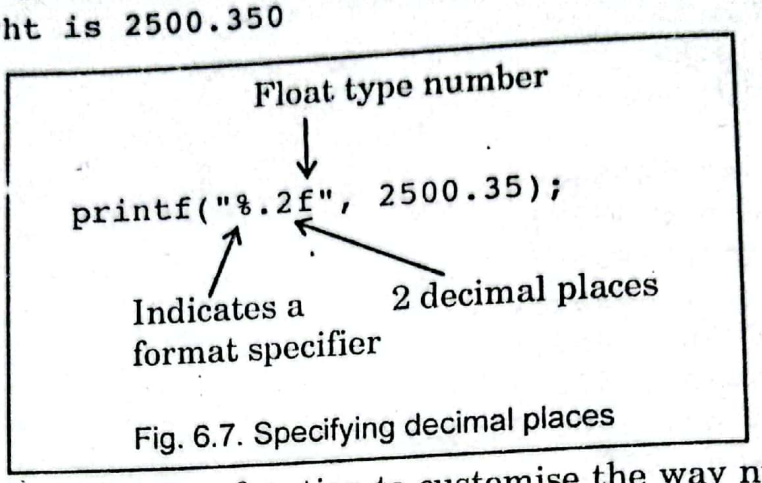


Fig. 6.7. Specifying decimal places

You can also use the `printf()` function to customise the way numbers and text are displayed. You control the spacing and number of characters displayed by using *field-width specifiers* as

```
%N.nf
```

where `N` is the total width of the field. For example,

```
printf("The height is %10.2f", height);
```

will display

```
The height is    2500.35
```

with three extra spaces before the number. Some other examples are shown in table 6.4 (*s* stands for single space).

Table 6.4. Some other Examples of Precision Specifiers

Example	Displays height As ...
`%.2f`	2500.35 (no width specified)
`%6.2f`	2500.35 (width ignore—too small)
`%9.2f`	*ss*2500.35 (pad blanks to 9 columns)
`%09.2f`	002500.35 (pad zeroes to 9 columns)
`%14f`	*sss*2500.350098 (same as `%14.6f`)
`%14.0f`	*sssssssss*2500 (note no decimal point)
`%#14.0f`	*sssssssss*2500. (unless you add a #)
`%-6.2f`	2500.35 (width ignore—too small)
`%-9.2f`	2500.35*ss* (pad right blanks to 9 columns)
`%-09.2f`	2500.35*ss* (same! The zero is ignored)
`%-14f`	2500.350098*sss* (same as `%-14.6f`)
`%-14.0f`	2500*sssssssss* (note no decimal point)
`%-#14.0f`	2500.*sssssssss* (unless you add a #)

CamScanner

Some of these printf() variations are listed in program 6-18. The entry %%f is needed to display %f—it is not a format specifier (printf statement uses %% to print % character in a character string).

```
/* program 6-18.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  float height = 2500.35;
  double depth = 3.12e5;
  clrscr();
  printf("%%f          height is %f\n",       height);
  printf("%%.2f        height is %.2f\n",     height);
  printf("%%9.2f       height is %9.2f\n",    height);
  printf("%%-9.2f      height is %-9.2f\n",   height);
  printf("%%09.2f      height is %09.2f\n",   height);
  printf("%%14.0f      height is %14.0f\n",   height);
  printf("%%-14.0f     height is %-14.0f\n",  height);
  printf("%%#14.0f     height is %#14.0f\n",  height);
  printf("%%e          height is %e\n",       height);
  printf("%%.3e        height is %.3e\n",     height);
  printf("%%f          depth  is %f\n",       depth);
  printf("%%E          depth  is %E\n",       depth);

  getch();
}
```

The output of the program will be:

```
  %f          height  is 2500.350098
  %.2f        height  is 2500.35
  %9.2f       height  is   2500.35
  %-9.2f      height  is 2500.35
  %09.2f      height  is 002500.35
  %14.0f      height  is           2500
  %-14.0f     height  is 2500
  %#14.0f     height  is          2500.
  %e          height  is 2.500350e+03
  %.3e        height  is 2.500e+03
  %f          depth   is 312000.000000
  %E          depth   is 3.120000E+05
```

The %c and %s format specifiers are used to print individual characters and strings, respectively. Format specifier %c requires a char argument whereas format specifier %s causes characters to be printed until a terminating null ('\0') character is encountered. The program 6-19 displays characters and strings with format specifiers %c and %s.

```
/* program 6-19.cpp */
#include<stdio.h>
```

CamScanner

```
#include<conio.h>
void main(void)
{
  char character = 'A';
  char string[] = "This is a string";
  clrscr();
  printf("%c\n", character);
  printf("%s\n", string);
  printf("%s\n", "This is also a string");

  getch();
}
```

And the output will be

```
A
This is a string
This is also a string
```

Field-width specifiers work with string and character data as well. Extra spaces will appear in front of the text, moving the string to the right side of the imaginary box. For example, if the string variable name `message` has the value "Hello", the command:

```
printf("I just called to say %7s there", message);
```

will display

```
I just called to say   Hello there
```

The two extra spaces are inserted before the string value.

Table 6.5 summarises the output commands you have learned in this chapter.

Table 6.5. Some Output Commands

Function	Data Type	Comments
`putchar()` and `putch()`	Single characters	Use to output a character declared as either an `int` or `char` type. Does not automatically move the cursor to the next line after displaying. Use it to display a character in single quotes, a `char` constant or variable, or a special character.
`puts()`	String type only	Use for outputting strings only. Outputs strings in quotes, variables, and constants. Automatically move the cursor to the next line after displaying.
`printf()`	All data types	Include a format specifier for each data item to be output. Can be used to output multiple arguments in one instruction. Does not automatically move the cursor to the next line—use `\n` code.

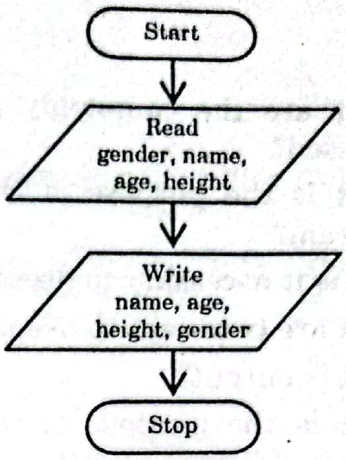
CamScanner

Algorithm

Step 1. Read gender.
Step 2. Read name.
Step 3. Read age.
Step 4. Read height.

Step 5. Write name.
Step 6. Write age.
Step 7. Write height.
Step 8. Write gender.
Step 9. Exit.

Flowchart



Source Code

```
/* Practical 2: Writing a Program that reads and prints the data
using escape sequence (asking the name, age, height and gender of
student using scanf and printf statement). */
#include<conio.h>
#include<stdio.h>
void main(void)
{
  char gender, name[25];
  int age;
  float height;

  clrscr();
  printf("\nEnter Your Gender(M/F): ");
  scanf("%c", &gender);
  printf("\nEnter Your Name: ");
  scanf("%s", name);
  printf("\nEnter Your Age (in years): ");
  scanf("%d", &age);
  printf("\nEnter Your Height (in ft.): ");
  scanf("%f", &height);
  clrscr();
  printf("\n\tName:          %s", name);
  printf("\n\tAge:           %d", age);
  printf("\n\tHeight:        %0.2f", height);
  printf("\n\tGender:        %c", gender);

  getch();
}
```

Output

```
Name:          Ikram_Azam
Age:           45
Height:        5.75
Gender:        M
```

CamScanner

Exercise

6.1. What are the commonly used input/output functions in C? How are they accessed?

6.2. What is the purpose of the `getchar` function? How is it used within a C program?

6.3. Why is it necessary to press ↵ after a `getchar()` input?

6.4. What are two ways to use `getchar()`?

6.5. What is output?

6.6. What is the purpose of the `putchar` function? How is it used within a C program? Compare with the `getchar` function.

6.7. What types of arguments can be used in a `puts()` function?

6.8. Write the `puts()` instructions for printing your name and address.

6.9. Write the `puts()` instruction that centers the word Title on the screen. The screen is 80 characters wide.

6.10. What types of arguments can be used in a `putchar()` function?

6.11. What is a character-type array? What does each element of a character-type array represent? How are character-type arrays used to represent multi-character strings.

6.12. How would you display the quote symbol on the screen?

6.13. What is the purpose of the `scanf` function? How is it used within a C program? Compare it with the `getchar` function.

6.14. What are the advantages of using `scanf()` for input?

6.15. When would you use `gets()` rather than `scanf()` to input a string?

6.16. What are the advantages and disadvantages of formatted `scanf()` input?

6.17. What is the purpose of the control string in a `scanf` function? What type of information does it convey? Of what is the control string composed?

6.18. If a control string within a `scanf` function contains multiple character groups, how are the character groups separated? Are whitespace characters required?

6.19. What is the address operator?

6.20. What special symbol must be included with the arguments, other than the control string, in a `scanf` function? In what way are array names treated differently than other arguments?

6.21. When entering data via the `scanf` function, what relationships must there be between the data items and the corresponding arguments? How are multiple data items separated from one another?

6.22. What happens if an input data item contains more characters than the maximum allowable field width? What if the data item contains fewer characters?

6.23. How can the minimum field width for a data item be specified within the `printf` function.

CamScanner

6.24. Is it necessary to assign initial values to all variables?

6.25. What is the purpose of the `printf` function? How is it used within a C program? Compare with the `putchar` function.

6.26. What are the two parts of the `printf()` instruction?

6.27. What are the advantages of `printf()` over `puts()`?

6.28. In what ways does the control string within a `printf` function differ from the control string within a `scanf` function?

6.29. If the control string within a `printf` function contains multiple character groups, how are the character groups separated? How are the separators interpreted?

6.30. Summarise the meaning of the more commonly used conversion characters within the control string of a `printf` function. Compare with the conversion characters that are used in a `scanf` function.

6.31. Write the `printf()` instructions for printing your name and address.

6.32. Write the `printf()` instruction that outputs the values of the following variables:

```
float length, width, height, volume;
```

6.33. Differentiate the following:

(a) `scanf()` and `printf()`

(b) `getchar()` and `putchar()`

(c) `gets()` and `puts()`

6.34. A program will display a person's name and age. Write a `printf()` instruction that displays the values of the variables:

```
char name[12];
int age;
```

6.35. A program has these variables:

```
char item[] = "Floppy disk";
float cost = 25.50;
float sale_tax = 0.15;
```

Write the `printf()` instructions that display:

```
Item Name:      Floppy disk
Item Cost:      25.50
Sale tax:       0.15
```

Note the alignment of the values.

6.36. Write a program that sounds a bell.

6.37. A C program contains the following statements:

```
#include <stdio.h>
char a, b, c;
```

(a) Write appropriate `getchar` statements that will allow values for `a`, `b` and `c` to be entered into the computer.

(b) Write appropriate `putchar` statements that will allow the current values of `a`, `b` and `c` to be displayed on monitor.

CamScanner

6.38. A C program contains the following statements:

```
#include <stdio.h>
int i, j, k;
```

Write an appropriate scanf function to enter numerical values for i, j and k assuming

(a) The values for i, j and k will be decimal integers.

(b) The values for i, j and k will be octal integers.

(c) The value for i will be a decimal integer, j an octal integer and k a hexadecimal integer.

(d) The values for i and j will be hexadecimal integers and k will be an octal integer.

6.39. Show the output displayed by the following program lines when the data entered are 5 and 7:

```
/* program 6-21.cpp */
#include<conio.h>
#include<stdio.h>
void main(void)
{
  int num1, num2;
  clrscr();
  printf("Enter two integers: ");
  scanf("%d %d", &num1, &num2);
  num1 = num1 + 5;
  num2 = 3 * num2;
  printf("\tnum1 = %d\n\tnum2 = %d\n", num1, num2);

  getch();
}
```

By: Sir Ahad

6.40. Interpret the meaning of the control string in each of the following scanf functions.

(a) `scanf("%12ld %15lf %15le, &a, &b, &c);`

(b) `scanf("%10lx %6o %14lu, &a, &b, &c);`

(c) `scanf("%ld %lf %15e, &a, &b, &c);`

(d) `scanf("%2d %2f %15lf, &a, &b, &c);`

6.41. A C program contains the following variable declarations.

```
int i = 12345, j = 0xabcd9, k = 077777;
```

Show the output resulting from each of the following printf statements.

(a) `printf("%d %x %o", i, j, k);`

(b) `printf("%3d %3x %3o", i, j, k);`

(c) `printf("%8d %8x %8o", i, j, k);`

(d) `printf("%-8d %-8x %-8o", i, j, k);`

(e) `printf("%+8d %+8x %+8o", i, j, k);`

(f) `printf("%08d %#8x %#8o", i, j, k);`

CamScanner

6.42. Write a program which prints two groups of five numbers each with right justification as,

```
    1
   12
  123
 1234
12345
   -1
  -12
 -123
-1234
-12345
```

Ans.

```
/* program 6-22.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  printf("%6d\n", 1);
  printf("%6d\n", 12);
  printf("%6d\n", 123);
  printf("%6d\n", 1234);
  printf("%6d\n", 12345);
  printf("%6d\n", -1);
  printf("%6d\n", -12);
  printf("%6d\n", -123);
  printf("%6d\n", -1234);
  printf("%6d\n", -12345);

  getch();
}
```

6.43. What is a format specifier?

6.44. How do you display the value of a numeric variable?

6.45. What is meant by the precision of an output data item? To what types of data does this apply?

6.46. How can the precision be specified within a `printf` function?

6.47. Write a program that inputs three numbers, then displays the numbers in reverse order—the last number input is displayed first, etc.

6.48. Write a program that uses `getchar()`, `gets()`, and `scanf()`.

6.49. Write a program that asks the user to enter the radius of a circle and then computes and displays the circle's area. Use the formula

$$area = PI \times radius \times radius$$

where *PI* is the constant whose value is `3.14159`.

6.50. Write a program to test the results of printing the integer value `12345` and the

floating-point value `1.2345` in various size fields. What happens when the values are printed in fields containing fewer digits than the values?

6.51. Write a program that prints the value `100.453627` rounded to the nearest digit, tenth, hundredth, thousandth and ten thousandth.

6.52. Write a program that converts Fahrenheit temperatures to Celsius temperatures with 3 digits of precision. Use the formula

```
Celsius = 5.0/9.0 * (Fahrenheit - 32);
```

to perform the calculation.

6.53. Write a program that determines whether `?` can be printed as part of a `printf` control string as a character rather than using the `\?` escape sequence.

6.54. Write a program that inputs the value `437` using each of the `scanf` integer format specifiers. Print each input value using all the integer format specifiers.

6.55. Write a program that uses each of the conversion specifiers `e` and `f` to input the value `1.2345`. Print the values of each variable to prove that each format specifier can be used to input this same value.

CamScanner

Objectives

6.1. Precise input formatting is accomplished with the `scanf` function.

6.2. The format specifier `le, lE or lf` can be used to input a `double` value.

6.3. Precise output formatting is accomplished with the `printf` function. The control string may contain format specifiers, field width, precision and escape sequences.

6.4. The format specifier `%d` may be used to output a singed decimal integer.

6.5. The format specifiers `%o`, `%u` and `%x` are used to display unsigned integers in octal, decimal and hexadecimal form, respectively.

6.6. The modifier `l` is placed before the integer format specifier to indicate the `long` integer value to be displayed.

6.7. The conversion specifier `e or E` is used to display a floating-point value in exponential notation.

6.8. The modifier `L` is place before any floating-point format specifier to indicate that a `long double` value is to be displayed.

6.9. The format specifiers `e`, `E` and `f` are displayed with six digits of precision to the right of the decimal point if no precision is specified.

6.10. The format specifiers `%s` and `%c` are used to print strings and characters, respectively.

6.11. The function `putchar()` stands for put character.

6.12. The function `gets()` stands for get string.

6.13. *Turbo C++* perform a new-line command after the `puts()` function.

6.14. All stings end in the `NULL ('\0')` character.

6.15. Find the error in each of the following and explain how the error can be corrected.

(a) The following statement should print the character `'C'`.

```
printf("%s\n", 'C');
```

Ans. Error: Format specifier `%s` expects an argument of type string.

Correction: To print the character `'c'`, use the format specifier `%c` or change `'c'` to `"c"`.

(b) The following statement should print `9.375%`.

```
printf("%.3f%", 9.375);
```

Ans. Error: Trying to print the character `%` without using the forma specification `%%`.

Correction: Use `%%` to print a character `%`.

(c) The following statement should print the first character of the strin `"Monday"`.

```
printf("%c\n", "Monday");
```

CamScanner

Ans. Error: Format specifier c expects an argument of type char.

Correction: To print the first character of "Monday" use the format specification %1s.

(d) `printf(""A string in quotes"");`

Ans. Error: Trying to print the character " without using the \" escape sequence.

Correction: Replace each quote in the inner set of quotes with \".

(e) `printf(%d%d, 12, 20);`

Ans. Error: The control string is not enclosed in double quotes.

Correction: Enclose %d%d in double quotes.

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

Ans. Error: The character x is enclosed in double quotes.

Correction: Character constants to be printed with %c must be enclosed in single quotes.

(g) `printf("%s\n", 'Abbas');`

Ans. Error: The string to be printed is enclosed in single quotes.

Correction: Use double quotes instead of single quotes to represent a string.

6.16. Write a statement for each of the following:

(a) Print 1234 right-justified in a 10 digit field.

Ans. `printf("%10d\n", 1234);`

(b) Print 123.456789 in exponential notation with a sign (+ or −) and 3 digits of precision.

Ans. `printf("%+.3e\n", 123.456789);`

(c) Read a double value into variable number.

Ans. `scanf("%lf", &number);`

(d) Print 100 in octal form.

Ans. `printf("%o\n", 100);`

(e) Read a string into character array string.

Ans. `scanf("%s", string);`

(f) Print 3.333333 as a long double value with a sign (+ or −) in a field of 20 characters with a precision of 3.

Ans. `printf("%+20.3Lf\n", 3.333333);`

CamScanner

CHAPTER 7

SELECTION CONTROL STRUCTURES

By: Sir Ahad

In the early days of programming, as programmers began writing longer and more sophisticated software, they quickly ran into difficulties. Their programs consisted of long lists of instructions, one after the other. It was hard to determine where one task stopped and another started. As a result, the logic of the programs was difficult to follow and making modifications at a later time was extremely difficult. In the mid-1960s, the idea of structure programming was conceived to solve these problems, and it is now the standard for programming. *Structured programming* is a standardised approach that creates a program as a series of interrelated sub-programs, or functions. It also makes programs easier to write, check, read, and modify. Here we will examine the basic concepts of structure in C language.

In a program, *control structures* control the flow of execution in a program or function. Structured programming uses a limited number of control structures to minimise the complexity of programs and thus cut down on errors. There are three basic control structures in structured programming:

1. Sequence, or sequential flow
2. Selection, or conditional flow
3. Iteration, or repetitive flow

The *sequence control structure* is the most straightforward: the statements are executed in the same order in which they appear within the program. The programs written in the previous chapters are all sequential programs. We say that such a program is a sequence of steps, meaning that

CamScanner

1. At a time only one step is to be carried out.
2. Every step is to be carried out once and only once; none is repeated and none is omitted.
3. The order of executing the steps is the same as that in which they are written.
4. Termination of the last step indicates the end of the program.

The Fig. 7.1 shows the general format of a sequence control structure using flowchart.

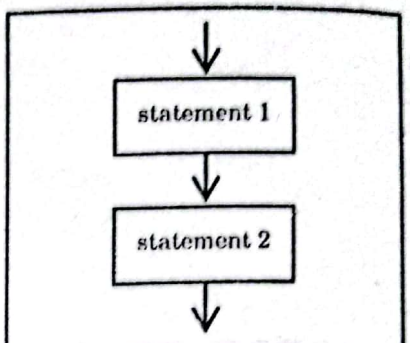


Fig. 7.1. Sequence control structure flowchart

This chapter describes the C control structures for *selection* and chapter 8 covers the control structures for *iteration* or *repetition*.

The *selection control structure* employs a number of conditions which lead to a selection of one out of several alternatives. C provides three types of selection structures. The `if` selection structure either performs (selects) an action if a condition is true or skips the action if the condition is false. The `if/else` selection structure performs an action if a condition is true and performs a different action if the condition is false. The `switch` selection structure performs one of many different actions depending on the value of an expression.

C provides three types of repetition structures, namely the `while`, and the `do/while` and the `for` (discussed in chapter 8).

That is all there is. C has only seven control structures: Sequence, three types of selection and three types of repetition. Each C program is formed by combining as many of each type of control structure as is appropriate for the program.

The `if` Selection Structure

The `if` statement tests a particular condition. Whenever that condition evaluates as *true*, an action is performed—but if it is not true, then the action is skipped and control transfers to the next step of the program.

The general structure of the command is:

```
if(condition)
   statement;
```

The command says "if this condition is true, then perform the following statement" (see Fig. 7.2). The computer performs the statement, then continues normally, performing the statements that follow the `if` command.

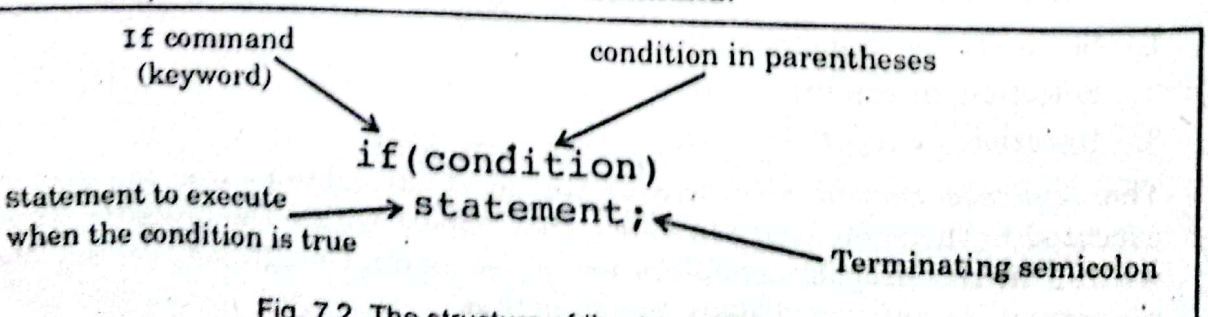


Fig. 7.2. The structure of the `if` command

CamScanner

If the condition is false, however, the computer skips the statement that is part of the `if` command, and moves on to the next statement in the program. Writing an `if` command is easy as long as you remember these basic points:

1. Place the condition in parentheses.
2. There is no semicolon after the condition—it appears after the statement.
3. While we paraphrase the command by saying "If condition then do statement," there is no *then* command in C.
4. C is a free-form language, so the condition and statement can be on the same line. Separating and indenting the condition, however, make a program easier to read.

The `if` statement can be visualised using the flowchart, as shown in Fig. 7.3.

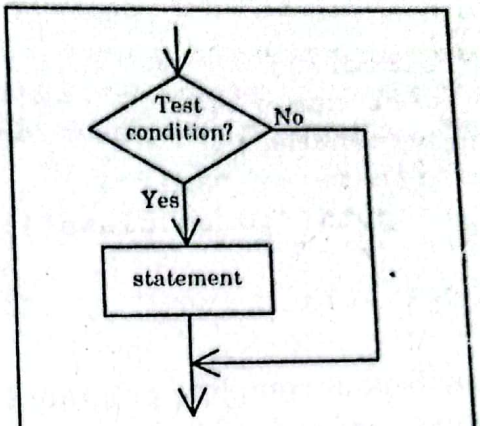


Fig. 7.3. `if` statement flowchart

Conditions in `if` commands compare values—a variable or constant is compared with another variable or constant. The comparison is done with one of these relational operators:

Operator	Meaning	Example
==	equal	`if(salary == 10000)`
<	less than	`if(hours < 40)`
>	greater than	`if(hours > 40)`
<=	less than or equal to	`if(cost <= limit)`
>=	greater than or equal to	`if(cost >= limit)`
!=	not equal to	`if(count != 1)`

Note that when testing whether two values are the same, you must use the == operator, two equal signs in a row. If you use a single equal sign, your compiler will generate an error. The single = symbol is reserved for assigning value to variables.

For example, to print the word `Overtime!` when hours are greater than 40, we would write

```
if(hours > 40)
   printf("Overtime!\n");
```

If the variable `hours` were 40 or less, the word `Overtime!` would not be displayed.

Similarly, in the example

```
if(percentage_marks >= 80)
   printf("Congratulations!\n");
printf("Your grade is %c\n", grade);
```

a congratulatory message is printed only when the value of `percentage_marks` is greater than or equal to 80. The second `printf()` is always executed.

You can use `if` conditions to test numeric and `char` variables, but not strings. The following program, for example, would compile without a problem but it would not display the correct result:

CamScanner

```
/* program 7-1.cpp */
#include<conio.h>
#include<stdio.h>
void main(void)
{
  clrscr();
  char name[10];
  gets(name);
  if(name == "amir")
     puts("go to class");

  getch();
}
```

Now look at complete program that uses an `if` command. In this program, a special luxury tax is added to items over Rs. 40,000.

```
/* program 7-2.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  float  cost, tax, luxury, total;
  luxury = 0.0;
  printf("Enter the cost of the item: ");
  scanf("%f", &cost);
  tax = cost * 0.06;
  if (cost > 40000.00)
     luxury = cost * 0.005;
  total = cost + tax + luxury;
  printf("The total cost is %.2f", total);

  getch();
}
```

The calculations is performed in the command:

```
if (cost > 40000.00)
   luxury = cost * 0.005;
```

The command says "If the value of `cost` is over Rs. 40,000, then assign the variable name `luxury` the value of `cost` times 0.5 percent". The last two instructions in the program are always performed, whether there is a luxury tax or not, because they come after the `if` command's terminating semicolon.

Notice that the variable `luxury` is assigned an initial value of 0.0 earlier in the program, but not the variable `tax`. The variable `tax` will always be assigned a value in the calculation `tax = cost * 0.06`, but `luxury` is only assigned a calculated value if the cost is over Rs. 40,000. If the condition is false, the calculation would not be performed and, if you did not assign an initial value, some unknown value would be

CamScanner

added to the total. So as a general rule, it is wise to assign initial values to variables calculated in if statements, as well as to counters and accumulators.

Multiple Statements

The basic if command performs one statement. If you want to perform more than one statement when the condition is true, you must use a second level of braces. The braces mark the beginning and end of a block of statements performed when the condition is true:

```
if(condition)
  {
    statement 1;
    statement 2;
    statement 3;
  }
```

In the following example two if statements are used:

```
if(j < k)
   minutes = j;
if(j < k)
   printf("j is smaller than k\n");
```

The more efficient and more understandable segment of the program can be written by using a single if statement with a compound statement for its body.

```
if(j < k)
  {
    minutes = j;
    printf("j is smaller than k\n");
  }
```

Now look at the program 7-3, which is the modified form of program 7-2. In this program two statements are performed when the condition is true.

```
/* program 7-3.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  float  cost, tax, luxury, total;
  luxury = 0.0;
  printf("Enter the cost of the item: ");
  scanf("%f", &cost);
  tax = cost * 0.06;
  if (cost > 40000.00)
  {
     luxury = cost * 0.005;
     printf("The luxury tax is %.2f\n", luxury);
  }
  total = cost + tax + luxury;
```

CamScanner

```
  printf("The total cost is %.2f", total);

  getch();
}
```

Nested if Statements

An if condition can be written as deeply as you need within the body of another if statement. This is called *nesting* of if's. For example an action is performed if two conditions are true. The condition 1 is tested and then condition 2 is tested.

```
if(condition 1)
   if(condition 2)
      statement;
```

The second if command in the following program segment is nested in the first:

```
if(income > 100000)
   if(status == 'S')
      tax_rate = 0.35;
```

The second if condition is tested only when the first condition is true, so the variable tax_rate is assigned only when both conditions are true. The same logic could be written as:

```
if(income > 100000 && status == 'S')
   tax_rate = 0.35;
```

Both perform the same task. The nested if statement can be visualised using the flowchart, as shown in Fig. 7.4.

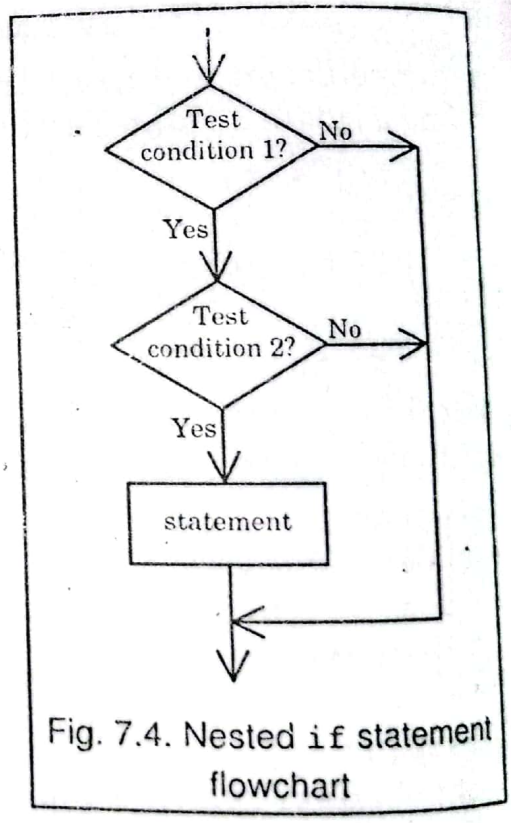


Fig. 7.4. Nested if statement flowchart

Consider another simple program:

```
/* program 7-4.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  if(getche() == 'm')
     if(getche() == 'e')
      printf("\nYou typed me.");

  getch();
}
```

In the above program, the inner if statement will not be executed unless the outer one is true, and the printf() statement will not be executed unless both if statements are true. Analysis of the program shows:

b	→	a non 'm' as first letter terminates the program
mf	→	a non 'e' as second letter terminates the program
me	→	only 'm' followed by 'e' executes printf() statement

CamScanner

Logical Operators

The `if` statement test only a single variable and a single value. But sometimes programs need to test for more than one value.

Look at the program 7-5, which asks whether the item is taxable or not, then if the item is taxable, adds a 6 percent sales tax.

```
/* program 7-5.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  int taxable;
  float cost, tax;
  tax = 0.0;
  printf("Enter the cost of the item: ");
  scanf("%f", &cost);
  printf("\nEnter Y if the item is taxable, N if non-taxable: ");
  taxable = getch();
  if(taxable == 'Y')
     tax = cost * 0.06;
  printf("\n\nThe total due is %.2f", cost + tax);

  getch();
}
```

This program has a serious flaw in it. The program assumes the user will enter an uppercase letter *Y* if the item is taxable. If the user enters a lowercase *y*, the program would assume the item was not taxable because the `if` command tests only for an uppercase *Y*.

In such situations, you should test for both possible inputs—*Y* or *y*. You could do this with two if commands. But instead, use the logical OR operator (||) like this:

```
if(taxable == 'Y' || taxable == 'y')
```

The instruction says "If `taxable` equals *Y* OR `taxable` equals *y*". So the item will be considered taxable if one or the other condition is true. If neither is true—you enter something other than *Y* or *y*—the item would be non-taxable.

You can use all the three logical operators with `if` statement: || for OR, && for AND, and ! for NOT. When you use the OR operator, one or the other condition (or both) must be true for the `if` statement to be true. When you use the AND condition, both must be true. When you use the NOT operator the condition must be false.

For example, suppose you need to write a program that inputs the user's income and number of dependents, as in program 7-6:

```
/* program 7-6.cpp */
#include<stdio.h>
#include<conio.h>
```

CamScanner

```
void main(void)
{
  clrscr();
  int dependents;
  float income;
  puts("Please enter your income");
  scanf("%f", &income);
  puts("Please enter the number of dependents");
  scanf("%d", &dependents);
  if(income < 5000 && dependents > 5)
     puts("You owe no income tax");

  getch();
}
```

The if condition tests two variables, income and dependents. The income must be less than Rs. 5000 AND the number of dependents must be greater than 5 for the message to be displayed. If either condition were false, such as one 2 dependents or Rs. 5001 of income, the puts() statement would not be performed.

Example 7.1

This program tells whether a number entered from the keyboard is positive or negative.

```
/* program 7-7.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  int num;
  printf("enter a number: ");
  scanf("%d", &num);
  if(num > 0)
     printf("number is positive");
  if(num < 0)
     printf("number is negative");

  getch();
}
```

Example 7.2

This program displays two numbers and asks the user what the answer is. Th program then tells the user if the answer is right or wrong.

```
/* program 7-8.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
```

CamScanner

```
{
  clrscr();
  int answer;
  printf("What is 10 + 15? ");
  scanf("%d", &answer);
  if(answer == 10+15)
     printf("Correct!");

  getch();
}
```

Example 7.3

This program either converts feet to metres or metres to feet, depending upon what the user requests.

```
/* program 7-9.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  float num;
  int choice;
  printf("Enter value: ");
  scanf("%f", &num);
  printf("\n1: feet to metres, \n2: metres to feet ");
  printf("\n\nEnter choice: ");
  scanf("%d", &choice);
  if(choice == 1)
     printf("%f", num/3.281);
  if(choice == 2)
     printf("%f", num*3.281);

  getch();
}
```

Example 7.4

The current luxury tax on cars is 10 percent but the tax is applied only to the portion of the sales price over Rs. 100,000. Write a program to calculate the tax for a given sales price.

Algorithm

Step 1. Input cost.
Step 2. IF cost > Rs. 100,000
Step 3. tax = (cost − 100,000)*tax_rate.
Step 4. Output tax.
Step 5. [End of IF structure.]
Step 6. Exit.

CamScanner

Source Code

```
/* program 7-10.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  const float cut_off = 100000;
  const float tax_rate = 0.1;
  float cost, tax;
  printf("\nLuxury tax computer ... Enter cost of car: ");
  scanf("%f", &cost);

  if(cost > cut_off)
     {
       tax = (cost - cut_off)*tax_rate;
       printf("\nFor a car costing Rs. %8.2f, a luxury tax of Rs.
       %8.2f is due.", cost, tax);
     }
  printf("\n\nThanks you for cooperation.");

  getch();
}
```

Output

```
Luxury tax computer ... Enter cost of car: 200000
For a car costing Rs. 200000.00, a luxury tax of Rs. 10000.00 is due.
Thanks you for cooperation.
```

The if-else Selection Structure

The if selection structure performs an indicated action only when the condition is true, otherwise the action is skipped. Whereas, if-else selection structure performs certain action when the condition is true and some different action when the condition is false.

The general structure of the command is:

```
if(condition)
   statement 1;
else
   statement 2;
```

The structure says "if the condition is true, perform the statement 1; else perform the statement 2". The statement 2 which follows else is performed only if the condition is false. If you need to perform more than one statement in either case, enclose the multiple statements in braces. You need a semicolon after each statement but not after the

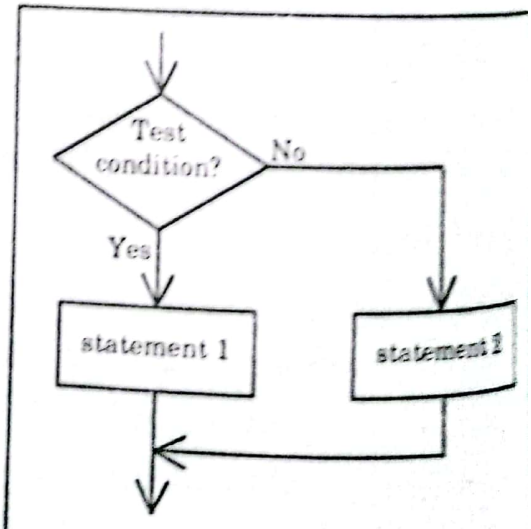


Fig. 7.5. if-else statement flowchart

else keyword. The if-else statement can be visualised using the flowchart as shown in Fig. 7.5.

For example, to find the larger of two variables you can use if-else statement:

```
if(x > y)
   max = x;          /*NOTE: semicolon required here! */
else
   max = y;
```

If x > y is *true*, then max will be assigned the value of x, and if it is *false*, then max will be assigned the value of y. As before, we can make statement 1 and statement 2 either single or compound by placing braces appropriately.

To print the message when no luxury tax is charged, you would modify the program 7-3 as illustrated below:

```
if (cost > 40000.00) .
{
   luxury = cost * 0.005;
   printf("The luxury tax is %.2f\n", luxury);
}
else
{
   puts("There is no luxury tax for this item");
   luxury = 0.0;
}
```

Note that we no longer need to assign luxury an initial variable at the start of the program because the if command now accounts for all possible conditions.

Example 7.5

You can use the else to create more efficient code in some cases. For example, here the else is used in place of a second if in the program given in example 7.1, which determines whether a number is positive or negative.

```
/* program 7-11.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  int num;
  printf("enter a number: ");
  scanf("%d", &num);
  if(num > 0)
     printf("number is positive");
  else
     printf("number is negative");

  getch();
}
```

CamScanner

Since there are only two possibilities, there is no reason to test num again to see if it is 0 or greater. The program in example 7.1 tested for num to be greater than 0 using a second `if` statement. Because of the way a C compiler generates code, the `else` requires far fewer machine instructions than an additional `if`.

Example 7.6

This program prompts the user for two numbers, divides the first by the second, and displays the result. However, division by 0 is undefined, so the program uses an `if` and an `else` statement to prevent division by 0 from occurring.

```
/* program 7-12.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  int num1, num2;
  printf("enter first number: ");
  scanf("%d", &num1);
  printf("enter second number: ");
  scanf("%d", &num2);
  if(num2 == 0)
     printf("cannot divide by zero");
  else
     printf("answer is: %d", num1/num2);

  getch();
}
```

Practical 3

Object

Writing a Program which uses operator (calculate the area of triangle, volume of sphere and arrange the resultant values in ascending order).

Algorithm

Step 1. Read a, b, r.

Step 2. Set Area of Triangle $t = \frac{1}{2} a\, b$.

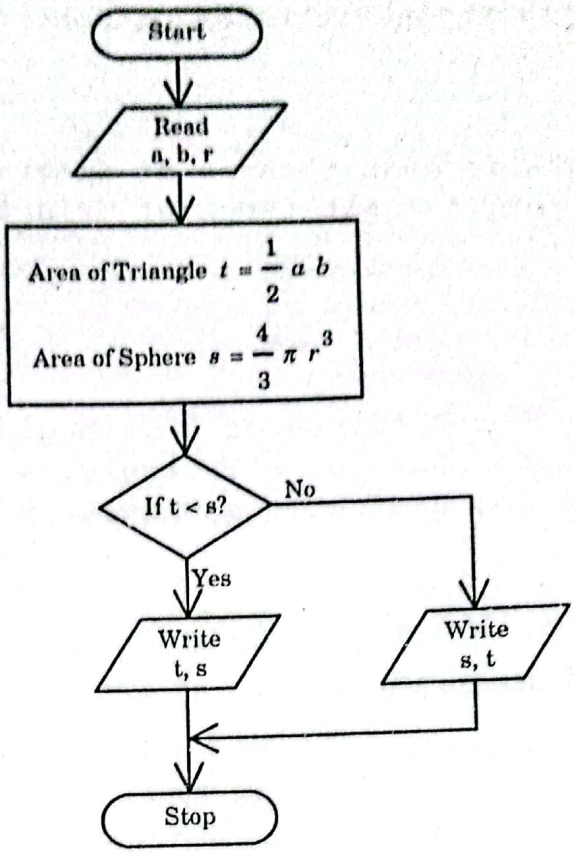
Area of Sphere $s = \frac{4}{3} \pi\, r^3$.

Step 3. IF t < s, then
Write t, s.
ELSE
Write s, t.
[End of IF-ELSE structure.]

Step 4. Exit.

CamScanner

Flowchart



Source Code

```
/* Practical 3: Writing a Program which uses operator (calculate
the area of triangle, volume of sphere and arrange the resultant
values in ascending order).*/

#include<stdio.h>
#include<conio.h>
#include<math.h>
#define pi 3.1415927
void main(void)
{
float a, b, r, t, s;
clrscr();
printf("\n\n\t\tEnter the value for Altitude of a Triangle. ");
scanf("%f", &a);
printf("\n\n\t\tEnter the value for Base of a Triangle. ");
scanf("%f", &b);
printf("\n\n\t\tEnter the value for Radius of a Sphere. ");
scanf("%f", &r);

t=(1.0/2.0)*(a*b);
s=(4.0/3.0)*pi*pow(r,3);
  . . if (t<s)
```

CamScanner

```
            {
            printf("\n\n\t\tArea of Triangle = %.2f",t);
            printf("\n\n\t\tArea of Sphere = %.2f",s);
            }
      else
            {
            printf("\n\n\t\tArea of Sphere = %.2f",s);
            printf("\n\n\t\tArea of Triangle = %.2f",t);
            }
 getch();
 }
```

Output

```
Enter the value for Altitude of a Triangle. 3.65
Enter the value for Base of a Triangle. 4.98
Enter the value for Radius of a Sphere. 9.29

Area of Triangle = 9.09
Area of Sphere = 3358.43
```

Nested if-else Statements

Nested `if-else` statements test for multiple conditions by placing `if-else` statements inside `if-else` statements. When a condition is evaluated as true, the corresponding statements are executed and the rest of the structure is skipped. This structure is referred to as the *`if-else-if` ladder*.

To demonstrate the use of the ladder consider the following program which convert a month number to a month name:

```
/* program 7-13.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  int month;
  printf("Input month number: ");
  scanf("%d", &month);

  if(month == 1)
     printf("January");
  else if (month == 2)
     printf("February");
  else if (month == 3)
     printf("March");
  else if (month == 4)
     printf("April");
  else if (month == 5)
     printf("May");
```

CamScanner

```
  else if (month == 6)
     printf("June");
  else if (month == 7)
     printf("July");
  else if (month == 8)
     printf("August");
  else if (month == 9)
     printf("September");
  else if (month == 10)
     printf("October");
  else if (month == 11)
     printf("November");
  else if (month == 12)
     printf("December");
  else
     printf("Error in month number!!!!");
  getch();
}
```

As another example of ladder, the following program will print A-1 for exam percentage greater than or equal to 80, A for percentage greater than or equal to 70, B for percentage greater than or equal to 60, C for percentage greater than or equal to 50, D for percentage greater than or equal to 40, E for percentage greater than or equal to 33, and F for Fails. The program may be written in C as

```
/* program 7-14.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  float percent;
  printf("Input percentage of marks: ");
  scanf("%f", &percent);
  if(percent >= 80)
     printf("Your grade is A-1\n");
  else if (percent >= 70)
     printf("Your grade is A\n");
  else if (percent >= 60)
     printf("Your grade is B\n");
  else if (percent >= 50)
     printf("Your grade is C\n");
  else if (percent >= 40)
     printf("Your grade is D\n");
  else if (percent >= 33)
     printf("Your grade is E\n");
  else
     printf("Your grade is F\n");
  getch();
}
```

CamScanner

If the variable percent is greater than or equal to 80, the first six conditions will be true, but only the printf statement after the first test will be executed. After that printf is executed, the else-part of the "outer" if-else statement is skipped.

The if selection structure expects only one statement in its body. To include several statements in the body of an if, enclose the set of statements in braces. A set of statements contained within a pair of braces is called a *compound statement*.

The following example includes a compound statement in the else part of an if-else structure.

```
if (percent >= 33)
   printf("Passed.\n");
else
   {
   printf("Failed.\n");
   printf("You must work hard.\n");
   }
```

In this case, if percent is less than 33, the program executes both printf statements in the body of the else and prints

```
Failed.
You must work hard.
```

The ladder is diagrammed in Fig. 7.6. Notice that in this structure when a true path is found, the corresponding statement or compound statement executes and the program skips over the rest of the structure. The final else although not necessary, is often included as a default case and executes only if none of the expressions is true.

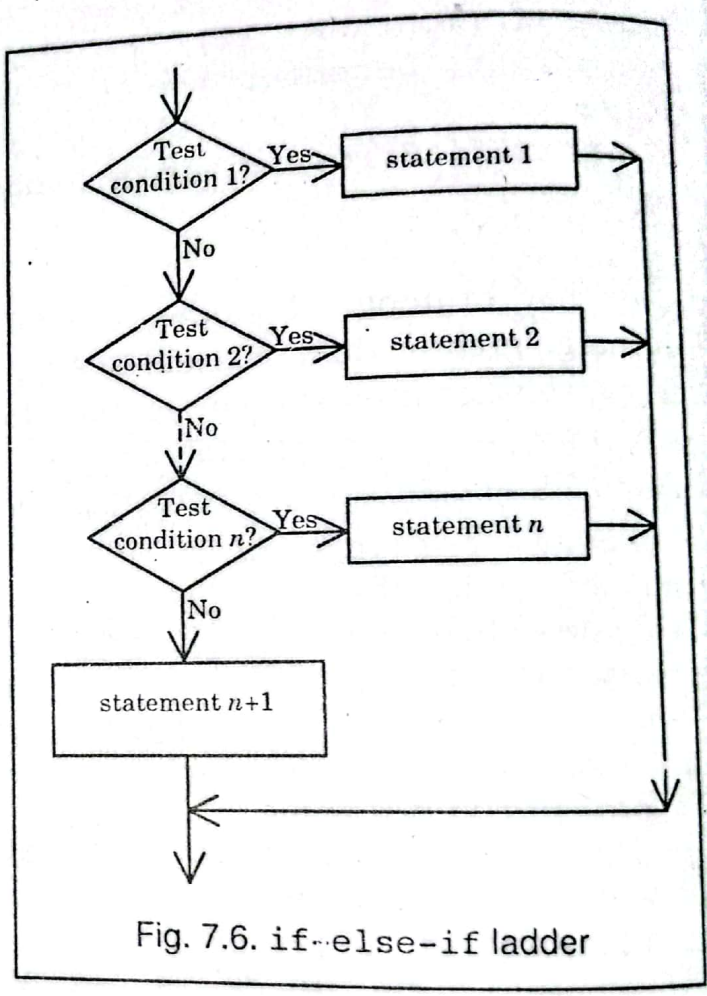


Fig. 7.6. if-else-if ladder

Example 7.7

We can improve the program 6-4 by using an if-else-if ladder, as shown here.

```
/* program 7-15.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
clrscr();
float a, b;
char ch;
printf("Do you want to:\n");
printf("add, subtract, multiply, or divide?\n\n\n");
printf("Enter first letter: ");
ch = getche();          /* read a character */
```

CamScanner

```
printf("\n");
printf("Enter first number: ");
scanf("%f", &a);
printf("Enter second number: ");
scanf("%f", &b);
if(ch=='a')
   printf("%.2f", a+b);
else if(ch=='s')
   printf("%.2f", a-b);
else if(ch=='m')
   printf("%.2f", a*b);
else if(ch=='d' && b!=0)
   printf("%.2f", a/b);
getch();
}
```

This is an improved version of the program 6-4 because once a match is found, any remaining `if` statements are skipped. This means that the program is not wasting time on needless operations.

The switch Statement

The `if-else` statement is good for choosing among few (two to three) alternatives. It quickly becomes confusing when several alternatives are needed. An alternative is the `switch` statement. The `switch` statement is used to select one of the several alternatives, such as menu options. The `switch` statement is especially useful when the selection is based on the value of a single variable (called *controlling variable*) or of a simple expression (called the *controlling expression*). The value of this variable may be of type `int` or `char`, but not of type `float`. The `switch` statement also differs from `if`, in that `switch` can only test for equality, whereas the `if` conditional expression can be of any type. The general form of the `switch` statement is

```
switch(variable)                    /* may use switch(expression) */
{
case constant 1:
   statement(s);
   break;
case constant 2:
   statement(s);
   break;
case constant 3:
   statement(s);
   break;
.
.
.
default:
   statement(s);
}
```

A *variable* is tested against a list of integer or character constants. When the value matches a `case` condition, the statement(s) following the case is executed. The `break` statement at the end of each `case` sends control to the end of the `switch`, so once a `case` is performed, the remaining `case` statements are ignored and the `switch` ends. The `default` statement(s) is performed if no matches are found. The `default` is optional. If all matches fail and `default` is absent, no action takes place.

The general `switch` multiple-selection structure (using a `break` in each `case`) is flowcharted in Fig. 7.7. The flowchart makes it clear that each `break` statement at the end of a `case` causes control to immediately exit the `switch` structure.

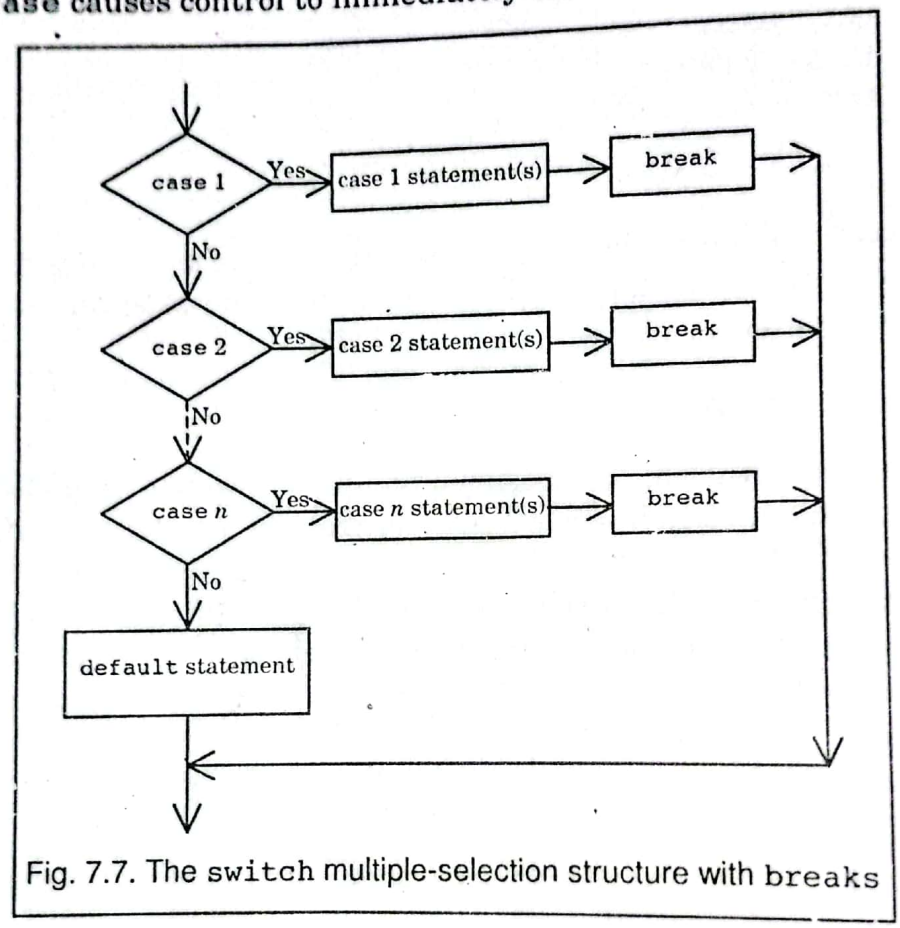


Fig. 7.7. The `switch` multiple-selection structure with `breaks`

As a very simple example, this program recognises the numbers 1, 2, 3, and 4 and prints the name of the one you enter. That is, if you enter 2, the program displays `two`.

```
/* program 7-16.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
clrscr();
int i;
printf("Enter a number between 1 and 4: ");
scanf("%d", &i);
switch(i)
   {
   case 1:
```

CamScanner

```
      printf("one");
      break;
   case 2:
      printf("two");
      break;
   case 3:
      printf("three");
      break;
   case 4:
      printf("four");
      break;
   default:
      printf("unrecognised number");
   }
getch();
}
```

Now consider the following example:

```
/* program 7-17.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
clrscr();
int num;
printf("Enter a Number from -1 to 4: ");
scanf("%d", &num);

switch(num)
   {
   case -1:
      printf("\nInput value = -1");
      break;
   case 0:
      printf("\nInput value = 0");
      break;
   case 1:
   case 2:
   case 3:
   case 4:
      printf("\nInput value between 1-4");
      break;
   default:
      printf("Input value < -1 or > 4");
   }

getch();
}
```

CamScanner

In this example there are three expected case values: −1, 0, and 1 through 4. The absence of a break statement for case values 1, 2, and 3 means that if any of those numbers happens to be the value of num, the program falls through to case value 4 and executes the statements there, ending with break. Any other input value is caught by the default label, which does not need a break statement if it is listed last.

In principle, a switch may occur within another, but in practise it is rarely done. Such statements would be called nested switch statements.

Example 7.8

The statement sequence associated with a case may be empty. This allows two or more cases to share a common statement sequence without duplication of code. For example, here is a program that categorises letters into vowels and consonants.

```
/* program 7-18.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
clrscr();
char ch;
printf("Enter the letter: ");
ch = getche();
switch(ch)
   {
   case 'a':
   case 'e':
   case 'i':
   case 'o':
   case 'u':
      printf(" is a vowel\n");
      break;
   default:
      printf(" is a consonant");
   }
getch();
}
```

Example 7.9

It is always possible to replace an if statement with an equivalent switch statement (and vice versa). So we can rewrite program 7-15 by using switch statement.

```
/* program 7-19.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
clrscr();
```

CamScanner

```
int c;
float a, b;
printf("\n\n\n");
printf("\t\t\tMENU");
printf("\n\t\t\t____\n\n");
printf("1. Add\n");
printf("2. Subtraction\n");
printf("3. Multiply\n");
printf("4. Divide\n");
printf("\n");
printf("Enter the Menu option: ");
scanf("%d", &c);
printf("Enter first number: ");
scanf("%f", &a);
printf("Enter second number: ");
scanf("%f", &b);

switch(c)
   {
   case 1:
      printf("%.2f", a+b);
      break;
   case 2:
      printf("%.2f", a-b);
      break;
   case 3:
      printf("%.2f", a*b);
      break;
   case 4:
      if (b != 0)
      printf("%.2f", a/b);
      break;
   default:
      printf("Invalid Menu option\n\n");
   }

getch();
}
```

Example 7.10

The program displays one of five messages based on the value of grade (type char). If grade is 'D', 'd', or 'F', 'f', the student is put on probation. If grade is not listed in the case labels, displays an error message.

```
/* program 7-20.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
```

CamScanner

```
{
clrscr();
char grade;
printf("Enter Grade of the Student: ");
scanf("%c", &grade);

switch(grade)
   {
   case 'A':
   case 'a':
      printf("Excellent");
      break;
   case 'B':
   case 'b':
      printf("Good");
      break;
   case 'C':
   case 'c':
      printf("O.K.");
      break;
   case 'D':
   case 'd':
   case 'F':
   case 'f':
      printf("Poor, student is ");
      printf("on probation");
      break;

   default:
      printf("Invalid letter grade");
   }

getch();
}
```

Practical 8

Object

Writing a Program which uses a switch statement and breaks the program if certain condition is observed *i.e.* writing a Program which uses a switch, case and break statements and interconvert the temperature scales using Menu options.

Algorithm

Step 1. Read switch variable.

Step 2. IF switch variable = 1

a. Read c.

b. Convert Celsius to Fahrenheit using $f = 32 + \frac{9}{5} c$.

CamScanner

IF switch variable = 2

a. Read f.

b. Convert Fahrenheit to Celsius using $c = \frac{5}{9}(f - 32)$.

IF switch variable = 3

a. Read c.

b. Convert Celsius to Kelvin using $k = c + 273.15$.

IF switch variable = 4

a. Read f.

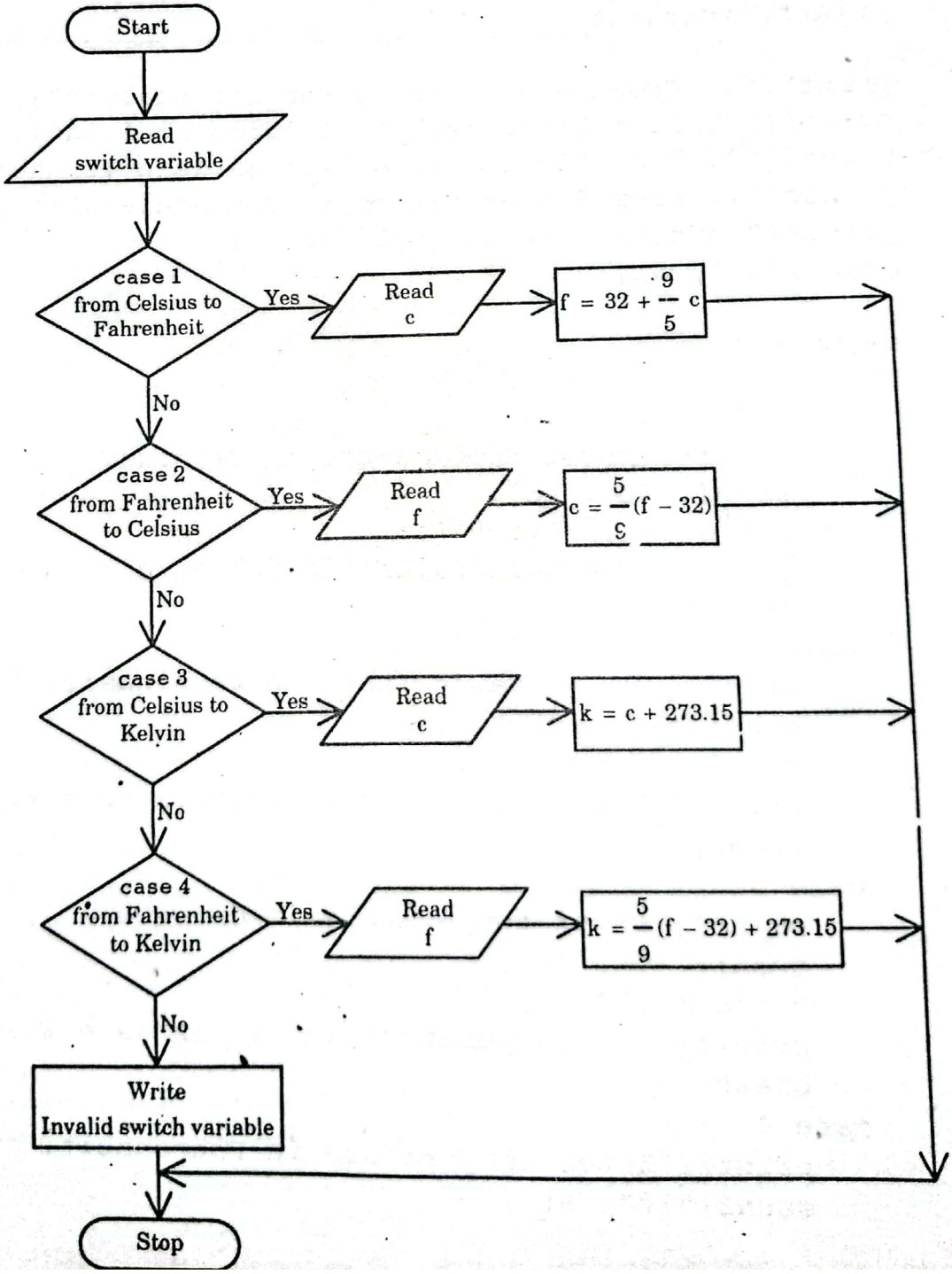
b. Convert Fahrenheit to Kelvin using $k = \frac{5}{9}(f - 32) + 273.15$.

Step 3. ELSE

Write Invalid switch variable.

Step 4. Exit.

Flowchart



Source Code

```
/* Practical 8: Writing a program which uses a switch statement
and breaks the program if certain condition is observed.
Writing a Program which uses a switch, case and break statements
and interconvert the temperature scales using Menu options.*/

#include<stdio.h>
#include<conio.h>
void main(void)
{
clrscr();
int i;
float c, f, k;
printf("\n\n\n");
printf("\t\t\tMENU FOR TEMPERATURE SCALE CONVERSION");
printf("\n\t\t\t_____\n\n");

printf("1. From Celsius to Fahrenheit scale\n");
printf("2. From Fahrenheit to Celsius scale\n");
printf("3. From Celsius to Kelvin scale\n");
printf("4. From Fahrenheit to Kelvin scale\n");
printf("\nEnter the MEMU option: ");
scanf("%d", &i);

switch(i)
   {
   case 1:
      printf("Enter temperature in Celsius: ");
      scanf("%f", &c);
      f = 32.0+(9.0/5.0)*c;
      printf("The temperature in Fahrenheit is = %.2f", f);
      break;
   case 2:
      printf("Enter temperature in Fahrenheit: ");
      scanf("%f", &f);
      c = 5.0/9.0*(f-32.0);
      printf("The temperature in Celsius is = %.2f", c);
      break;
   case 3:
      printf("Enter temperature in Celsius: ");
      scanf("%f", &c);
      k = c + 273.15;
      printf("The temperature in Kelvin is = %.2f", k);
      break;
   case 4:
      printf("Enter temperature in Fahrenheit: ");
      scanf("%f", &f);
```

CamScanner

```
        k = (5.0/9.0*(f-32.0)) + 273.15;
        printf("The temperature in Kelvin is = %.2f", k);
        break;
    default:
        printf("Invalid MENU option\n\n");
    }

getch();
}
```

Output

MENU FOR TEMPERATURE SCALE CONVERSION

```
1. From Celsius to Fahrenheit scale
2. From Fahrenheit to Celsius scale
3. From Celsius to Kelvin scale
4. From Fahrenheit to Kelvin scale

Enter the MEMU option: 2
Enter temperature in Fahrenheit: 145.3
The temperature in Celsius is = 62.94
```

CamScanner

Exercise

7.1. Write a program that asks the user for an integer and then tells the user if that number is even or odd. (Hint: Use C's modulus operator %).

7.2. Write a program that inputs a number, then reports whether the number is in the range from 1 to 100.

7.3. How do you perform more than one statement when a condition is true?

7.4. The following is supposed to cause an action whenever x is greater than y.

```
if(x > y) then
printf("x is greater than y");
```

Why is this wrong?

Ans. Unlike other languages, in C the `if` statement does not use the then keyword.

7.5. The following is supposed to cause an action whenever x is equal to y.

```
if(x = y)
printf("x equals y");
```

Is there anything wrong with this?

Ans. This is a common mistake in C. The expression mistakenly uses the assignment operator = instead of a relational operator ==. The effect is to assign the value of y to x. The expression then takes on the value of the assignment. If y happens to be 0, the assignment makes x equal 0. Because 0 is interpreted as *false*, the `printf` statement cannot execute even though x does in fact equal y!

7.6. Write a program that demonstrates multiple statements following `if`.

Ans.

```
/* program 7-21.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  char ch;
  ch = getche();
  if(ch == 'z')
    {
    printf("\n\nYou typed z.");
    printf("\nNot some other letter.");
    }

  getch();
}
```

7.7. What is the propose of the `else` command?

7.8. Write a program that requests two numbers and then displays either their sum or product, depending upon what the user selects.

CamScanner

7.9. Explain how an `if` command can be used to test for valid input.

7.10. Explain the use of nested `if` statements.

7.11. Write a program that reads temperature and prints a message as given below:

Temperature	*Message*
> 35	Its hot day!
≥25 and ≤ 35	Its nice day!
< 25	Its cold day!

7.12. Write a program that reads three numbers and prints the largest.

7.13. What is the difference between the `if-else-if` ladder and the series of one-way `if` statements?

Ans. The one-way `if` series is not as efficient as the ladder and does not show mutual exclusion because more than one statement can execute. In fact, all the expressions could be true, so each `if` statement has to be checked to determine its logical value.

7.14. Correct the syntax errors.

```
if x > 25.0
{
   y = x
else
   y = z;
}
```

7.15. What value is assigned to `fee` by the `if` statement when `speed` is 75?

```
if(speed > 35)
   fee = 20.0;
else if(speed > 50)
   fee = 40.0;
if(speed > 75)
   fee = 60.0;
```

7.16. Explain the difference between the statements on the left and the statements on the right. For each group of statements, give the final value of `x` if the initial value of `x` is 1.

```
if(x >= 0)
   x = x + 1;
else if(x >= 1)
   x = x + 2;
```

```
if(x >= 0)
   x = x + 1;
if(x >= 1)
   x = x + 2;
```

7.17. The ASCII codes for the lowercase letters are separated from the uppercase letters by a difference of 32. Therefore, to convert a lowercase letter to an uppercase one, simply subtract 32 from it. Write a program that reads characters from the keyboard and displays lowercase letters as uppercase ones. Stop when Enter is pressed.

7.18. Using a nested `if` statement, write a program that prompts the user for a number and then reports if the number is positive, zero, or negative.

7.19. How does the `switch` command differ from the `if` command?

7.20. What is wrong with this segment of program?

CamScanner

```
float f;
scanf("%f", &f);
switch(f)  {
case 12.05:
.
.
```

7.21. Write a switch statement that outputs messages indicating what day has been numerically input (for example, 1 → Monday, 2 → Tuesday ...).

7.22. What output line(s) are displayed by the statements that follow when grade is 'I'? When grade is 'B'? When grade is 'b'?

```
switch(grade)
   {
   case 'A':
      points = 4;
      break;
   case 'B':
      points = 3;
      break;
   case 'C':
      points = 2;
      break;
   case 'D':
      points = 1;
      break;
   case 'E':
   case 'I':
   case 'Y':
      points = 0;
   }
   if(points > 0)
      printf("Passed, points earned = %d\n", points);
   else
      printf("Failed, no points earned\n");
```

7.23. What will be printed by this carelessly constructed switch statement if the value of colour is 'R'?

```
switch(colour)          /* break statements missing */
   {
   case 'R':
      printf("red\n");
   case 'B':
      printf("blue\n");
   case 'Y':
      printf("yellow\n");
   }
```

7.24. Write a program that computes the area of either a circle, rectangle, or triangle. Use an if-else-if ladder. Rewrite this program using switch statement.

7.25. Improve the program 7-20 by using an toupper() or tolower() function.

CamScanner

Objectives and MCQs

Choose the answer that best completes the statement.

7.1. Structured programming
- (a) is hard to follow but worthwhile
- (b) calls for *one way in, one way out* structures
- (c) makes debugging difficult
- (d) relies heavily on `if-else` statements

7.2. Which of these expressions are true
- (a) 0
- (b) 1
- (c) 5*3 < 15
- (d) 1 == 1
- (e) −1

7.3. The statement `if(x = 0) printf("X = 0");`
- (a) is incorrect syntax
- (b) is correct syntax, but `X = 0` will never print
- (c) causes a run-time error
- (d) has no effect on the program

7.4. In a simple `if` statement with no `else`, what happens if the condition following the `if` is false?
- (a) the program searches for the last `else` in the program
- (b) nothing, or control falls through to the statements following the `if`
- (c) the body of the `if` statement is executed
- (d) the program as a whole is executed

7.5. The statements following a particular `if-else` in an `if-else-if` ladder are executed when
- (a) the conditional expression following the `if-else` is true and all previous conditions are true
- (b) the conditional expression following the `if-else` is true and all previous conditions are false
- (c) the conditional expression following the `if-else` is false and all previous conditions are true
- (d) the conditional expression following the `if-else` is false and all previous conditions are false

7.6. Which `if` in a program does an `else` pair up with?
- (a) the last `if` with the same indentation as the `else`
- (b) the last `if` not matched with its own `else`
- (c) the last `if` not enclosed in braces
- (d) the last `if` not enclosed in braces and not matched with its own `else`

CamScanner

7.7. The advantage of a `switch` statement over an `if-else-if` construction is
(a) a default condition can be used in the `switch`
(b) the `switch` is easier to understand
(c) several different statements can be executed for each case in a `switch`
(d) several different conditions can cause one set of statements to be executed in a `switch`

7.8. The `switch` statement
(a) can always be used as a substitute for the ladder structure
(b) can be used only with an `if-else` statement
(c) is most useful when testing a variable having a limited range of possibilities
(d) all of the above

Objectives

7.9. Structured programming is a standardised approach that creates a program as a series of interrelated sub-programs, or functions.

7.10. An `if` condition can be written as deeply as you need within the body of another `if` statement. This is called nesting of `if`'s.

7.11. The `if` selection structure performs an indicated action only when the condition is true, otherwise the action is skipped.

7.12. The `if-else` selection structure performs certain action when the condition is true and some different action when the condition is false.

7.13. The `if-else` statement behaves like the logical operator XOR.

7.14. Nested `if-else` statements test for multiple conditions by placing `if-else` statements inside `if-else` statements.

7.15. An expression containing the OR operator is true if either or both of its operands is true.

7.16. When evaluating a ladder structure, the program stops testing after the first true expression is found.

7.17. The ladder structure facilitates testing by mutual exclusion.

7.18. The switch statement can sometimes be used as an alternative to the ladder structure.

7.19. The `switch` statement also differs from `if`, in that `switch` can only test for equality, whereas the `if` conditional expression can be of any type.

7.20. The switch selection structure is used to test a particular variable or expression for each of the constant integral values it may assume.

7.21. A `switch` may occur within another, such statements are called nested `switch` statements.

7.22. The default case is optional in the switch selection structure.

7.23. The `break` statement is used to exit the `switch` structure.

CamScanner

CHAPTER 8

ITERATION CONTROL STRUCTURES

Iteration is often referred to as a loop, because the program will keep repeating the activity until the condition becomes false. Loops simplify programming because they enable the programmer to specify certain instructions only once to have the computer execute them many times. C provides following three types of iteration structures:

1. `for` loop
2. `while` loop
3. `do-while` loop

Any of these can repeat either an instruction, a series of instructions. or an entire program. The `while` and `do-while` loops are used for *indefinite repetitions*. That is, they do not define the number of repetitions that will occur when the loop is executed. They merely give conditions for looping to stop. The `for` loop is used for *definite repetition*. It defines the number of repetitions that will occur during execution.

The `for` Loop

The `for` loop is used to repeat a statement or a block of statements for a specified number of times. The `for` loop statement includes the initialisation of the counter, the condition, and the increment. The general form of the `for` statement is illustrated in Fig. 8.1. Notice that you put the terminating semicolon after the

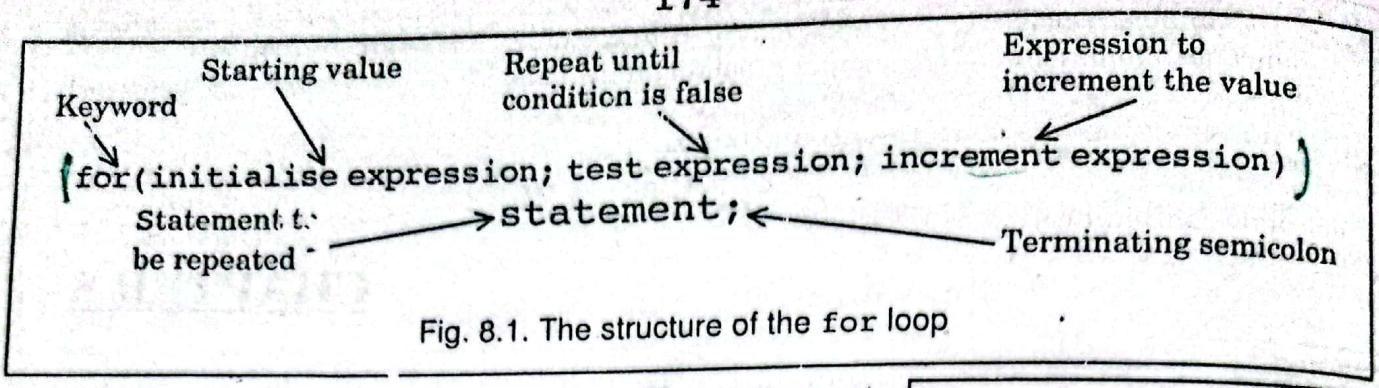


Fig. 8.1. The structure of the for loop

statement, not after the for argument. The statement can be simple or compound enclosed in braces. The three expressions within the for loop statement, however, must be separated by semicolons. The for loop statement can be visualised using the flowchart as shown in Fig. 8.2.

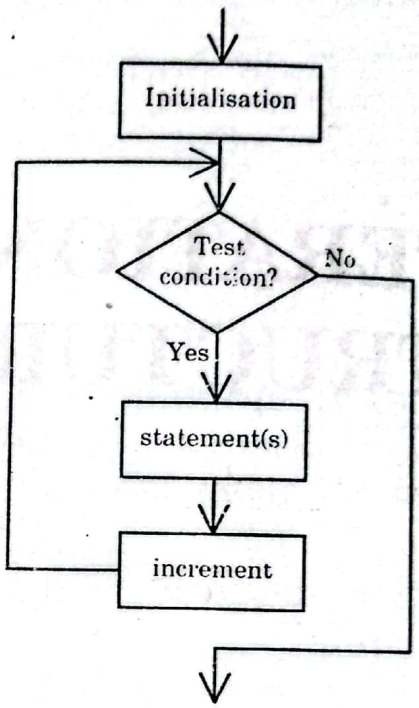


Fig. 8.2. Flowchart format of for loop

For example, this program uses a for loop to print numbers from 1 to 10.

```
/* program 8-1.cpp */
#include<stdio.h>
#include<conio.h>

void main(void)
{
  clrscr();
  int count;
  for(count = 1; count <= 10; count++)
     printf("%3d", count);

  getch();
}
```

Lets analyse the syntax of the for loop using the output of the program which is:
1 2 3 4 5 6 7 8 9 10.

`count = 1` Initialises the starting value of the variable count. You can use any name of the variable but it must be an integer type.

`count <= 10` Sets the condition to repeat the loop as long as count has a value less than or equal to 10.

`count++` Increments the value of count after each repetition.

With each repetition of the loop, the program displays the current value of the variable count.

When this program first encounters the loop, it sets the value of count to 1. It then checks the condition to see if the value of count is less than or equal to 10. Since that condition is true, it will perform the instruction that is associated with the loop, displaying the value of the variable.

CamScanner

After performing the instruction, it increments the value by 1, and again checks the condition. Since the condition is still true, it performs the instruction a second time, displaying the current value of the variable. The process is repeated until the value is incremented to 11. The condition count <= 10 is then false, so the instruction is not performed and the loop stops.

You may decrement the counter, but you have to use suitable test expression and initial value, for example:

```
for (count = 10; count >= 1; count--)
   printf("%3d", count);
```

Here the variable count starts from 10 and is decremented by 1. The loop will be executed as long as the count is greater than or equal to 1, in other words, ten times. The output will be 10 9 8 7 6 5 4 3 2 1.

Lets consider some variations. Suppose we want to print the odd integers from 11 through 21. The program segment would look like this:

```
for (count = 11; count <= 21; count = count + 2)
   printf("%3d", count);
```

The count is initialised to 11 and the expression has been changed to check for count <= 21. Because we want counting by 2, we substitute an assignment for the increment operator.

The following example prints the message "Take Care" five times on the screen:

```
/* program 8-2.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  int i;
  for (i = 1; i < 6; i++)
  {
     printf("Take Care\n");
  }
  getch();
}
```

The following program continues to loop until a q is entered from the keyboard. Instead of testing the repeat condition, the test expression in this for loop checks the value of a character entered by the user.

```
/* program 8-3.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  int i;
  char ch = 'a';  /* declare and give ch an initial value */
```

CamScanner

```
    for (i = 0; ch != 'q'; i++)
    {
       printf("Pass: %d\n", i);
       ch = getch();
    }
}
```

Here, the condition that controls the loop has nothing to do with the initialisation. The reason ch is given an initial value is to prevent it from accidentally containing a q when the program begins.

Each component of the for loop control structure is optional but the semicolons are needed. For example, this program asks the user for a value and then counts down to 0 from this number. Here, the loop is initialised by the user outside the loop, so initialisation portion of the loop is empty.

```
/* program 8-4.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  int i;
  printf("Enter an integer: ");
  scanf("%d", &i);
  for ( ; i; i--)
    printf("Take Care\n");
  getch();
}
```

Multiple Statements

To perform more than one statement in the loop, enclose the whole series in braces. For example, here is a program that converts 9 consecutive temperatures (from 32 to 40) to Celsius:

```
/* program 8-5.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  int f;
  float c;
     puts("Fahrenheit\tCelsius\n");
     for (f=32; f <= 40; f++)
     {
     c = 5.0/9.0*(f-32.0);
     printf("%d\t\t%6.2f\n", f, c);
     }
  getch();
}
```

CamScanner

Two statements are performed in each repetition of the loop. The value of the initialisation determines both the number of repetitions and the values to be converted to Celsius.

Nested for Loops

When one for command is performed within another, they are said to be nested. The inside loop is completely repeated for each repetition of the outside loop. Picture nested for loops as being two-dimensional and single for loops as one-dimensional.

As an example, look at the following program:

```
/* program 8-6.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  int row, column;
  for(row = 1; row <= 10; row++)    /* outer loop */
  {
      for (column = 1; column <= 10; column++)    /* inner loop */
      printf("*");

    printf("\n");        /* outside of second loop, inside first*/
      }

  getch();
}
```

The program displays 10 rows of 10 asterisks. It uses two integer variables, row and column. The outer loop increments the value of row from 1 to 10. With each repetition of the loop, however, the inner loop is performed 10 times, incrementing column and displaying a row of ten asterisks. So in total, 100 asterisks are displayed—10 inner loops (column positions) for each of 10 outer loops (rows). The output is shown in Fig. 8.3.

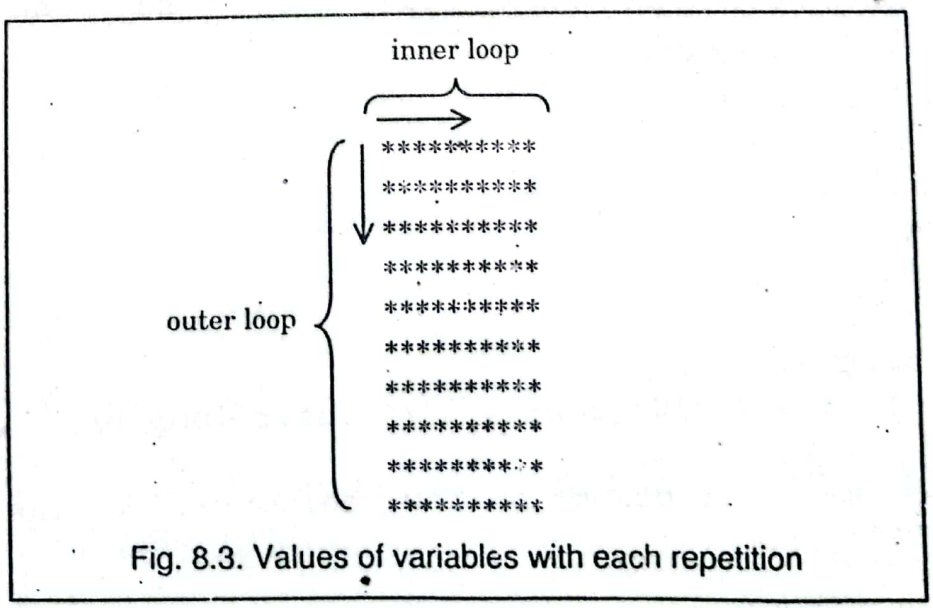


Fig. 8.3. Values of variables with each repetition

CamScanner

Program 8-7 illustrates another example of nested loops. It perform ten outer and ten inner loops to create a multiplication table. Instead of merely displaying an asterisk, it displays the value of row multiplied by the value of column.

```
/* program 8-7.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  int row, column;
  puts("\t\tMy Handy Multiplication Table\n");
  for(row = 1; row <= 10; row++)
    {
      for (column = 1; column <= 10; column++)
      printf("%6d", row*column);

      printf("\n");     /* outside of second loop, inside first*/
    }

  getch();
}
```

The output of the program is shown below:

```
                My Handy Multiplication Table

     1     2     3     4     5     6     7     8     9    10
     2     4     6     8    10    12    14    16    18    20
     3     6     9    12    15    18    21    24    27    30
     4     8    12    16    20    24    28    32    36    40
     5    10    15    20    25    30    35    40    45    50
     6    12    18    24    30    36    42    48    54    60
     7    14    21    28    35    42    49    56    63    70
     8    16    24    32    40    48    56    64    72    80
     9    18    27    36    45    54    63    72    81    90
    10    20    30    40    50    60    70    80    90   100
```

Finally, consider this program:

```
/* program 8-8.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  int row, column;
  for(row = 1; row <= 10; row++)    /* outer loop */
  {
     for (column = 1; column <= row; column++)  /* inner loop */
     printf("*");
```

CamScanner

```
        printf("\n");          /* outside of second loop, inside first*/
          }

    getch();
}
```

It displays the series of asterisks shown below:

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

Each row of asterisks has a different length. Instead of the inner loop always repeating the same number of times, the number increases with each repetition of the outer loop—one asterisk on the first line, two asterisks on the second line, and so on. The number of repetitions of the column is the same number as the row. We do this by using the variable row in the test expression of the inner loop.

Be cautious when writing programs that have two or more levels of for statements. A program that repeats 100 outer and 100 inner loops actually performs 10,000 repetitions!

Example 8.1

Write a program to print the sum of odd numbers from 1 to 100, *i.e.*,

sum = 1 + 3 + 5 + ... + 99

```
/* program 8-9.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  int i, sum=0;
    for(i=1; i<100; i=i+2)
       sum = sum + i;
    printf("\n\n\n\t\tThe sum is = %d", sum);

  getch();
}
```

Example 8.2

Write a program which uses for loop to print out a table of ASCII codes from 32 to 127 (compare it with program 4-1).

CamScanner

```
/* program 8-10.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  int i;
   for(i=32; i<128; i++)                /*code from 32 to 127*/
     printf("%3d = %c\t", i, i);        /*print as number and as char*/

  getch();
}
```

The output is shown below:

```
 32 =    33 = !  34 = "  35 = #  36 = $  37 = %  38 = &  39 = '  40 = (  41 = )
 42 = *  43 = +  44 = ,  45 = -  46 = .  47 = /  48 = 0  49 = 1  50 = 2  51 = 3
 52 = 4  53 = 5  54 = 6  55 = 7  56 = 8  57 = 9  58 = :  59 = ;  60 = <  61 = =
 62 = >  63 = ?  64 = @  65 = A  66 = B  67 = C  68 = D  69 = E  70 = F  71 = G
 72 = H  73 = I  74 = J  75 = K  76 = L  77 = M  78 = N  79 = O  80 = P  81 = Q
 82 = R  83 = S  84 = T  85 = U  86 = V  87 = W  88 = X  89 = Y  90 = Z  91 = [
 92 = \  93 = ]  94 = ^  95 = _  96 = `  97 = a  98 = b  99 = c 100 = d 101 = e
102 = f 103 = g 104 = h 105 = i 106 = j 107 = k 108 = l 109 = m 110 = n 111 = o
112 = p 113 = q 114 = r 115 = s 116 = t 117 = u 118 = v 119 = w 120 = x 121 = y
122 = z 123 = { 124 = | 125 = } 126 = ~ 127 = □
```

Example 8.3

Write a program which reads a number and prints its factorial using for loop.

```
/* program 8-11.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  double f=1;
  int num, i;
  // computes the product n*(n-1)*(n-2)*(n-3)*...*2*1
  printf("\n\n\n\t\tEnter the number to find its Factorial: ");
  scanf("%d", &num);
  if (num >= 0)
  {
   for(i=num; i>1; --i)
       f = f * i;
   printf("\n\n\n\t\tThe factorial of %d is %lf", num, f);
  }
  else
     printf("\n\n\n\t\t Sorry!!");
  getch();
}
```

CamScanner

The for Loop Variations

The for loop in C is significantly more powerful and flexible than in most other computer languages. The reason that for is so flexible is that each of the three parts inside the parentheses of the for statement are optional. You may omit any of them as long as your program contains the necessary statements to take care of the loop execution. For example the loop

```
for( ; index>20; )
   statement(s);
```

contains only the condition for termination. The initialisation is done before the loop. The incrementing must be included inside the loop block.

You can even omit all the expressions as in this example:

```
for(;;)
   printf("Hello\n");
```

This type of loop is usually called an *infinite loop*. It is repeated continuously unless you include a suitable condition inside the loop block to terminate the execution. If there is no such condition, you can only stop the repeated loop by pressing Ctrl-Break to abort the program execution. Even the statement(s) inside the loop are optional. Although accidentally creating an infinite loop is a bug, you will sometimes want to create one on purpose.

Further, there does not need to be any loop-control variable because the test expression may use some other means of stopping the loop. For example, following segment of program continue to loop until a p is entered at keyboard. Instead of testing a loop-control variable, the test expression in this for checks the value of a character.

```
int i;
char ch;
ch = 'a';          /* give ch an initial value */
for(i=0; ch != 'p'; i++)
   {
   printf("pass: %d\n", i);
   ch = getche();
   }
```

The initialise expression and increment expression of the for loop can also contain more than one statement separated by a comma. For instance,

```
for(j=0, k=100; k-j>0; j++, k--)
```

In this example, both j and k are initialised before the loop is entered. After each iteration, j is incremented and k is decremented. The use of multiple statements in for loop expression also demonstrates why semicolons are used to separate the three expressions. If commas had been used, they could not also have been used to separate multiple statements in the initialise expression and increment expression, without confusing the compiler.

CamScanner

Practical 4

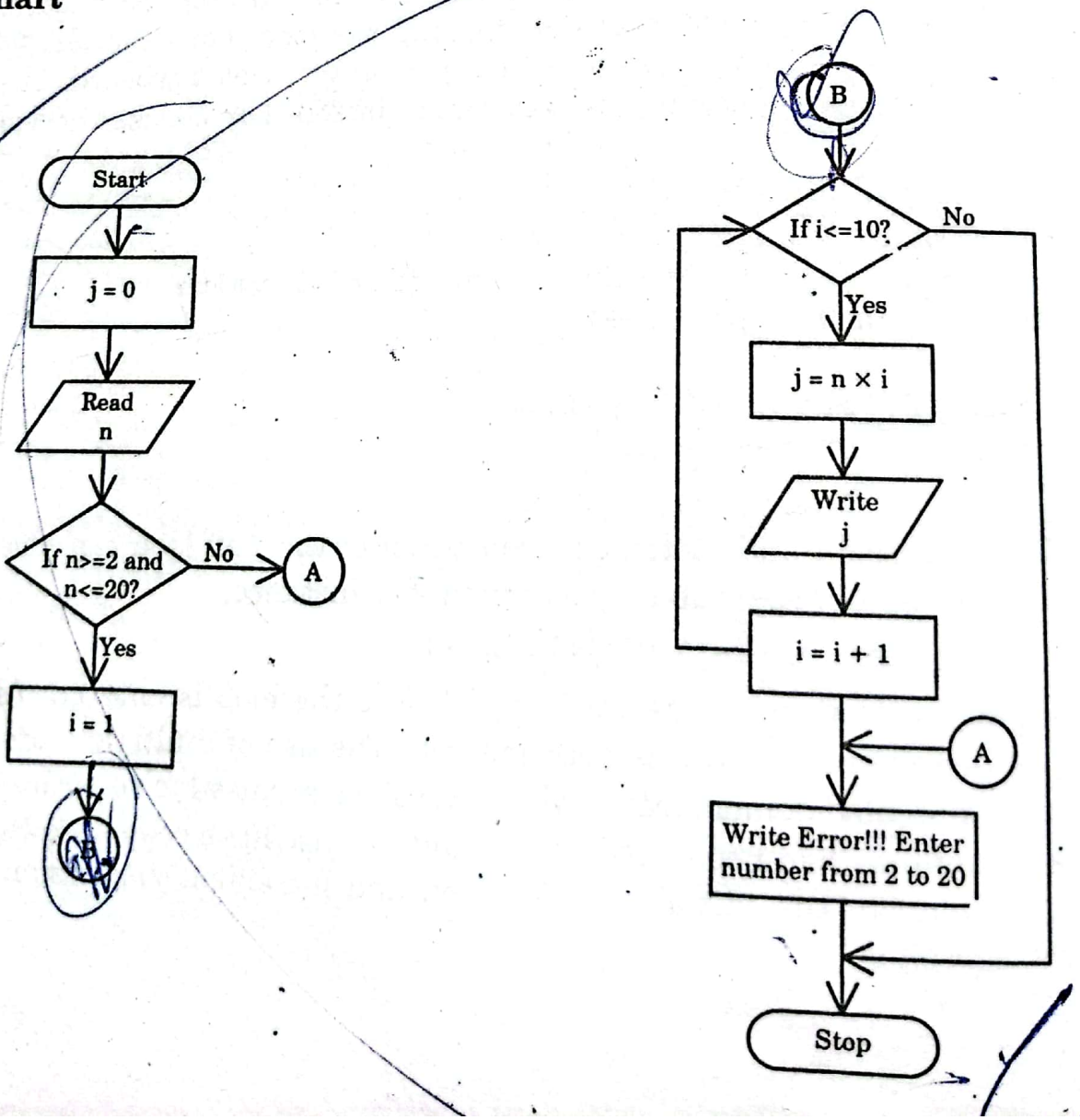
Object

Writing a Program which uses 'for' loop statement, (generate the multiplication tables from 2 to 20).

Algorithm

Step 1. Set j = 0.
Step 2. Read n.
Step 3. IF n >= 2 and n <= 20, then
Step 4. Repeat FOR i = 1 to 10
  a. Set j = n × i.
  b. Write j.
  [End of FOR loop.]
Step 5. ELSE
  Write Error!!! Enter number from 2 to 20.
  [End of IF-ELSE structure.]
Step 6. Exit.

Flowchart



Source Code

```
/* Practical 4: Writing a Program which uses 'for' loop
statement, (generate the multiplication tables from 2 to 20). */

#include<stdio.h>
#include<conio.h>
void main (void)
{
 int i,n,j=0;
 clrscr();
 printf("\n\t\tEnter Table No. ");
 scanf("%d",&n);

 if(n>=2 && n<=20)
    for(i=1;i<=10;i++)
       {
       j=n*i;
       printf("\n\t\t%d * %d = %d",n,i,j);
       }
 else
 printf("\nError!!! Enter number from 2 to 20");

 getch();
}
```

Output

```
Enter Table No. 7

        7 * 1 = 7
        7 * 2 = 14
        7 * 3 = 21
        7 * 4 = 28
        7 * 5 = 35
        7 * 6 = 42
        7 * 7 = 49
        7 * 8 = 56
        7 * 9 = 63
        7 * 10 = 70
```

CamScanner

Practical 7(a)

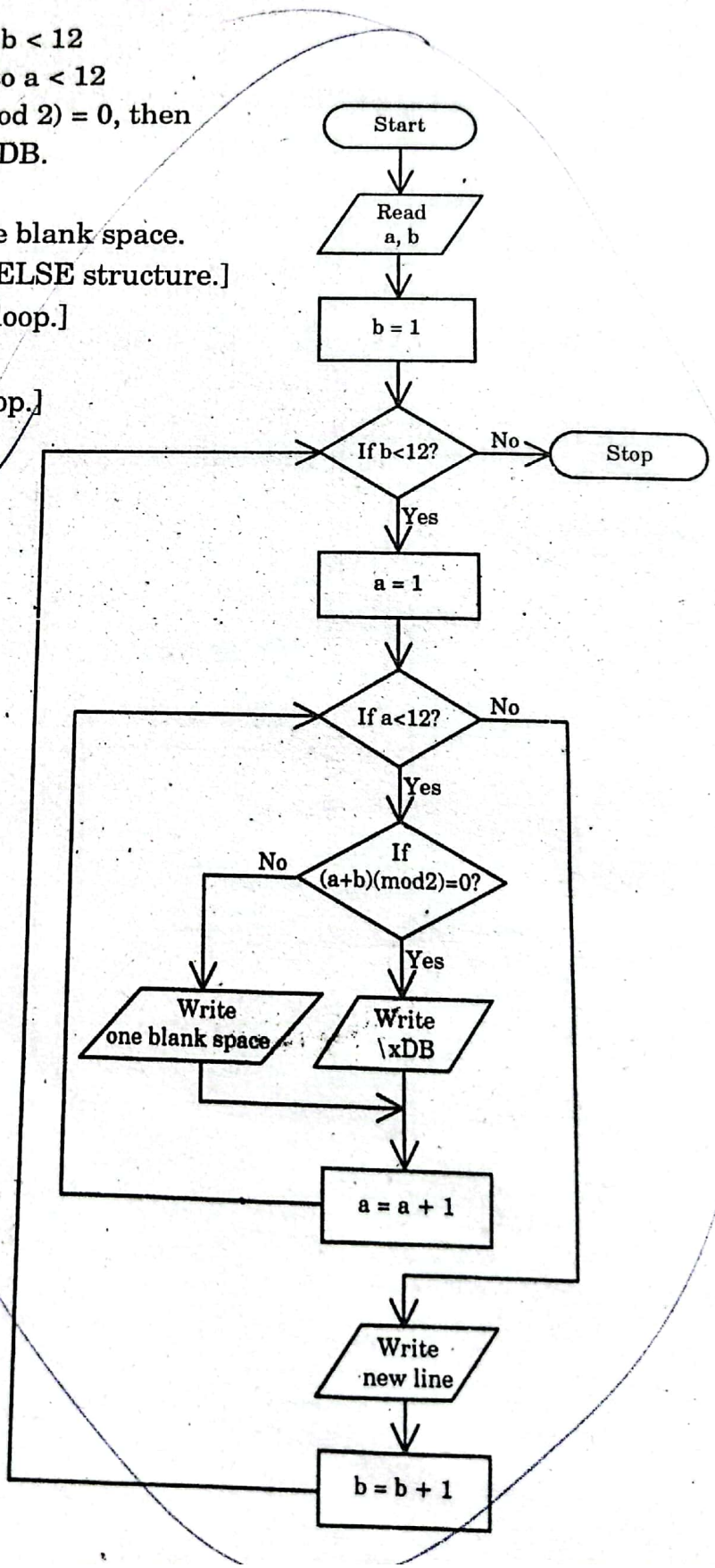
Object

Draw a Check-board and print it using if-else statement, and extend the program using nested if-else.

Algorithm

Step 1. Read a, b.
Step 2. Repeat FOR b = 1 to b < 12
Step 3. Repeat FOR a = 1 to a < 12
a. IF (a+b)(mod 2) = 0, then
Write \xDB.
b. ELSE
Write one blank space.
[End of IF-ELSE structure.]
[End of inner FOR loop.]
Write new line.
[End of outer FOR loop.]
Step 4. Exit.

Flowchart



CamScanner

Source Code

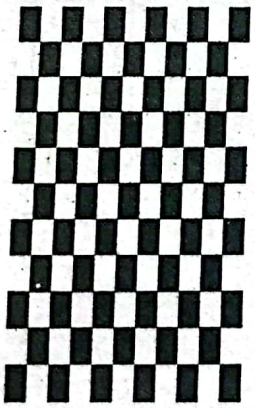
```
/* Practical 7(a): Draw a Check-board and print it using if-else
statement, and extend the program using nested if-else. */

#include<stdio.h>
#include<conio.h>
void main (void)
{
 clrscr();
 int a, b;

 for(b=1; b<12; b++)              /* stepping down */
    {
    for(a=1; a<12; a++)           /* stepping right */
       if((a+b)%2 == 0)           /* even numbered rectangle */
        printf("\xDB");           /* print filled rectangle */
       else
        printf(" ");              /* print blank rectangle */
    printf("\n");                 /* new line */
    }

 getch();
}
```

Output



Practical 7(b)

Object

Extend the Check-board using nested if-else statement.

Algorithm

Step 1. Read a, b.

Step 2. Repeat FOR b = 1 to b < 12

Step 3. Repeat FOR a = 1 to a < 12

a. IF a = b, then

Write \xB0.

CamScanner

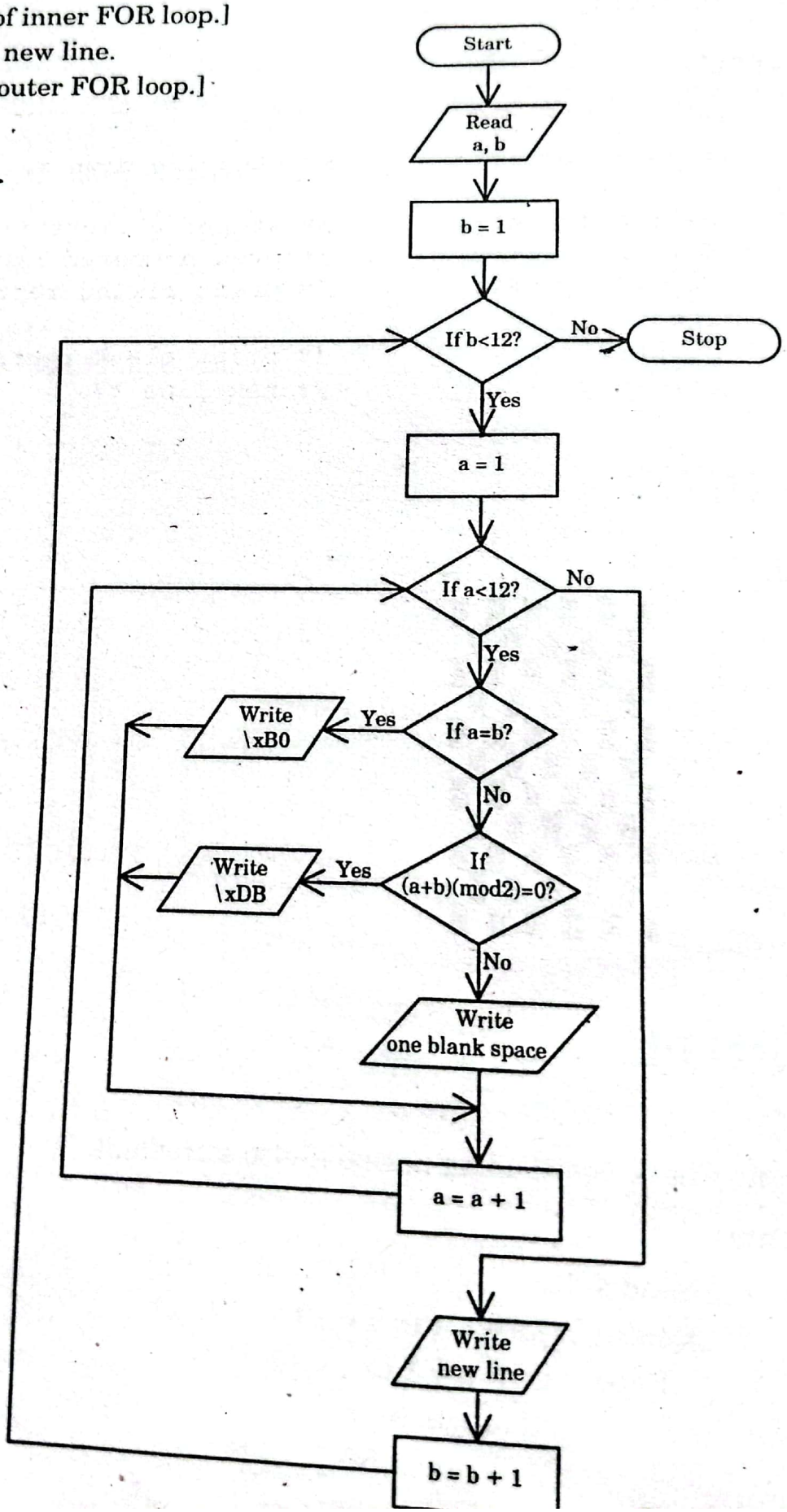
b. ELSE IF (a+b)(mod 2) = 0, then
Write \xDB.

c. ELSE
Write one blank space.
[End of IF-ELSE-IF structure.]

[End of inner FOR loop.]
Write new line.
[End of outer FOR loop.]

Step 4. Exit.

Flowchart



CamScanner

Source Code

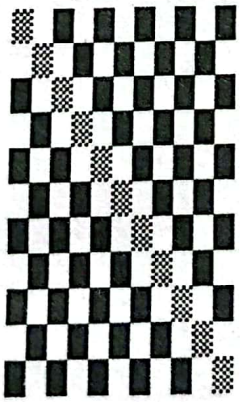
```
/* Practical 7(b): Extend the Check-board using nested if-else
statement. */

#include<stdio.h>
#include<conio.h>
void main (void)
{
 clrscr();
 int a, b;

 for(b=1; b<12; b++)                    /* stepping down */
    {
    for(a=1; a<12; a++)                 /* stepping right */
       if(a==b)                         /* left top diagonal */
         printf("\xB0");                /* print grey */
       else if((a+b)%2==0)
         printf("\xDB");                /* print filled rectangle */
       else
         printf(" ");                   /* print blank rectangle */
    printf("\n");                       /* new line */
    }

 getch();
}
```

Output



The while Loop

The while loop allows the programmer to specify that an action is to be repeated while some condition remains true. It is used when the exact number of loop repetitions before the loop execution begins are not known. The general form of the while statement is:

```
while(condition)
   statement;
```

In other words, as long as the condition is true, the statement continues to execute. More specifically, the *condition* is evaluated and if it is false, the structure is skipped. If the condition is true, the statement executes. The statement can be simple or

CamScanner

compound enclosed in braces. Once the statement executes, control is automatically returned to the condition, which is then re-evaluated. If it is false, the loop is terminated; otherwise the entire structure is repeated. This is diagrammed in Fig. 8.4.

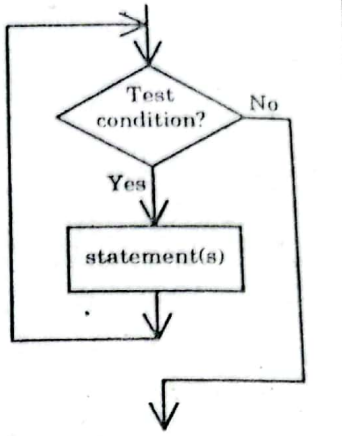


Fig. 8.4. Flowchart format of while loop

Consider the example:

```
/* program 8-12.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  char ch;
  printf("\nEnter a character: ");
  ch = getche();

  while(ch != 'q')
  {
    printf("\nInput character = %c", ch);
    printf("\n\nEnter a character: ");
    ch = getche();
  }
}
```

As long as the user does not type q, this while loop continues to output the input character (*echo* the input) and request another input character.

As another example, this program uses a while loop to print numbers from 1 to 10.

```
/* program 8-13.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  int count = 1;

 while (count <= 10)
 {
   printf("%3d", count);
   ++count;
}
getch();
```

his program can be written more concisely by replacing printf and increment atements with one as:

```
while (count <= 10)
  printf("%3d", count++);
```

CamScanner

Now look at this simple example, which takes an input number and displays an equivalent number of asterisk.

```
/* program 8-14.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  int num;
  printf("\n\n\n\t\tEnter a number: ");
  scanf("%d", &num);

  while(num != 0)
  {
    printf("*");
    num--;
  }
  printf("\n");
  getch();
}
```

To insure proper input, place the first input command outside the loop. You can then continue asking for input as long as the value is invalid, like this:

```
/* program 8-15.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  float num;
  printf("\n\nEnter an input number: ");
  scanf("%f", &num);

  while(num < 0 || num >=1)
  {
    printf("\n\nYou got it wrong");
    printf("\n\nEnter a character: ");
    scanf("%f", &num);
  }
}
```

Example 8.4

Write a program which uses while loop to print out a table of ASCII codes from 32 to 127 (compare it with program 8-10).

```
/* program 8-16.cpp */
#include<stdio.h>
#include<conio.h>
```

CamScanner

```
void main(void)
{
  clrscr();
  int i;

  i = 32;                                  /* initialisation to 32 */
  while(i < 128)                           /* code from 32 to 127 */
    {
     printf("%3d = %c\t", i, i);           /*print as number and as char*/
     i++;                                  /* increment */
     }
  getch();
}
```

Example 8.5

Write a program to calculate average of a list of numbers using while loop.

```
/* program 8-17.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  int n, count = 1;
  float x, average, sum = 0;
// initialise and read in a value for n
  printf("How many numbers? ");
  scanf("%d", &n);
// read in the numbers
  while(count <= n)
    {
      printf("X = ");
      scanf("%f", &x);
      sum += x;
      ++count;
    }
// calculate the average and print the answer
  average = sum/n;
  printf("\nThe average is %f", average);

  getch();
}
```

The Relationship of the for and while Loops

In many other languages the for loop and the while loop are very different constructs that are used for different purposes. In C the two loops are simply two ways of doing the same thing. The while structure can be used to increment through a loop much like a for loop. For example, instead of

CamScanner

```
for(i = 1; i <= n; i++)
   print i;
```

you could write

```
i = 1;
while(i++ <= n)
   print i;
```

The decision to use a for loop or a while loop is not determined by C but by conventional usage, readability, and consistency.

Practical 6

Object

Finding the Factorial of n using 'while' loop, read value of n using scanf, and print the factorial of n.

Algorithm

Step 1. Set f = 1 and i = 1.

Step 2. Read num.

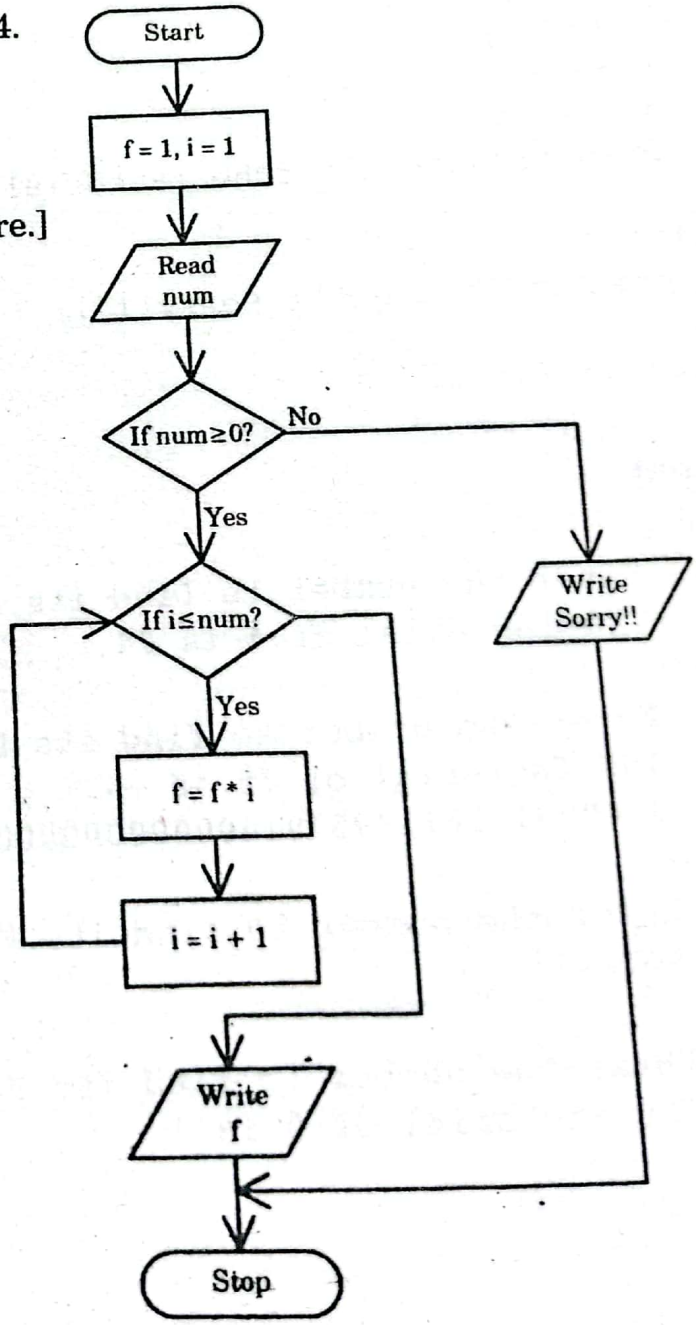
Step 3. IF num ≥ 0 then goto Step 4.
ELSE
Write Sorry!!
goto Step 6.
[End of IF-ELSE structure.]

Step 4. Repeat WHILE i ≤ num
a. f = f * i.
b. i = i + 1.
[End of WHILE loop.]

Step 5. Write f.

Step 6. Exit.

Flowchart



CamScanner

Source Code

```
/* Practical 6: Finding the Factorial of n using 'while' loop,
read value of n using scanf, and print the factorial of n.*/

#include<stdio.h>
#include<conio.h>
void main(void)
{
clrscr();
double f=1;
int num, i;

printf("\n\n\n\t\tEnter the number to find its Factorial: ");
scanf("%d", &num);

if (num>=0)
{
   i = 1;
   while (i<=num)
   {
     f=f*i;
     i++;
   }
   printf("\n\n\n\t\tThe factorial of %d is %lf", num, f);
}
else
   printf("\n\n\n\t\t Sorry!!");

getch();
}
```

Output

1st Run

```
Enter the number to find its Factorial: 4
The factorial of 4 is 24
```

2nd Run

```
Enter the number to find its Factorial: 75
The factorial of 75 is
2.480914081139539140000000000000000000000e+109
```

3rd Run

```
Enter the number to find its Factorial: -7
Sorry!!
```

4th Run

```
Enter the number to find its Factorial: 0
The factorial of 0 is 1
```

CamScanner

Nested while Loops

The while loop can be nested just like for loop. The following program shows a while loop nested within another while loop to read a number and print its factorial (compare this program with program 8-11).

```
/* program 8-18.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  double f;
  int num;
  // computes the product n*(n-1)*(n-2)*(n-3)*...*2*1
  while(num != 0)
  {
   printf("\n\n\n\t\tEnter the number to find its Factorial: ");
   scanf("%d", &num);
     f = 1;
     while(num > 1)
        {
         f = f * num;
         num--;
        }
   printf("\t\tThe factorial is %lf", f);
  }
}
```

This program uses an outer while loop to recycle until a 0 value is entered. The inner loop uses the decrement operator to reduce the variable num—which starts from the value typed in by the user—by 1 each time through the loop. When num reaches 1, the loop terminates.

The do-while Loop

One important characteristic of the while statement is that the test condition is at the top of the loop. This means that if the condition is false the *first* time, the body of while loop will never be executed. But there are certain situations where you need to execute the body at least once. These situations are not common, but when they do occur, you should use the do-while statement.

The only difference between a do-while and a while loop is that the test condition is at the bottom of the loop. This means that the program always executes the statement(s) at least once, even if the condition fails for the first time itself. Then, depending on the value of test expression, it may loop back to do or it may continue with the next statement. The general form of the do-while statement is:

```
do
   statement;
while(condition);
```

CamScanner

The statement will be performed once even if the condition is false when the loop first begins because the condition is not tested at the top. Then the statement is performed repeatedly as long as the condition in the while statement is true. The statement can be simple or compound enclosed in braces. The do-while loop statement can be visualised using flowchart as shown in Fig. 8.5. This flowchart makes it clear that the test condition is not executed until after the action is preformed at least once.

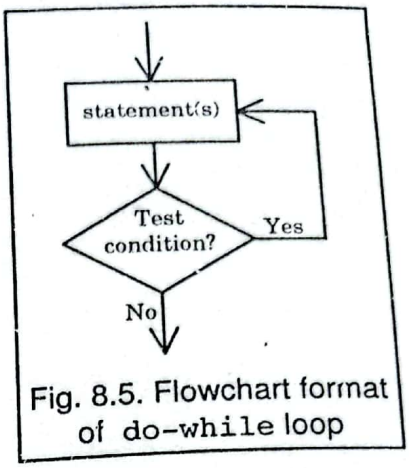


Fig. 8.5. Flowchart format of do-while loop

For example, we could print numbers from 1 to 10 using C code (compare it with program 8-13):

```
/* program 8-19.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  int count = 1;
  do
    {
     printf("%3d", count);
    }
  while(++count <= 10);

  getch();
}
```

The do-while structure is often used to repeat a program until the user decides there is no more input:

```
/* program 8-19.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  int num;
  char repeat;
  do
  {
   printf("\n\nPlease enter a number: ");
   scanf("%d", &num);
   printf("The square of your number is = %d", num*num);
   printf("\n\nDo you wish to continue y/n: ");
   repeat = getche();
   }
  while(repeat == 'y' || repeat == 'Y');
}
```

CamScanner

The entire program, except for the variable declarations enclosed in one large do-while block. The loop continues until you enter a character other than y or Y in response to the prompt.

The do-while structure is also used to insure that the proper input is entered. For example, suppose you write a program that inputs a discount rate in decimal format. If the user enters a rate less than 0, or 1 or above, you want to ask for another input. The do-while loop looks like this:

```
do
{
  printf("Please enter the discount: ");
  scanf("%f", &discount);
}
while(discount < 0 || discount >= 1);
```

The while condition uses the logical OR operator to test whether the value entered is outside of the range. The loop will repeat as long as the user enters an invalid number.

Example 8.6

The do-while loop is especially useful when your program is waiting for some event to occur. For example, this program waits for the user to type a q.

```
/* program 8-21.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  char ch;
  do
    {
     ch = getche();
    }
  while(ch != 'q');
  printf("\n\nFound the q");

  getch();
}
```

Since the loop condition is tested at the bottom, it is not necessary to initialise ch prior to entering the loop.

Example 8.7

Modify the program 8-17 to calculate average of a list of numbers using do-while loop.

```
/* program 8-22.cpp */
#include<stdio.h>
#include<conio.h>
```

CamScanner

```
void main(void)
{
  clrscr();
  int n, count = 1;
  float x, average, sum = 0;

// initialise and read in a value for n
  printf("How many numbers? ");
  scanf("%d", &n);

// read in the numbers
  do
    {
      printf("X = ");
      scanf("%f", &x);
      sum += x;
      ++count;
    }
  while(count <= n);

// calculate the average and print the answer
  average = sum/n;
  printf("\nThe average is %f", average);

  getch();
  }
```

Nested do-while Loops

You can nest do-while loops to provide two levels of repetition. You can use an outer loop to repeat the entire program until the user decides otherwise, and inner loops to take a series of inputs, as shown in program 8-23. The inner loop is used to calculate average of a list of numbers. The outer loop is repeated as long as this condition is true:

```
while(repeat == 'y' || repeat == 'Y');
```

When you enter any other character than y or Y, the outer loop, and thus the program, ends. (Compare this program with program 8-22 and note the changed position of initialisation. Why it is necessary to make this change?)

```
/* program 8-23.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
 clrscr();
 int n, count;
 float x, average, sum;
 char repeat;
```

CamScanner

```
// initialise and read in a value for n
  do
  {
   printf("How many numbers? ");
   scanf("%d", &n);
   sum = 0;
   count = 1;

// read in the numbers
    do
      {
      printf("X = ");
      scanf("%f", &x);
      sum += x;
      ++count;
      }
      while(count <= n);

// calculate the average and print the answer
   average = sum/n;
   printf("\nThe average is %f", average);
   printf("\n\nDo you wish to continue y/n: ");
   repeat = getche();
   printf("\n\n");
  }
  while(repeat == 'y' || repeat == 'Y');
}
```

Inner loop takes a series of inputs

Outer loop repeats the entire program

Combining Loop Types

You can nest any combination of `for`, `while`, and `do-while` loops if the program logic requires it. The following program, for instance, nests a `while` loop within a `for` repetition to convert 10 temperatures, between 0 and 100, input from the keyboard:

```
/* program 6-24.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
clrscr();
int temp, count;
float celsius;
for(count=1; count<=10; count++)
   {
    printf("Enter a temperature between 1 and 100: ");
    scanf("%d", &temp);
    while(temp < 0 || temp > 100)
```

CamScanner

```
        {
         printf("Invalid temperature, try again: ");
         scanf("%d", &temp);
        }
      celsius = (5.0/9.0)*(temp-32);
      printf("%d fahrenheit is %6.2f celsius\n", temp, celsius);
     }
  getch();
  }
```

The outer for loop will only be repeated 10 times. The inner while loop will be repeated as long as you enter a valid entry.

The break Statement

The break statement allows you to exit a loop or a switch from any point within its body, bypassing its normal termination expression. It can be used within any control structure *e.g.* for, while, do-while, or switch statement.

The general form of the break statement is

```
loop(condition)
    {
      …
      if(…)
      break;
      …
    }
```

When the break statement is encountered inside a loop, the loop is immediately terminated, and program control resumes at the next statement following the loop. But if a break statement is found in nested loops, the break only transfers control to the outside of the loop actually containing the break.

Program 8-25 demonstrates the break statement in a for loop. When the structure detects that count has become 10, break is executed. This terminates the for statement and loop only fully executes 10 times.

```
/* program 8-25.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  int count;
  for(count = 1; count <= 100; count++)
     {
       printf("%3d", count);
       if(count==10)
       break;                    /* exit the loop */
     }
  getch();
}
```

CamScanner

The break statement is commonly used in loops in which a special condition can cause immediate termination. This is an example of such a situation, where a key press can stop the execution of the program.

```
/* program 8-26.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  int i;
  char ch;
  // display all numbers which are multiples of 6
  for(i = 1; i < 10000; i++)
     {
       if(!(i%6))
       {
       printf("%d more (y/n) ", i);
       ch = getche();
       if(ch=='n')
       break;
       printf("\n");
       }
     }
}
```

Example 8.8

A break will cause an exit from only the innermost loop. For example, this program prints the numbers 1 to 5 five times.

```
/* program 8-27.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  int i, j;

  for(i = 0; i < 5; i++)
    {
     for(j = 0; j < 100; j++)
       {
      printf("%3d", j);
      if(j==5)
      break;
       }
       printf("\n");
    }
  getch();
}
```

CamScanner

The continue Statement

The continue statement is just the opposite of the break statement. It takes the control to the beginning of the loop, bypassing the statements inside the loop which have not yet been executed.

The general form of the continue statement is

```
loop(condition)
    {
      …
      if(…)
      continue;
      …
    }
```

In while and do-while structure, the loop-continuation test is evaluated immediately after the continue statement is executed. In the for structure, the increment expression is executed, then the loop-continuation test is evaluated. The continue statement is not used with the switch statement.

Program 8-28 uses the continue statement in a for structure to skip the printf statement in the structure if count is equal to 5 and begin the next iteration of the loop.

```
/* program 8-28.cpp */
#include<stdio.h>
#include<conio.h>

void main(void)
{
 clrscr();
 int count;
 for(count = 1; count <= 10; count++)
    {
      if(count==5)
      continue;          /* skip remining code in loop if x==5 */
      printf("%3d", count);
    }

 getch();
}
```

The output of the program will be

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

The program used continue to skip printing the value 5.

CamScanner

Practical 5(a)

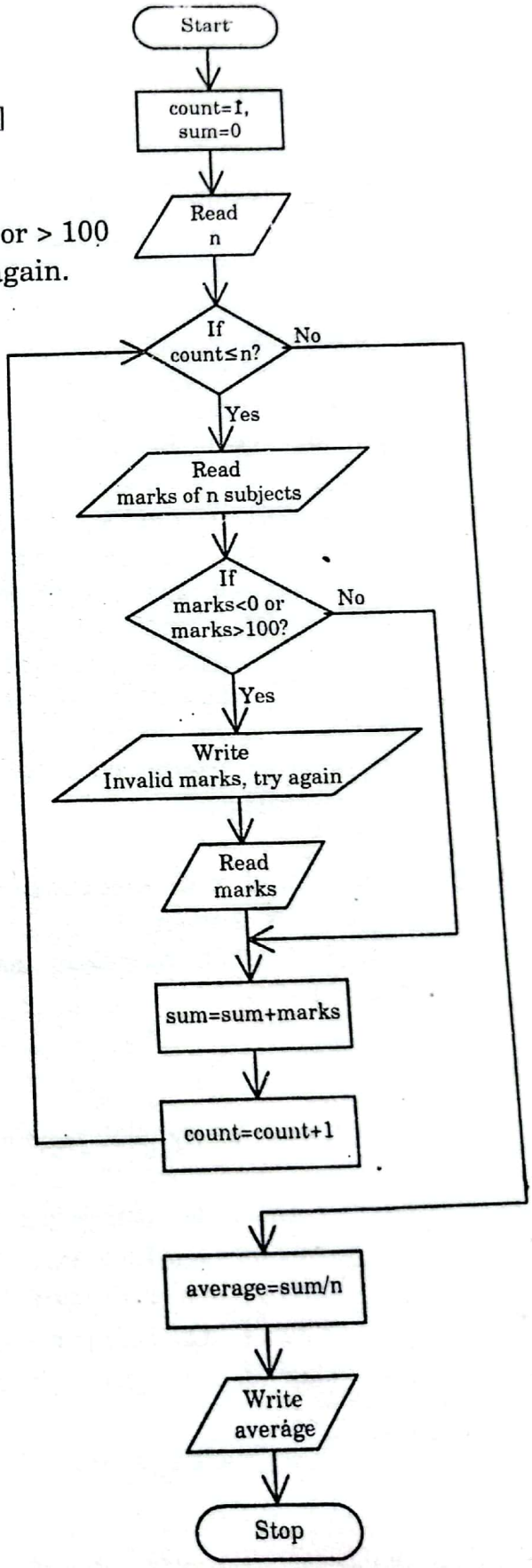
Object

Writing a program which uses 'while' loop and nested while loop, (use 'for' loop and continue the process in 'while' loop satisfying this condition *i.e.* write a program to calculate average marks of a student using while and nested while loop.

Algorithm

Step 1. Set count = 1 and sum = 0.
Step 2. Read n. [number of subjects.]
Step 3. Repeat WHILE count ≤ n
  a. Read marks of n subjects.
  b. Repeat WHILE marks are < 0 or > 100
    i. Write Invalid marks, try again.
    ii. Read marks.
    [End of inner WHILE loop.]
  c. sum = sum + marks.
  d. count = count + 1.
  [End of outer WHILE loop.]
Step 4. average = sum/n.
Step 5. Write average.
Step 6. Exit.

Flowchart



CamScanner

Source Code

```
/* Practical 5(a): Writing a program which uses 'while' loop and
nested while loop, (use 'for' loop and continue the process in
'while' loop satisfying this condition).

Write a program to calculate average marks of a student using
while and nested while loop. */

#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  int n, count = 1;
  float marks, average, sum = 0;
// initialise and read in a value for n
  printf("How many Subjects? ");
  scanf("%d", &n);
  printf("\n");
// read marks
  while(count <= n)
      {
      printf("Marks in Subject %d = ", count);
      scanf("%f", &marks);
      while(marks < 0 || marks > 100)
          {
            printf("Invalid marks, try again: ");
            scanf("%f", &marks);
          }
     sum += marks;
     ++count;
     }
// calculate the average and print the answer
  average = sum/n;
  printf("\nThe average marks are = %f", average);
  getch();
}
```

Output

```
How many Subjects? 5

Marks in Subject 1 = 69
Marks in Subject 2 = 58.5
Marks in Subject 3 = 37
Marks in Subject 4 = 83
Marks in Subject 5 = 65

The average marks are = 62.500000
```

CamScanner

Practical 5(b)

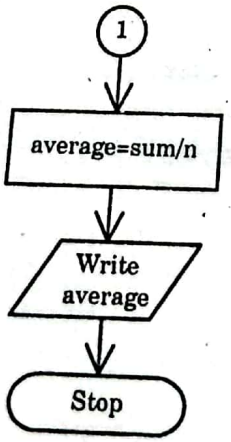
Object

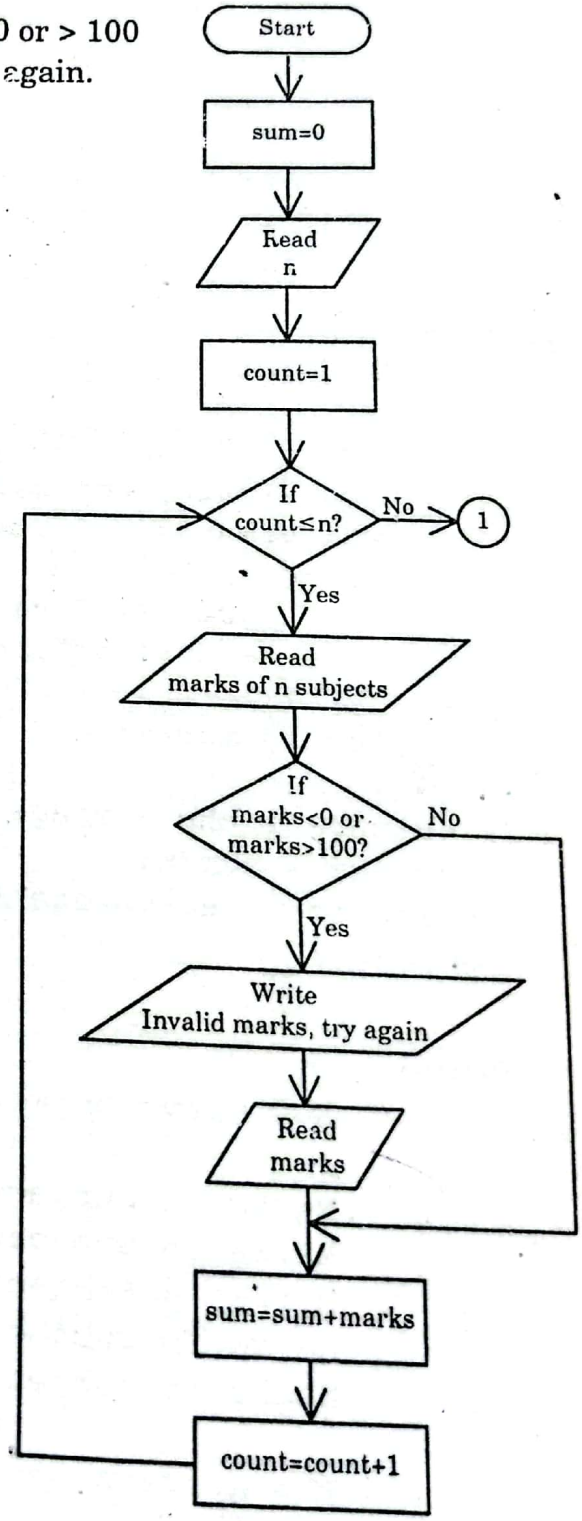
Writing a program which uses 'while' loop and nested while loop, (use 'for' loop and continue the process in 'while' loop satisfying this condition *i.e.* write a program to calculate average marks of a student using for and nested while loop.

Algorithm

Step 1. Set sum = 0.
Step 2. Read n. [number of subjects.]
Step 3. Repeat FOR count = 1 to n
  a. Read marks of n subjects.
  b. Repeat WHILE marks are < 0 or > 100
    i. Write Invalid marks, try again.
    ii. Read marks.
    [End of WHILE loop.]
  c. sum = sum + marks.
  [End of FOR loop.]
Step 4. average = sum/n.
Step 5. Write average.
Step 6. Exit.

Flowchart





CamScanner

Source Code

```
/* Practical 5(b): Writing a program which uses 'while' loop and
nested while loop, (use 'for' loop and continue the process in
'while' loop satisfying this condition).

Write a program to calculate average marks of a student using for
and nested while loop. */

#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  int n, count;
  float marks, average, sum = 0;
// initialise and read in a value for n
  printf("How many Subjects? ");
  scanf("%d", &n);
  printf("\n");
// read marks
  for(count =1; count <= n; count++)
      {
      printf("Marks in Subject %d = ", count);
      scanf("%f", &marks);
      while(marks < 0 || marks > 100)
          {
            printf("Invalid marks, try again: ");
            scanf("%f", &marks);
          }
      sum += marks;
      }
// calculate the average and print the answer
  average = sum/n;
  printf("\nThe average marks are = %f", average);
  getch();
}
```

Output

```
How many Subjects? 5

Marks in Subject 1 = 69
Marks in Subject 2 = 58.5
Marks in Subject 3 = 37
Marks in Subject 4 = 83
Marks in Subject 5 = 65

The average marks are = 62.500000
```

CamScanner

Exercise

8.1. What is meant by looping?

8.2. What is the purpose of the `for` statement? How does it differ from the `while` statement and the `do-while` statement?

8.3. How many times will a `for` loop be executed? Compare with the `while` loop and the `do-while` loop.

8.4. How would you use a compound assignment operator to count by 2?

Ans.
```
for(count = 1; count <= 21; count += 2)
```

8.5. Write a `for` loop that counts backwards from 100 to 0 by 10.

Ans.
```
for(count = 100; count >= 0; count -= 10)
```

8.6. Find the error in the following statement (Note: there may be more than one error).
```
for(i = 100; i >= 1; i++)
```

8.7. What are the functions performed in a `for` argument?

8.8. What stops a `for` loop?

8.9. What is a nested loop?

8.10. Write a program that displays the multiplication table for numbers 0 to 9.

8.11. Write nests of loops that cause the following output to be displayed:
```
* * * * *
* * * *
* * *
* *
*
* *
* * *
* * * *
* * * * *
```

8.12. What is displayed by the following program segment, assuming `m` is 3 and `n` is 5?
```
for(i = n; i > 0; --i)
   {
      for(j = m; j > 0; --j)
         {
           printf("*");
         }
         printf("\n");
   }
```

8.13. During execution of the following program segment, how many lines of asterisks are displayed?
```
for(i = 0; i < 10; ++i)
   for(j = 0; j < 5; ++j)
      printf("**********\n");
```

Ans. 50.

CamScanner

8.14. Compare and discuss program 8-29 with one given in practical 7(b). Discuss the similarities and differences.

```
/* program 8-29.cpp */
#include<stdio.h>
#include<conio.h>
void main (void)
{
 clrscr();
 int a, b;

 for(b=1; b<12; b++)                    /* stepping down */
    {
    for(a=1; a<12; a++)                 /* stepping right */
       if((a+b)%2==0 && a!=b)
     printf("\xDB");                    /* print filled square */
       else if(a==b)                    /* left top diagonal */
     printf("\xB0");                    /* print gray */
       else
     printf(" ");                       /* print blank square */
    printf("\n");                       /* new line */
    }

 getch();
}
```

8.15. Explain what is wrong with this program:

```
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  float row, column;
  puts("\t\tMy Handy Multiplication Table\n");
  for(row = 1; row <= 10; row++)
  {
      for (column = 1; column <= 10; column+)
      printf("%d", row*column);

      printf("\n");
   }

  getch();
}
```

8.16. What does the following program output?

```
/* program 8-30.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
```

CamScanner

```
{
  clrscr();
  int i, j, k;
  k = 0;
  for(i = 1; i <= 30; i++)
     for(j = 1; j <= 20; j++)
   if(j%3 == 0)
      k++;
  printf("\nk = %d", k);
  getch();
}
```

8.17. What will be the output of the following programs:

(a)
```
void main(void)
{
  char i = 1;
  while(i <= 212)
    {
     printf("%d\n", i);
     i = i + 1;
    }
}
```

(b)
```
void main(void)
{
  int i = 1;
  while(i <= 212)
     printf("%d\n", ++i);
}
```

(c)
```
void main(void)
{
  int i;
  for(i = 1; i <= 212; i++)
     printf("%d", i);
}
```

8.18. Write a program that inputs ten numbers between 0 and 25.

8.19. Write a program to print the sum of even numbers from 1 to 100.

8.20. Write a program to print the squares of all the numbers from 1 to 20.

8.21. Write a program to print prime numbers up to a specified range using `for` and nested `for` loop.

Ans. We can test whether a number is prime or not, by dividing it successively by all numbers from 2 to one less than itself. If remainder of any of these divisions is zero, the number is not a prime.

Following is the program to do this

```
/* program 8-31.cpp */
#include<stdio.h>
#include<conio.h>
```

CamScanner

```
void main(void)
{
  clrscr();
  int i, j, num;
  printf("Enter the number: ");
  scanf("%d", &num);
  for(i = 2; i <= num; i++)
     {
       for(j = 2; j < i; j++)
         {
          if(i%j==0)
          break;
         }
      if(i==j)
      printf("%d\t", i);
     }

  getch();
  }
```

8.22. What is the purpose of the `while` statement? When is the logical expression evaluated? What is the minimum number of times that a `while` loop can be executed?

8.23. How is the execution of a `while` loop terminated?

8.24. When are a `for` loop and a `while` loop the same?

8.25. What is the purpose of the `do-while` statement? How does it differ from the `while` statement?

8.26. What is the minimum number of times that a `do-while` loop can be executed? Compare with a `while` loop and explain the reasons for the differences.

8.27. What is the value of `i` after each of these segments?

(a)
```
while(i++ <= 10);
```

(b)
```
while(++i <= 10);
```

(c)
```
i = 0;
do
  ++i;
while(i++ < 10);
```

8.28. What criteria do you use to select a `for`, `while` or `do-while` loop?

8.29. State all the loop control structures supported by C and compare them.

8.30. What does the following code segment display? Try each of these inputs: 345, 82, 6. Then, describe the action of the code.

```
printf("\nEnter a positive integer: ");
scanf("%d", &num);
do
  {
   printf("%d", num%10);
   num /= 10;
  }
```

CamScanner

```
    while(num > 0);
    printf("\n");
```

Ans.
```
Enter a positive integer: 345
5  4  3
Enter a positive integer: 82
2  8
Enter a positive integer: 6
6
```

The code displays the digits of an integer in reverse order and separated by spaces.

8.31. Write a loop that will calculate the sum of every third integer, beginning with `i = 2` (*i.e.*, calculate the sum 2 + 5 + 8 + 11 + ...) for all values of `i` that are less than 100. Write the loop in three different ways.

(a) Using a `for` statement.

(b) Using a `while` statement.

(c) Using a `do-while` statement.

8.32. What is the purpose of the `break` command?

8.33. State the use of the following giving an example of each.

(a) break

(b) continue

CamScanner

Objectives and MCQs

Choose the answer that best completes the statement.

8.1. The main difference in operation between an `if` statement and a `while` statement is

(a) the conditional expression following the keyword is evaluated differently

(b) the `while` loop body is always executed, the `if` loop body only if the condition is true

(c) the body of the `while` statement may be executed many times, the body of the `if` statement only once

(d) the conditional expression is evaluated before the `while` loop body is executed but after the `if` loop body

8.2. Assuming `x` does not equal 0, the statement `while(x==0) printf("X=0")`

(a) is an infinite loop

(b) will never output

(c) causes a syntax error

(d) will output once

8.3. Assuming `x` does not equal 0, the statement `do printf("X=0") while(x==0)`

(a) is an infinite loop

(b) will never output

(c) causes a syntax error

(d) will output once

8.4. A single-statement `for` loop is terminated with a

(a) right bracket

(b) right brace

(c) comma

(d) semicolon

8.5. The statement `for(x = 0; x < 1; x++) printf("X=0")`

(a) is an infinite loop

(b) will never output

(c) causes a syntax error

(d) will output once

8.6. The statement `for(x = 0; x <= -1; x++) printf("X=0")`

(a) is an infinite loop

(b) will never output

(c) causes a syntax error

(d) will output once

CamScanner

8.7. A `while` loop is more appropriate than a `for` loop when
   (a) the terminating condition occurs unexpectedly
   (b) the body of the loop will be executed at least once
   (c) the program will be executed at least once
   (d) the number of times the loop will be executed is known before the loop is executed

8.8. A `do-while` loop is useful when
   (a) the body of the loop will never be executed
   (b) the body of the loop will be executed at least once
   (c) the body of the loop may never be executed
   (d) none of the above

8.9. The advantage of putting complete assignment statements inside loop expressions is
   (a) to avoid awkward program constructions
   (b) to simplify the flow of control in the program
   (c) to reduce the number of program statements
   (d) to clarify the operation of the program

8.10. The `break` statement is used to exit from which part of a loop?
   (a) beginning
   (b) middle
   (c) end
   (d) none of the above

Objectives

8.1. The `for` loop is generally used with a counter.

8.2. The `for` loop counter variables should not be of the `float` type or of the `char` type. They should only be of the `int` type.

8.3. The three parts of the `for` loop expression are (i)the initialise expression, (ii)the test expression, and the (iii)increment expression.

8.4. A semicolon is used to separate the three parts of the loop expression in a `for` loop.

8.5. The `while` and `do-while` loops are used for indefinite repetitions.

8.6. The `for` loop is used for definite repetition.

8.7. The `do-while` loop uses post testing.

8.8. A program that repeats 5 outer and 10 inner loops actually performs 50 repetitions.

8.9. The `continue` statement, when executed in a repetition structure causes the next iteration of the loop to be performed immediately.

8.10. The `break` statement, when executed in a repetition structure or a `switch` causes immediate exit from the structure.

CamScanner

Find all errors (syntax or logic) in each of the following program segments. The programs attempt to print the integers from 1 through 100.

8.1.
```
x = 1;
while(x < 100);
printf("X = %d", x);
x++;
```

8.2.
```
num = 100;
while(num > 1);
printf("Number = %d", num);
num++;
```

8.3.
```
for(x = 1; x <= 100; x++)
{
printf("X = %d", x);
x++;
}
```

8.4.
```
x = 1;
do
  printf("X = %d", x);
  x++;
while(x <= 100);
```

8.5.
```
x = 0;
do
{
  printf("X = %d", x);
  x++;
}
while(x < 100);
```

8.6.
```
for(x = 1; x <= 100; )
printf("X = %d", x);
x++;
```

CamScanner

CHAPTER 9

FUNCTIONS

One of the most fundamental ideas in problem solving is dividing a problem into sub-problems and solving each sub-problem independently. C facilitates this approach through the use of functions.

A function is a block of instructions that together perform a specific task. Every C program has at least one function, `main()`. You can also write your own functions and use them just as you use functions in a C library.

When your program starts, `main()` is called automatically and it is always executed first. `main()` might called other functions, some of which might called still others.

Advantages of Functions

The following are some advantages of using functions.

1. The complexity of the entire program can be divided into simple subtasks and function subprograms can be written for each subtask.
2. Functions help us to avoid unnecessary repetition of code. Using functions a single section of code can be used many times in the same program.
3. Functions can have inputs and outputs and can process information.
4. The functions or subprograms are short, easier to write, understand and debug.
5. A function can be shared by other programs by compiling it separately and loading them together.
6. Functions are mostly independent of each other, just as one program is independent of another program. Most of the activities in a function are hidden from the rest of the program.
7. In *Turbo C++*, a function can call itself again and again. It is called *recursiveness*. Many calculations can be done easily using recursive process such as calculation of factorial of a number etc.

CamScanner

Classification of Functions

Functions are of two types. These are:

1. Built-in Functions/Library Functions
2. User-Defined Functions

Built-in Functions

Many of the operations, like taking the square root of a number, sine value of a number etc., will be frequently used by many programmers in their programs. Writing a program to find the square root of a number or sine value of a number every time in each program is an unnecessary and lengthy job. Such operations are programmed and stored in *Turbo C++* library, so that they can be called through any program in the form of functions. These functions are called *library functions* or *built-in functions*. Table 9.1 shows some of the commonly used mathematical library functions.

Table 9.1. Some Mathematical Built-in Library Functions

Function	Library File	Purpose: Example	Argument(s)	Result
`abs(x)`	`<math.h>`	Returns the absolute value of its integer argument: if x is –5, `abs(x)` is 5	`int`	`int`
`ceil(x)`	`<math.h>`	Returns the smallest whole number that is not less than x: if x is 45.23, `ceil(x)` is 46.0	`double`	`double`
`cos(x)`	`<math.h>`	Returns the cosine of angle x: if x is 0.0, `cos(x)` is 1.0 (radians)	`double`	`double`
`exp(x)`	`<math.h>`	Returns e^x where e = 2.71828...: if x is 1.0, `exp(x)` is 2.71828	`double`	`double`
`fabs(x)`	`<math.h>`	Returns the absolute value of its type double argument: if x is –8.432, `fabs(x)` is 8.432	`double`	`double`
`floor(x)`	`<math.h>`	Returns the largest whole number that is not greater than x: if x is 45.23, `floor(x)` is 45.0	`double`	`double`
`log(x)`	`<math.h>`	Returns the natural logarithm of x for x>0.0: if x is 2.71828, `log(x)` is 1.0	`double`	`double`
$\log_{10}(x)$	`<math.h>`	Returns the base-10 logarithm of x for x>0.0: if x is 100.0, $\log_{10}(x)$ is 2.0	`double`	`double`
`pow(x,y)`	`<math.h>`	Returns x^y. If x is negative, y must be a whole number: if x is 0.16 and y is 0.5, `pow(x,y)` is 0.4	`double`	`double`
`sin(x)`	`<math.h>`	Returns the sine of angle x: if x is 1.5708, `sin(x)` is 1.0 (radians)	`double`	`double`
`sqrt(x)`	`<math.h>`	Returns the non-negative square root of x ($\sqrt{x}$) for x≥0.0: if x is 2.25, `sqrt(x)` is 1.5	`double`	`double`
`tan(x)`	`<math.h>`	Returns the tangent of angle x: if x is 0.0, `tan(x)` is 0.0 (radians)	`double`	`double`

CamScanner

User-Defined Functions

Apart from the library functions that are built in *Turbo C++*, users can also define functions to do a task relevant to their programs. Such function are called *user-defined functions*. These functions should be codified by the user, so that any call to the function can refer to it.

General Form of Function

The overall structure of the function looks almost exactly like that of the `main()` function in a program. That is because `main()` is also a function. The only difference is that `main` is a special name. Use of the function in a program requires three things:

1. Function declaration, which is also called function prototype.
2. Function definition.
3. Function calling.

Program 9-1 shows the form of a user-defined `display()` function.

```
/* program 9-1.cpp */
#include<stdio.h>
#include<conio.h>
int y=4;
void display(void);
void main(void)
{
  clrscr();
  display();
  printf("\nThe capital city is ISLAMABAD\n");
  display();
  printf("\nThe colour of its flag is Green and White\n");
  printf("Pakistan has %d provinces", y);
  getch();
}
void display(void)
{
  int x;
  for(x=0; x<y; x++)
    printf("*");
  printf("\n*****PAKISTAN*****\n");
  for(x=0; x<y; x++)
    printf("*");
}
```

global variable

function declaration or function prototype

function calling

return data type

function name

note: no semicolon at the end

function header

accept data type

local variable

body of function

function definition for display()

The declaration tells the compiler that at some later point we plan to present a function called `display()`. The first `void` in the function header means that `display()` does not return anything, and the second means that it takes no arguments. Note that the function header does not end with a semicolon. It is not a program statement, whose execution causes something to happen. Rather, it tells the compiler that a function is being defined.

CamScanner

The function definition continues with the *body of the function*: the program statements that do the work.

The variable `x` used in `display()` function is only known to `display()`. It is invisible to the `main()` function. If we use this variable in `main()` function without declaring it, the compiler will give an error message because `main()` would not know any thing about this variable.

This is very important to know while writing functions in C. Variables used in a function are unknown outside the function. Thus a *local variable* will be known to the function it is defined in, but not to others. A local variable used in this way in a function is known in C as an *automatic variable*, because it is automatically created when a function is called and destroyed when the function returns. On the other hand placing a variable *globally* (outside a function) allows any other function to call it.

1. Function Prototype

Using functions in your program requires that you first declare the function and then you define the function. No function can be called from any other function that has not first been declared. The declaration of a function is called its *prototype*. It is a line written *before* the `main()` function. The prototype tells the compiler in advance about some characteristics of a function used in the program.

The general structure of the function prototype is:

```
type function_name(type argument-1, type argument-2, ...);
```

It has three main components. These are:

1. Data type of the Function
2. Name of the Function
3. Arguments

Data type of the Function: The data type of the function is the type of the returned value. If the function does not return a value, the type is defined as `void`.

Name of the Function: The name of the function is any legal identifier followed by the parentheses without any spaces in between.

Arguments: The arguments come inside the parentheses, preceded by their types and separated by commas. If the function does not use any arguments, the word `void` is used inside the parentheses.

The function prototype and the function definition must agree exactly about the return type, the name, and the argument list. If they do not agree, the compiler will give an error message. Note, however, that the function prototype does not need to contain the names of the arguments, just their types.

2. Function Definition

The function definition is the function itself. The general structure of the function definition is:

```
type function_name(type argument-1, type argument-2, ...)
{
   body of function;
}
```

CamScanner

The function definition has two parts:

1. Function Header
2. Body of Function

Function Header: The function begins with a header which is exactly same as the function prototype except it must not be terminated with semicolon (;). Header specifies the name of the function. It also describes whether you are using a simple function or passing the arguments or returning the value from the function.

Body of Function: It is the main body of the function. All statements are written in this part of the program. The statements in the body of the function are written to perform a specific task.

3. Function Calling

A user defined function is called from the main program simply by using its name, including the parentheses which follow the name. The parentheses are necessary so the compiler knows you are referring to a function and not a variable that you forgot to declare.

Functions with or without Arguments

It would not be useful if all functions were as independent of each other as `main()`. Most functions depend on other functions to some extent. For example, it does not make sense to call the pow (*raise to the power*) function without specifying values for the base and the exponent. Moreover, pow is not useful unless it sends back the result of the computation. Therefore, we might say that pow depends on the caller to provide it with the two values (for the base and the exponent, respectively). Similarly, the caller depends on pow to return a meaningful result. These dependencies are sometimes called the *interface* between the caller and the called function.

Sending data to a function is called *passing arguments*—that is, sending variables, constants, or expression whose values are needed by the function. The arguments are the input data that the function needs to compute its result. This result is then returned for use by the caller.

Functions can be used in four variations:

1. Function that does not need any argument and also returns no value.
2. Function that does not need any argument but returns a value.
3. Function that needs one or more arguments but returns no value.
4. Function that needs one or more arguments and also returns a value.

Some examples of prototypes of user-defined functions are:

```
void display(void);
int average(void);
void subset(int x);
float big_power(double x, double y);
```

The first function neither returns nor accept any arguments. Such type of function is called *simple function*. The second function returns an integer value and does not

CamScanner

accept any arguments. The third function returns no value but it accepts an argument x of the type integer. The fourth function accepts two double arguments (the value of x and the value of y) and returns a value of the type float.

Notice that in the function definition at the beginning of the function itself, the semicolon is not used at the end. Consider the program:

```
/* program 9-2.cpp */
#include<stdio.h>
#include<conio.h>

void two_beeps(void);              /* function prototype */
void main(void)
{
  clrscr();
  printf("My computer will beep\n");
                                   /* function calling */
  two_beeps();
  getch();
  printf("My computer will beep\n");
                                   /* function calling */
  two_beeps();
  getch();
  printf("My computer will beep\n");
                                   /* function calling */
  two_beeps();
  getch();
}
/*This function will make a beep sound twice*/
void two_beeps(void)               /* function header */
{
  int x;
  for(x=0; x<2; x++)
    printf("\a");                  /* printf("\x7"); */
}
```

This program first prints *My computer will beep* and then calls a function, `two_beeps()`, which sounds two beeps. Then if you strike a key, the process will be repeated twice. The function is called three times from `main()`. Each of the three calls looks like this:

```
two_beeps();
```

This is all we need to call a function: the function name, followed by parentheses.

Passing Arguments to a Function

You can pass an argument to your own functions. Any data you want to send to the function must be in the parentheses. You can pass more than one argument, as long as the number of arguments in the function call, and their data types, match those expected by the function. The list of variables being passed to a function is called an *argument list.*

Now let's see how to pass an argument to your own custom function. Consider this program:

CamScanner

```
/* program 9-3.cpp */
#include<stdio.h>
#include<conio.h>

void table(int num);
void main(void)
{
  clrscr();
  int num1, num2;

  printf("\nEnter Table No. ");
  scanf("%d", &num1);
  table(num1);
  printf("\nEnter Table No. ");
  scanf("%d", &num2);
  table(num2);
  getch();
}

void table(int num)
{
  int i;
  printf("\n***Table of %2d***", num);
  for(i=1; i<=10; i++)
     printf("\n%2d * %2d = %3d", num,i,num*i);
}
```

The function `table()` is called twice in the `main()`, first with the argument num1 and second time with the argument num2. The `table()` function receives the value of the argument into a variable named num each time. The variable num will have the same value as the variable num1 for the first call and equal to the mum2 for the second call. Also notice that the variable num is declared as `int` in braces. It tells the compiler what type of value will be received by the function. If the function needs any other variables, they must be declared after the brace as usual.

Here is what happens in the program:

1. The `table()` function is called and is passed the value of num1.
2. Suppose the value of 5 is received by the argument named num in the function.
3. The program will print the table of 5.
4. The process will be repeated second time for variable num2.

Passing Multiple Arguments to a Function

You can pass as many items to the function as you need, of any type. The function `area()` calculates the area of a floor. The length and width of the floor, and the floor's number, are input in `main()`, then passed to the function in the call

```
area(length, width, fnum);
```

CamScanner

Here is the program:

```
/* program 9-4.cpp */
#include<stdio.h>
#include<conio.h>

void area(float size, float wide, int num);
void main(void)
{
  clrscr();
  float length, width;
  int fnum;
  printf('\nEnter the floor number: ");
  scanf("%d", &fnum);
  printf("\nEnter the length of the floor: ");
  scanf("%f", &length);
  printf("\nEnter the width of the floor: ");
  scanf("%f", &width);
  area(length, width, fnum); //length, width and fnum are actual arguments

  getch();
}
void area(float size, float wide, int num)
                        //size, wide and num are formal arguments
{
  float area;
  area = size * wide;
  printf("\nThe area of floor %d is: %.2f", num, area);
}
```

The output from the program will be as:

```
Enter the floor number: 4
Enter the length of the floor: 17.4
Enter the width of the floor: 21.7
The area of floor 4 is: 377.58
```

In the above example, the function call `area()` contain arguments `length`, `width` and `fnum` which are different from the corresponding argument names in the function header (which are `size`, `wide` and `num`). This is not a C requirement and in fact we sometimes use same names. But the complier would still treat them as different variables since they are in different functions.

The variables `length`, `width` and `fnum` are called *actual arguments*, whereas the variables `size`, `wide` and `num` are called *formal arguments*. Any number of arguments can be passed to a function being called. However, the type, order and number of actual and formal arguments must always be same.

In program 9-4 notice that three arguments are received in the same order as they are passed. In this case, the value of `length` is received by `size`, the value of `width` is received by `wide`, and the value of `fnum` is received by `num`. The argument types match—two `floats` are received by two `floats`, and an `int` is received by an `int`.

CamScanner

The function `area()` also needs a variable to store the results of the calculation. The variable, also named `area`, is declared after the opening brace.

If you made a mistake and called the function like this:

```
area(width, length, fnum);
```

the value of `width` would be received by `size`, and the value of `length` would be received by `wide`. Because the types match, your compiler would not generate an error, and in this case the program would still generate the correct result—the order in which you multiply the numbers does not matter. But suppose you called the function using this argument list:

```
area(fnum, width, length);
```

The function would receive `fnum` into the argument `size`; `width` into `wide`, and `length` into num. Two of the arguments do not match in type. If your compiler did not detect an error, the results would certainly be incorrect.

Function that Return a Value

A function can receive values and return them as well. You call the function, it gets a certain piece of information and returns it to you. The function `getche()` operates in the same way; you call it—without giving it any information—and it returns the value of the first character typed on the keyboard.

Your own functions return a value or return `void`. As stated earlier that `void` is a signal to the complier that no value will be returned. To return a value from a function, write the keyword `return()` followed by the value you want to return. The `return()` statement has two purposes:

1. Executing it immediately and transferring control from the function back to the calling program.
2. Whatever is inside the parentheses following return is returned as a value to the calling program.

Fig. 9.1 shows a function returning a value to the calling program.

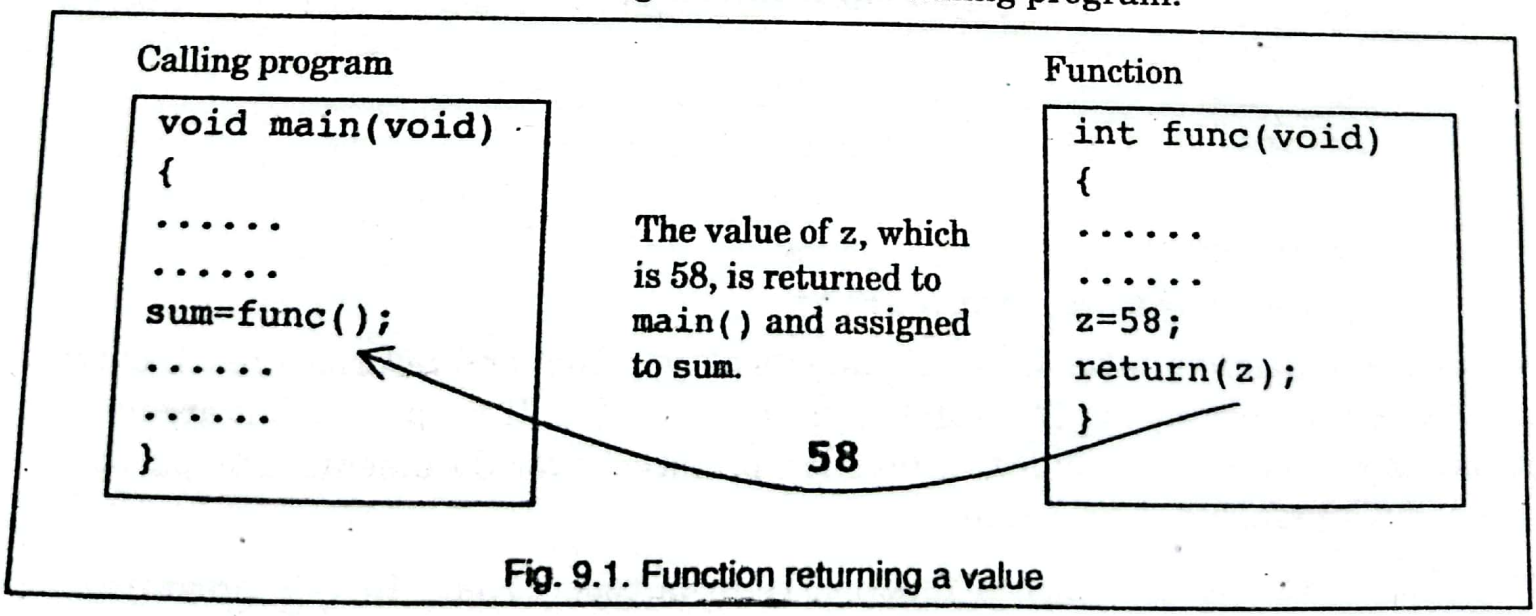


Fig. 9.1. Function returning a value

The `return` statement need not be at the end of the function. It can be placed anywhere in the function; as soon as it is encountered, control will return to the calling program, and any statement following the `return` will not be executed.

CamScanner

Main limitation in the use of the `return()` statement is that you can only use it to return *one value*.

The program 9-5 uses a user-defined function `maximum` to determine and return the largest of the three integers. The three integers are input with `scanf`. Next, the integers are passed to `maximum` which determines the largest integer. This value is returned to main by the `return` statement in `maximum`. The value returned is assigned to the variable `largest` which is then printed in the `printf` statement.

```
/* program 9-5.cpp */
#include<stdio.h>
#include<conio.h>

int maximum(int, int, int);    /* function prototype */
void main(void)
{
   clrscr();
   int a, b, c;

   printf("Enter three integers: ");
   scanf("%d%d%d", &a, &b, &c);
   printf("Maximum is: %d\n", maximum(a, b, c));

   getch();
}

/* Function to find maximum of three integers */
int maximum(int x, int y, int z)
{
   int max = x;

   if (y > max)
      max = y;
   if (z > max)
      max = z;
   return(max);
}
```

The function prototype

```
int maximum(int, int, int);
```

states that maximum takes three arguments of type `int`, and returns a result of type `int`. Note that arguments (x, y, and z) are not included. It is optional but arguments names are sometimes included in function prototypes for documentations purposes. The compiler ignores these names.

Here is an example in which a function returns type `float`. In this program, the `main()` function asks the user for the radius of a sphere, and then calls a function named `area()` to calculate the area of the sphere. The `area()` function returns the area of the sphere in the form of a floating point value.

CamScanner

```
/* program 9-6.cpp */
#include<stdio.h>
#include<conio.h>

float area(float rad);          /* function prototype */
void main(void)
{
  clrscr();
  float radius;
  printf("\nEnter radius of sphere: ");
  scanf("%f", &radius);
  printf("\nArea of sphere is %.2f", area(radius));
  getch();
}

float area(float rad)    /* returns area of sphere */
{
 return(4*3.14159*rad*rad);
}
```

Practical 9

Object

Writing a function, which generates factorial of *n* and calls this function in the 'main' program.

Algorithm

Step 1. Declare a function `double factorial(int x)`, which accepts an integer as an argument and returns a double value.

The Function main

Step 2. Read num.

Step 3. IF num ≥ 0, then
  Call the function factorial(num) and store the value it returns in fact.
  (where num is the actual argument)
  Write fact.
ELSE
  Write Sorry!!
[End of IF-ELSE structure.]

Step 4. Exit.

Defining the Function double factorial(int x)

(where x is the formal argument)

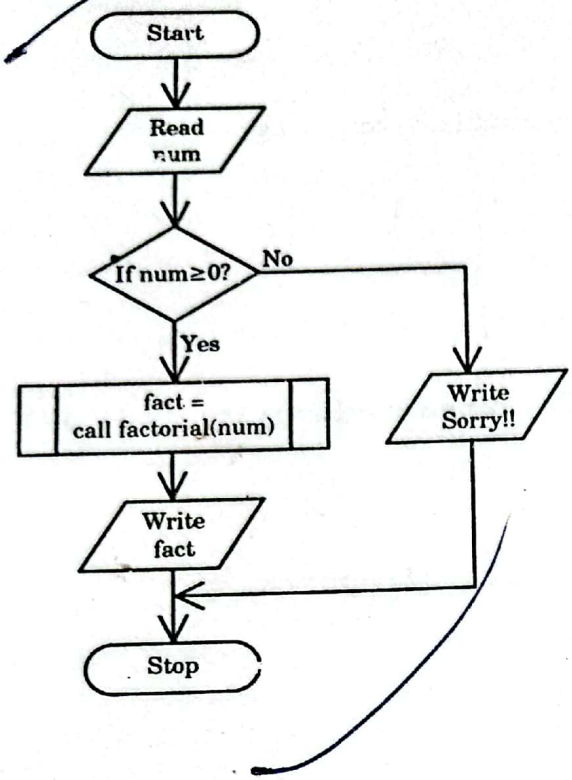
Step 5. Set f = 1 and i = 1.

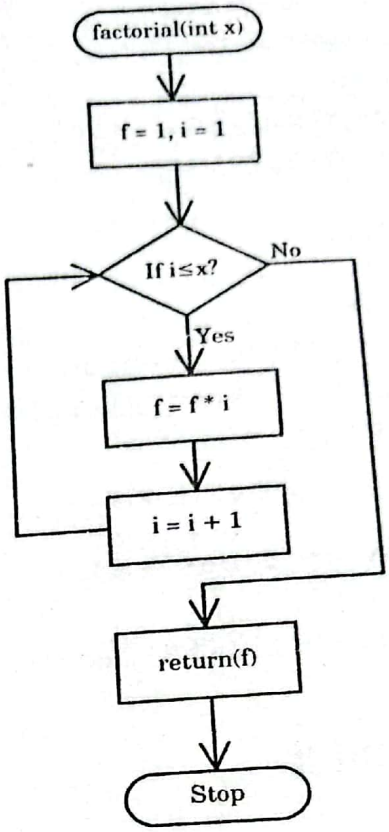
Step 6. Repeat FOR i =1 to i ≤ x
  f = f * i.
[End of FOR loop.]

Step 7. Return f.

CamScanner

Flowchart





Source Code

```
/* Practical 9: Writing a function, which generates Factorial of
n and calls this function in the 'main' program. */

#include<stdio.h>
#include<conio.h>
double factorial(int x);                    /* function prototype */
void main(void)
{
clrscr();
int num;
double fact;
printf("\n\n\n\t\tEnter the number to find its Factorial: ");
scanf("%d", &num);

if (num>=0)
{
   fact = factorial(num);
   printf("\n\n\n\t\tThe factorial of %d is %lf", num, fact);
}
else
   printf("\n\n\n\t\t Sorry!!");
getch();
```

CamScanner

```
double factorial(int x)                    /* factorial function */
{
int i;
double f=1;
for(i=1; i<=x; i++)
   f=f*i;
return(f);                                 /* return function */
}
```

Output

1st Run

```
Enter the number to find its Factorial: 4
The factorial of 4 is 24.000000
```

2nd Run

```
Enter the number to find its Factorial: 75
The factorial of 75 is
2.48091408113953914000000000000000000000e+109
```

3rd Run

```
Enter the number to find its Factorial: -7
Sorry!!
```

4th Run

```
Enter the number to find its Factorial: 0
The factorial of 0 is 1.000000
```

Using More Than One Function in a Program

You can use as many functions as you want in a program, and any of the functions can call any of the other functions. All the functions are accessible to all other functions.

Let us look at a program 9-7 that uses three functions including the `main()`. In this program `main()` calls one of the two functions, `fact()` or `table()`. The function `fact()` calculates the factorial of a number and returns its value to `main()` whereas the function `table()` prints the table of a number.

```
/* program 9-7.cpp */
#include<stdio.h>
#include<conio.h>

double fact(int m);          /* function prototypes */
void table(int n);
void main(void)
{
clrscr();
int c,num;                   /* user-supplied values */
double answer;

printf("\n\n\n");
printf("\t\t\tMENU");
printf("\n\t\t\t____\n\n");
printf("1. Factorial of a number\n");
```

CamScanner

```
        printf("2. Table of a number\n");
        printf("\n");
        printf("Enter the Menu option: ");
        scanf("%d", &c);

        if(c==1)
        {
           printf("\nEnter the number to find its Factorial: ");
           scanf("%d", &num);
           answer=fact(num);
           printf("\nThe factorial of %d is %lf", num, answer);
           getch();
        }
        else if(c==2)
        {
           printf("\nEnter the number whose table is required: ");
           scanf("%d", &num);
           table(num);
           getch();
        }
        else
        {
           printf("\n\nInvalid Menu option");
           printf("\nPress any key to continue...");
           getch();
        }
     }
     double fact(int m)          /* factorial function */
     {
      int i;
      double f=1;
      for(i=1; i<=m; i++)
         f=f*i;
      return(f);                 /* returns factorial */
     }

     void table(int n)           /* table function */
     {
      int i;
      printf("\n***Table of %2d***", n);
      for(i=1; i<=10; i++)
         printf("\n%2d * %2d = %3d", n,i,n*i);
     }
```

Here is example of output from the program:

```
      MENU
1. Factorial of a number
2. Table of a number
```

```
Enter the Menu option: 1

Enter the number to find its Factorial: 21

The factorial of 21 is 51090942171709440000.000000
```

Notice that none of the functions is nested inside any other. Another point to note is that functions can appear in any order in the program listing. But arranging the functions in an order makes them easier to read, especially in larger programs with number of functions. Also note that the `main()` function does not need to be the first one in the program, although it usually is.

Practical 10

Object

Writing a program which uses multiple arguments in a function. (Develop a user-defined function to generate a rectangle. Use the function for passing arguments to draw different sizes of rectangles and squares).

Algorithm

Step 1. Declare a function `void rectangle(int, int)`, which accepts two integer arguments but returns no value.

The Function main

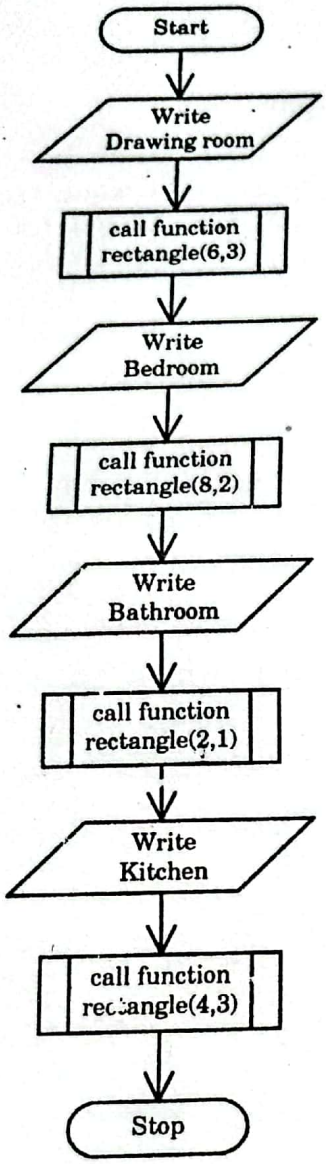
Step 2. Write Drawing room.
Step 3. Call function rectangle(6,3).
(6 and 3 are length and width of the room)
Step 4. Write Bedroom.
Step 5. Call function rectangle(8,2).
Step 6. Write Bathroom.
Step 7. Call function rectangle(2,1).
Step 8. Write Kitchen.
Step 9. Call function rectangle(4,3).
Step 10. Exit.

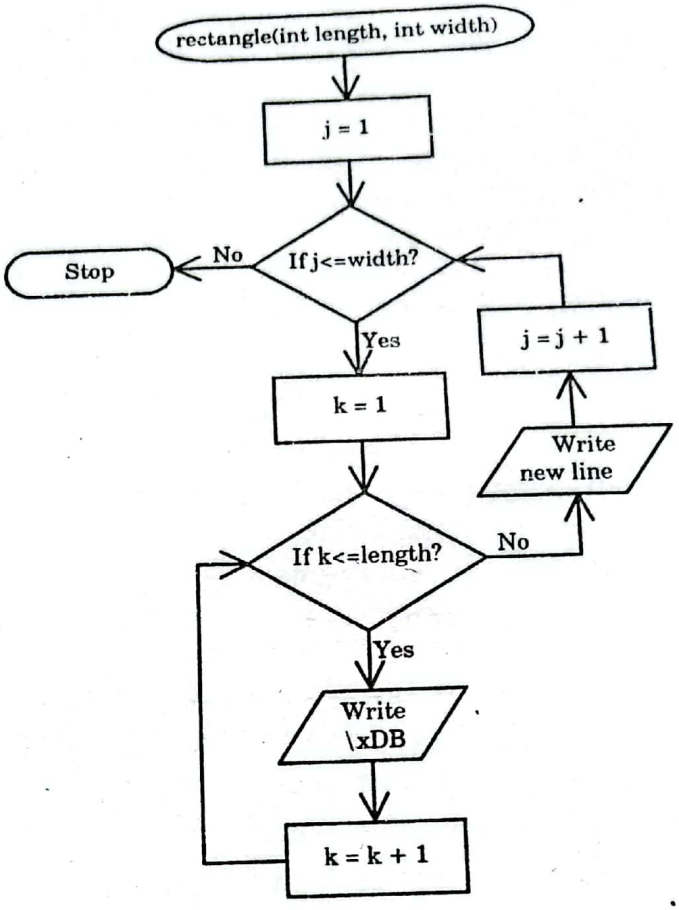
Defining the Function void rectangle(int length, int width)

Step 11. Repeat FOR j = 1 to j <= width
Step 12. Repeat FOR k = 1 to k <= length
Write \xDB.
(where \xDB is the code for one rectangle)
[End of inner FOR loop.]
Write new line.
[End of outer FOR loop.]
Step 13. Exit.

CamScanner

Flowchart





Source Code

```
/* Practical 10: Writing a program which uses multiple arguments
in a function. (Develop a user-defined function to generate a
rectangle. Use the function for passing arguments to draw
different sizes of rectangles and squares). */

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

void rectangle(int length, int width);   /* function prototype */

void main(void)
{
clrscr();
```

CamScanner

```
    printf("\nDrawing room\n");          /* print room name */
    rectangle(6,3);                      /* draw room */
    printf("\nBedroom\n");                /* etc. */
    rectangle(8,2);
    printf("\nBathroom\n");
    rectangle(2,1);
    printf("\nKitchen\n");
    rectangle(4,3);
    getch();
    }

    /* function draws rectangle of particular length and width */
    void rectangle(int length, int width)
    {
    int j,k;
    for(j=1; j<=width; j++)              /* number of lines */
       {
        for(k=1; k<=length; k++)         /* line of rectangles */
           printf("\xDB");               /* print one rectangle */
        printf("\n");                    /* new line */
       }
    }
```

Output

```
Drawing room
```

```
Bedroom
```

```
Bathroom
```

```
Kitchen
```

Recursive Function

Recursion is a method of problem solution that depends on successively restating the problem in a consistently similar manner, but in a simpler form until a position is reached where no more simplification can take place and the simplest case reaches a *base value*. This fundamental solution can then be used to recycle through the decomposed problem elements thus reaching a solution.

CamScanner

When applied to computer languages, recursive function means that a function can call *itself*. Although at first this seems mysterious, recursive solutions can sometimes be simpler to write (and read) than iterative solutions.

We shall consider an example which is easy to describe and evaluate.

Multiplication can be considered in a recursive manner since it can be written as a repeated addition, each successive term being a smaller instance of the problem with the terminating condition occurring when the last term evaluates to zero

$$x \times n = x \times (n-1) + x$$
$$x \times (n-1) = x \times (n-2) + x$$
until $x \times 0 = 0$

the function could be called multiply(x,n); multiply(x,n) can be decomposed until $n = 0$.

Consider the multiplication of 5×3 and decompose it successively into 'smaller' instances

5×3	decomposes to	$5 \times 2 + 5$
5×2	decomposes to	$5 \times 1 + 5$
5×1	decomposes to	$5 \times 0 + 5$
5×0	decomposes to	0, *i.e.* the termination value.

recycling from this position the evaluation is

5×1	becomes	$5 \times 0 + 5 = 5$
5×2	becomes	$5 \times 1 + 5 = 10$
5×3	becomes	$5 \times 2 + 5 = 15$

In computing terms the procedure or function is made to *called itself* until the final case is reached. The function calls are then recycled in reverse order to produce the solution.

A recursive definition must have some terminating value. Note that if there is not a *base case* then disaster will occur since the recursion cannot conclude.

The factorial is another common recursively defined structure, *i.e.*

$$\text{factorial } n = n! = n \times (n-1) \times (n-2) \times \ldots \times 1 \qquad \text{for } n > 0$$

This can clearly be expressed recursively as

$$\text{factorial}(n) = \text{factorial}(n-1) \times n$$

and the terminating condition happens at

$$\text{factorial}(1) = 1$$

for example $7! = 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1$

Also

$n! = n \times (n-1)!$	*e.g.* $7! = 7 \times 6!$
$(n-1)! = (n-1) \times (n-2)!$	*e.g.* $6! = 6 \times 5!$
$(n-2)! = (n-2) \times (n-3)!$	*e.g.* $5! = 5 \times 4!$ and so on.

Here $n!$ is defined in terms of $(n-1)!$, which is defined in terms of $(n-2)!$, which is defined in terms of $(n-3)!$, and so on until we reach 0! which is defined as 1.

CamScanner

Example 9.1

Let us calculate 4! using the recursive definition. This calculation requires the following nine steps.

1. $4! = 4 \times 3!$
2. $3! = 3 \times 2!$
3. $2! = 2 \times 1!$
4. $1! = 1 \times 0!$
5. $0! = 1$
6. $1! = 1 \times 1 = 1$
7. $2! = 2 \times 1 = 2$
8. $3! = 3 \times 2 = 6$
9. $4! = 4 \times 6 = 24$

That is

Step 1. This defines 4! in terms of 3!, so we must postpone evaluating 4! until we evaluate 3! This postponement is indicated by indenting the next step.

Step 2. Here 3! is defined in terms of 2!, so we must postpone evaluating 3! until we evaluate 2!

Step 3. This defines 2! in terms of 1!

Step 4. This defines 1! in terms of 0!

Step 5. This step can explicitly evaluate 0!, since 0 is the *base value* of the recursive definition.

Step 6 to 9. We backtrack, using 0! to find 1!, using 1! to find 2!, using 2! to find 3!, and finally using 3! to find 4! This back-tracing is indicated by the reverse indention.

The following is the program that calculates factorial of a given number.

```
/* program 9-8.cpp */
#include<stdio.h>
#include<conio.h>

double factorial(int mum);      /* function prototype */
void main(void)
{
  clrscr();
  int num;
  double fact;

  printf("\nEnter the number to find its Factorial: ");
  scanf("%d", &num);
  fact = factorial(num);
  printf("\nThe factorial of %d is %lf", num, fact);
```

CamScanner

```
    getch();
}

double factorial(int mum)
{
  if(mum==0)                              /* Base case */
     return(1);
  else
     return(mum*factorial(mum-1));        /* Recursive case */
}
```

When num is positive, the function computes num *times the result of the next lower value of num factorial*. Each successive call diminishes the value of num until it reaches 0. At that point the function stops the *recursive decent* and returns 1. As it would with any chain of unfinished function calls, the computer retraces its steps and completes the processing, filling in the missing values. At the final retrace, the function returns the value of num!.

CamScanner

Exercise

9.1. What is the difference between a function in the C library and a function you write yourself?

9.2. What are five advantages of using functions?

9.3. What is a function prototype?

9.4. What occurs after a function is performed?

9.5. Explain the difference between local and global variables.

9.6. How do you pass a value to a function?

9.7. How can you return a float value?

9.8. Is the use of functions a more efficient use of the programmer's time or the computer's time? Explain your answer.

9.9. When do you use the keyword `return` while defining a function? When do you not use the keyword return when defining a function?

9.10. Writ a quiz program that asks four questions, each question and answer in a different function.

9.11. Write a program in C to call a function which returns the cube of a given number.

9.12. Write a C program that asks for a distance in metres and converts it into feet.

9.13. Write a program in C that prints out the larger of two numbers entered from the keyboard. Use a function to do the actual comparison of the two numbers. Pass the two numbers to the function as arguments, and have the function return the answer.

9.14. Write a program that inputs a number and then uses a function to calculate and display the number to the fourth power.

9.15. Write a program in C that asks the user to input length of the two sides of a right angled triangle. Make use of the built-in functions `pow` and `sqrt` functions to compute the length of the hypotenuse using the Pythagorean theorem.

9.16. Explain what is wrong with the following program:

```
#include<stdio.h>
void dothis(void)
{
   puts("This is first");
   main();
}
void main(void)
{
   puts("This is second");
}
```

CamScanner

9.17. Is the following a correctly structured program?

```
#include<stdio.h>
void main(void)
{
   int three=3;
   type(three);
}
void type(float num)
{
   printf("%f", num);
}
```

9.18. Is this correctly written function?

```
abs(num);
{
 int num;
 if (num<0)
    num = -num;
 return(num);
}
```

9.19. Write a program using a function to calculate the area of a rectangle.

9.20. Write a function `pow(x,y)` to calculate the value of `x` raised to `y`.

9.21. Write a function which receives a `float` and an `int` from `main()`, finds the product of these two and returns the product which is printed through `main()`.

9.22. Write a function which receives 5 integers and returns the product, sum and average of these numbers. Call this function from `main()` and print the results in `main()`.

9.23. What happens if the user calls recursive function with a negative argument?

Ans. The function goes into infinite recursion if the argument is negative since the base case is never reached.

9.24. Rewrite the following mathematical expressions using C functions:

(a) $\sqrt{(x-y)^4}$ (b) $\sqrt{u+v} \times w^4$ (c) $\left|xy - \frac{w}{z}\right|$

9.25. A 5-digit positive integer is entered through the keyboard, write a function to calculate sum of digits of the 5-digit number:

(a) without using recursion.

(b) using recursion.

9.26. Write a recursive function to obtain the first 25 numbers of a Fibonacci sequence. In a Fibonacci sequence the sum of two successive terms gives the third term. Following are the first few terms of the Fibonacci sequence:

0 1 1 2 3 5 8 13 21 34 55 89 ...

CamScanner

Objectives and MCQs

9.1. True or false: Functions are called mini-programs because they are usually short and have input, output, and computations.

9.2. A program module in C is called a function.

9.3. According to the principles of software design, functions
   (a) should compute at least two results to be worth the overhead.
   (b) should be short and compute one result.
   (c) should be as long as necessary but compute one result.
   (d) are not really necessary for good software design.

9.4. Which of these are valid reasons for using functions?
   (a) they use less memory than repeating the same code.
   (b) they run faster.
   (c) they keep different program activities separate.
   (d) they keep variables safe from other parts of the program.

9.5. True or false: The function header and the function call have identical syntax.

9.6. A function prototype
   (a) is an outline of the purpose of the function.
   (b) is the same as the function header.
   (c) is the same as the function call.
   (d) is not necessary in *Turbo C++*.

9.7. True or false: Function prototypes are not needed in C.

9.8. Function is invoked with a function call.

9.9. Use the `void` data type when the function does not return a value.

9.10. True or false: A function can still be useful even if you cannot pass it any information and cannot get any information back from it.

9.11. True or false: In C language, you can place the function `main()` any where in your program but it will still be executed first.

9.12. A variable that is known only within the function in which it is defined is called a local variable.

9.13. A global variable is defined in a declaration
   (a) in `main()` only.
   (b) in the first function that uses it.
   (c) in any function that uses it.
   (d) outside of any function.

9.14. A global variable can be referenced in a declaration
   (a) in `main()` only.
   (b) in the first function that uses it.
   (c) in any function that uses it.
   (d) outside of any function.

CamScanner

9.15. Which of the following are valid reasons for using arguments in functions?
(a) to tell the function where to locate itself in memory.
(b) to convey information to the function that it can operate on.
(c) to return information from the function to the calling program.
(d) to specify the type of the function.

9.16. True or false: A global variable can be used in `main()` only.

9.17. True or false: The variables commonly used in C functions are accessible to all other functions.

9.18. Which of the following can be passed to a function via arguments?
(a) constants.
(b) variables (with values).
(c) preprocessor directives.
(d) expressions (that evaluate to a value).
(e) functions (that return values).

9.19. Which of the following is true?
(a) C functions are all equal.
(b) C functions can be nested within each other.
(c) C functions are arranged in a strict hierarchy.
(d) C functions can only be called from `main()`.

9.20. True or false: To return from a function you must use the keyword `return`.

9.21. A function that calls itself is a recursive function.

CamScanner

CHAPTER 10

ARRAYS

C has several methods for organising variables of simple types (*e.g.*, int, float). These include grouping of homogeneous variables (arrays), heterogeneous variables (structures) and overlapping variables (unions).

An *array* is a collection of variables of the same type that are collectively referred to by the common name. In C all arrays are stored respectively in successive memory locations. We can manipulate array data type collectively or individually depending on the requirements of each particular application.

An individual variable in the array is called an *array element*. You can access individual array element by specifying the array name followed by an *index* (sometimes called *subscript*), with each index must be expressed as a nonnegative integer. In the past, computers did not display subscripts (literally, below the line script such as "x_5") so many languages use brackets or parentheses to denote an array index. C uses square brackets. For example, if x is an array then `x[5]` refers to the element in x whose index is 5. This is read *x sub 5* for x subscript 5.

Arrays may have from one to several dimensions.

One-Dimensional Arrays

You declare an array by designating the type of values it will store, name of the array and the maximum number of elements.

The general structure of the command is:

```
type array_name[size];
```

where *type* is a valid C data type (*e.g.* int, float, char, double etc.), *array_name* is the name of the array, and *size* specifies the number of elements in an array.

CamScanner

For example, to declare an array of 5 integer numbers called *temps*, use this statement.

```
int temps[5];
```

Note the use of square brackets, rather than parentheses or braces. The square brackets [ ] tells the compiler that we are dealing with an array. As with any other variable type, you declare an array before `void main(void)` to make it external (available to all functions), or inside `void main(void)` or another function to make it automatic.

There are two restrictions in the use of arrays. First, you have to specify the amount of storage for a declared array at compile time (before execution). Therefore, an array must have a fixed (non-variable) size. Second, the data type of an array applies uniformly to all the elements and for this reason, an array is called a *homogeneous* data structure.

The first element in every array is the *zeroth element*. Thus, the first element of array `temps` is referred to as `temps[0]`. The second element is `temps[1]`, the third element is `temps[2]`, and so on. Notice that the elements are numbered starting with 0, so the five-element array shown in the Fig. 10.1 has elements numbered 0 to 4. There is no element `temps[5]`. This sometimes confusing style of indexing, called *zero-based indexing*, has to do with the method by which C accesses array elements.

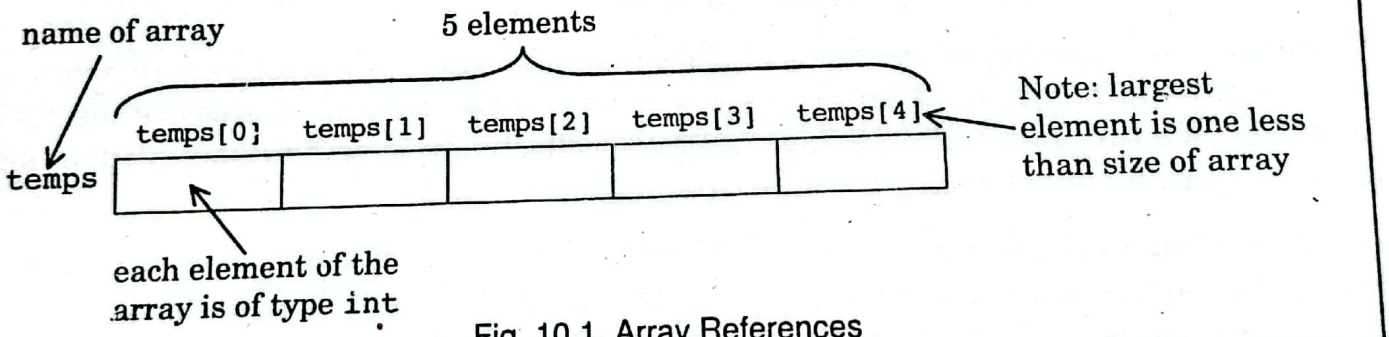


Fig. 10.1. Array References

You cannot use an index number higher than one less than the declared number of elements. If you declare a 5-element array, you must make sure you never use an index number higher than 4. Your compiler won't pick up this potential problem during the compilation process. Instead, you will either receive a runtime error when you execute the program, or get strange results.

Declaration of more than One Array

Declaration of more than one array can be made in one statement as given below.

```
int temps[5], index[7];
float fact[9], m, total;
```

In the above statements, *temps* and *index* are the arrays with five and seven elements of the data type `int`. The *fact* is an array with nine elements of the data type `float`. In addition, *m* and *total* are two simple variables of data type `float`.

CamScanner

Array Initialisation

We can initialise a simple variable when we declare it as:

```
int total = 0;
```

Like this we can also initialise an array at the time of its declaration. Let temps is one-dimensional int array having 5 elements, can be declared and initialised as:

```
int temps[5] = {53, 85, 29, 19, 63};
```

This creates five elements with these values:

```
temps[0]    53
temps[1]    85
temps[2]    29
temps[3]    19
temps[4]    63
```

You can also omit the size of an array as

```
int temps[] = {53, 85, 29, 19, 63};
```

because size can be deduced from the initialisation list.

A sequence of statements that manipulate this array is shown in Table 10.1.

Table 10.1. Statements that Manipulate Array temps

Statement	Explanation
`printf("%d", temps[0]);`	Displays the value of temps[0], which is 53.
`temps[4] = 37;`	Stores the value 37 in temps[4].
`sum = temps[2] + temps[4];`	Stores the sum of temps[2] and temps[4], which is 66 in the variable sum.
`sum += temps[1];`	Adds temps[1] to sum. The new sum is 151.
`temps[4] += 2;`	Adds 2 to temps[4]. The new temps[4] is 39.
`temps[1] = temps[0] + temps[3];`	Stores the sum of temps[0] and temps[3] in temps[3]. The new temp[1] is 72.

Entering Values into an Array

Once you declare an array, you can enter information into it. If you want to assign values to every element, however, it is faster to use a loop. If you know the number of elements you want to assign, use a for loop:

```
/* program 10-1.cpp */
#include<stdio.h>
#include<conio.h>
void main(void)
{
  clrscr();
  int temps[5];
  int index;
  for(index = 0; index < 5; index++)
```

CamScanner

```
    {
     printf("Enter temperature %d: ", index);
     scanf("%d", &temps[index]);
    }

  getch();
}
```

The variable `index` is used here to determine the number of repetitions. In this case, the loop is repeated 5 times, once for each element in the array. Each time the loop repeats, you will see the prompt

```
Enter temperature
```

followed by the subscript number of the element to be input and colon.

Notice that the 5 repetitions are performed by incrementing the variable from 0 to 4, not from 1 to 5.

In the `scanf()` statement, we have used the address operator (`&`) on the element of the `&temps[index]` array, just as we have used it earlier on other variables (`&cost`, for example in program 7-2) to be read in by the `scanf()` function. In so doing, we have passing the *address* of this particular array element to the function, rather than its value; this is what `scanf()` requires.

If not enough data values are specified, the remaining elements of the array are set to 0. You can also quickly initialise an array to zero using this loop:

```
for(index = 0; index < 5; index++)
    temps[index]=0;
```

Using an array, you can input any number of elements by just changing the declaration and the loop condition.

Manipulating Arrays

You can use an array element in any instruction where you would use a variable—in input or output commands or in expressions. You always refer to the individual element by its subscript number.

The values stored in an array will be accessible whenever needed in the program. The program 10-2, for instance, reads in seven temperatures of the week and uses an array in two different tasks. The values of the array elements are averaged, and then the array is used again to print a conversion chart. Each time, the array is accessed in a `for` loop that increments the index variable from 0 to the last array element.

```
/* program 10-2.cpp */
#include<stdio.h>
#include<conio.h>

void main(void)
{
  clrscr();
  int temps[7];
```

CamScanner

```
    int index, total;
    flcat average, celsius;
    total=0.0;
       /*load the array with values */
    for(index=0; index < 7; index++)
       {
        printf("Enter temperature %d: ", index);
        scanf("%d", &temps[index]);
       }
       /* read the values to average */
   for(index=0; index < 7; index++)
       total += temps[index];
   average=total/7.0;
   printf("\nAverage temperature of the week is: %f\n\n\n", average);
   puts("Fahrenheit\tCelsius\n");

      /* read the values to convert */
   for(index=0; index < 7; index++)
      {
       celsius = (5.0/9.0)*(temps[index]-32);
       printf("%d\t\t%6.2f\n", temps[index], celsius);
      }

  getch();
}
```

In program 10-3, the `for` loops of the previous program have been replaced with `do-while` loops, and the instruction

```
index=0;
```

is used before each loop to initialize the repetition to start with the first array element. A separate instruction like this is not needed with a `for` loop, because the value is initialized in the loop arguments.

```
/* program 10-3.cpp */
#include<stdio.h>
#include<conio.h>

void main(void)
{
 clrscr();
 int temps[7];
 int index, total;
 float average, celsius, count;
 total = 0.0;
   /*load the array with values */
index=0;
do
{
    printf("Enter temperature %d: (555 to quit): ", index);
```

CamScanner

```
        scanf("%d", &temps[index]);
        index++;
    }
    while(index < 7 && temps[index-1] != 555);

     /*average the array*/
    index=0;
    do
    {
       total += temps[index];
       index++;
    }
    while(index < 7 && temps[index] != 555);
       count=index;
    average=total/count;
    printf("\nAverage temperature is: %f\n\n\n", average);
    puts("Fahrenheit\tCelsius\n");

     /* convert the values*/
    index=0;
    do
    {
       celsius = (5.0/9.0)*(temps[index]-32);
       printf("%d\t\t%6.2f\n", temps[index], celsius);
       index++;
    }
    while(index < 7 && temps[index] != 555);
    getch();
}
```

The values are input until all 7 array elements have been entered or until you enter the flag value 555, using this condition:

```
while(index < 7 && temps[index-1] != 555);
```

Notice that 1 is subtracted from the variable index because the variable is incremented after the number is entered. If you do not enter all 7 elements, the number 555 that you enter to end the repetition is inserted into the array element after the last actual temperature.

The number 555 is used as a flag. The program logic will watch for that value to determine whether you stopped entering temperatures before the end of the array.

When the array is averaged, a do-while loop totals all of the values up to the end of the array or until it reaches the value 555. While the variable index contains the number of temperatures, it must be an integer to be used as an array subscript. To obtain a float value from a division operation, however, the values divided must be floats. So the program defines a float variable called count and assigns it the number of temperatures before using it to perform the mathematics:

```
count = index;
average = total/count;
```

CamScanner

Because the last line of the loop increments the index variable, the value of index will actually be one more than the last array element that contains a valid temperature. If you enter values into array elements 0 through 4, for example, the value of index will be 5. That value, however, also represents the number of elements you entered—0 through 4 is five elements—so it is used as the count to compute the average.

Examining an Array

By using repetition, you can scan the array elements, comparing the values in the array or searching for a specific value. Consider this program:

```
/* program 10-4.cpp */
#include<stdio.h>
#include<conio.h>

void main(void)
{
  clrscr();
  int array[10];
  int index, large, small;

  /*load the array with values */
  for(index=0; index < 10; index++)
     {
      printf("Enter a number for position %d: ", index);
      scanf("%d", &array[index]);
     }

 /* find large and small numbers */
 large = array[0];
 small = array[0];
 for(index=0; index < 10; index++)
    {
     if(array[index] > large)
        large = array[index];
     if(array[index] < small)
        small = array[index];
    }
 printf("\n\nThe largest of the 10 Elements is: %d ", large);
 printf("\n\nThe smallest of the 10 Elements is: %d ", small);

 getch();
}
```

Assuming you have declared `large` and `small` as integer variables, this program examines each element in the array to determine the largest and smallest values. The key is to assign the first array element initially to both the `large` and `small` variables. If you only looked at one element, it would naturally be both the highest and the lowest. This "seeds" the variables, which are then compared against the

CamScanner

other array variables. If an array value is greater than the current value of large, then it becomes the new largest value. If it is less than the current value in small, it becomes the new lowest value.

Example 10.1

The #define pre-processor directive is introduced in program 10-5. Line 5

```
#define SIZE 10
```

defines a *symbolic constant* SIZE whose value is 10. A symbolic constant is a programmer-defined identifier that is replaced with replacement *text* by the C pre-processor before the program is compiled. When the program is preprocessed, all occurrences of the symbolic constant SIZE are replaced with the replacement text 10. Using symbolic constants to specify array sizes makes programs more *scalable*. In program 10-5, the first for loop could fill a 1000-element array by simply changing the value of SIZE in the #define directive from 10 to 1000. If the symbolic constant SIZE had not been used, we would have to change the program in three separate places to scale the program to handle 1000 array elements. As programs get larger, this technique becomes more useful for writing clear programs.

The program reads numbers from an array and graphs the information in the form of a bar chart or histogram—each number is printed, then a bar consisting of that many asterisks is printed beside the number. The nested for loop actually draws the bars.

```
/* program 10-5.cpp
   Histogram printing program */
#include<stdio.h>
#include<conio.h>
#define SIZE 10

void main(void)
{
  clrscr();
  int n[SIZE];
  int i, j;
  for(i=0; i <= SIZE-1; i++)       /*load the array with values */
     {
      printf("Enter any number %d: ", i);
      scanf("%d", &n[i]);
     }
  printf("\n%s%13s%17s\n", "Element", "Value", "Histogram");
  for(i = 0; i <= SIZE-1; i++)
     {
      printf("%7d%13d          ", i, n[i]);
       for(j = 1; j <= n[i]; j++)          /* print one bar */
          printf("%c", '*');
      printf("\n");
     }
 getch();
}
```

CamScanner

Example 10.2

To copy the contents of one array to another, you must copy each element of the array separately. For example, this program loads array a1 with the numbers 1 through 5 and then copies them into array a2.

```
/* program 10-6.cpp */
#include<stdio.h>
#include<conio.h>

void main(void)
{
  clrscr();
  int a1[5], a2[5];
  int i;
  for(i=1; i < 6; i++)
     {
      printf("Enter any number %d: ", i);
      scanf("%d", &a1[i-1]);
     }
  for(i=1; i < 6; i++)
     a2[i-1] = a1[i-1];

  printf("\n\n\t%s\t%s\n", "First array", "Second array");

  for(i=1; i < 6; i++)
     printf("\t%d\t\t%d\n ", a1[i-1], a2[i-1]);

  getch();
}
```

Example 10.3

Create three arrays. Read data into the first two of them. Subtract each element in the second array from the corresponding element in the first array. Store the differences in the third array. Print all the arrays.

Algorithm

Step 1. Loop all array elements
  Read first[*i*].

Step 2. Loop all array elements
  Read second[*i*].

Step 3. Loop all array elements
  difference[*i*] = first[*i*] − second[*i*].

Step 4. Loop all array elements
  Write first[*i*], second[*i*], difference[*i*].

Step 5. Exit.

CamScanner

Source Code

```
/* program 10-7.cpp */
#include<stdio.h>
#include<conio.h>
#define SIZE 5

void main(void)
{
  clrscr();
  int first[SIZE], second[SIZE], diff[SIZE];
  int i;

  /*load the first array with values */
  for(i=0; i < SIZE; i++)
     {
      printf("Enter data item for first array for position %d: ", i);
      scanf("%d", &first[i]);
     }
  printf("\n\n");

  /*load the second array with values */
  for(i=0; i < SIZE; i++)
     {
      printf("Enter data item for second array for position %d: ", i);
      scanf("%d", &second[i]);
     }

  for(i=0; i < SIZE; i++)       /*compute differences */
      diff[i] = first[i] - second[i];

  printf("\n\n%s\t%s\t%s","First Array","Second array","Difference");
  for(i=0; i < SIZE; i++)
     printf("\n%d \t\t%d \t\t%d", first[i], second[i], diff[i]);

  getch();
}
```

Output

```
Enter data item for first array for position 0: 37
Enter data item for first array for position 1: 6
Enter data item for first array for position 2: 106
Enter data item for first array for position 3: 84
Enter data item for first array for position 4: -24

Enter data item for second array for position 0: 5
Enter data item for second array for position 1: 6
Enter data item for second array for position 2: 0
Enter data item for second array for position 3: 114
Enter data item for second array for position 4: 17
```

CamScanner

First Array	Second array	Difference
37	5	32
6	6	0
106	0	106
84	114	-30
-24	17	-41

Sorting an Array

Sorting refers to the operation of arranging data elements in a desired sequence (order), such as increasing or decreasing. For example, a bank sorts all cheques by account number so that it can prepare individual bank statements at the end of each month. Telephone companies sort their lists of accounts by last name and, within that, by first name to make it easy to find phone numbers. Virtually every organisation must sort some data and in many cases massive amounts of data.

Some important and frequently used methods of sorting are:

1. Bubble sort.
2. Insertion sort.
3. Quick sort.
4. Merge sort.
5. Radix sort.

Of these five methods bubble sort is very simple to understand and easy to code. The general concept behind the bubble sort, indeed how it got its name, is the repeated comparisons and, if necessary, exchanges of adjacent elements. This is a little like bubbles in a tank of water with each bubble, in turn, seeking its own level.

Suppose an array a with n elements a[1], a[2], a[3], … , a[n] is in memory. The bubble sort algorithm works as follows:

Pass 1. Compare a[1] with a[2] and arrange them in the desired order, so that a[1] < a[2]. Then compare a[2] with a[3] and arrange them so that a[2] < a[3]. Continue until we compare a[n-1] with a[n] and arrange them so that a[n-1] < a[n].
Observe that Pass 1 involves n−1 comparisons. (During Pass 1, the largest element is "Bubble up" to the n^{th} position). When Pass 1 is completed, a[n] will contain the largest element.

Pass 2. Repeat Pass 1 with one less comparison; that is, now we stop after compare and possibly rearrange a[n-2] and a[n-1]. (Pass 2 involves n−2 comparisons and, when Pass 2 is completed, the second largest element will occupy a[n-1]).

Pass 3. Repeat Pass 1 with two less comparisons; that is, we stop after we compare and possibly rearrange a[n-3] and a[n-2].

. .....

. .....

. .....

Pass $n-1$. Compare a[1] with a[2] and arrange them so that a[1] < a[2].

CamScanner

After n−1 passes, the list will be sorted in increasing order.

The process of sequentially traversing through all or part of a list is frequently called a "pass", so each of the above steps is called a *pass*. Accordingly, the bubble sort algorithm requires n−1 passes, where n is the number of array elements.

```
/* program 10-8.cpp */
#include<stdio.h>
#include<conio.h>

void main(void)
{
  clrscr();
  int array[100];
  int i, j, t;
  int count;

     /*read in numbers */
  printf("How many numbers? ");
  scanf("%d", &count);
  printf("\n");

  for(i=0; i < count; i++)
     {
      printf("Enter number %d: ", i);
      scanf("%d", &array[i]);
     }
  printf("\n");

     /* now, sort them using a bubble sort */
  for(i=1; i < count; ++i)
     for(j=count-1; j >= i; --j)
     {
     /* compare adjacent elements */
      if(array[j-1] > array[j])
      {
      /*exchange element */
         t = array[j-1];
         array[j-1] = array[j];
         array[j] = t;
      }
     }
     /*dispaly sorted list */
     printf("\nThe sorted list is:\n");
     for(t=0; t < count; t++)
     printf("\t%d\n", array[t]);

  getch();
}
```

CamScanner

Searching an Array

Suppose an array a with n elements a[1], a[2], a[3], ... , a[n] is in memory, and suppose a specific *item* of information is given. *Searching* refers to the operation of finding the location of *item* in array a, or printing some message that *item* does not appear there. The search is said to be *successful* if *item* does appear in array a and *unsuccessful* otherwise.

There are many different searching methods. The method that one chooses generally depends on the way the information in array a is organised.

Two important and frequently used methods of searching are:

1. Linear search.
2. Binary Search.

Of these two methods we will discuss only linear search here.

Suppose a is a linear array with n elements in memory as:

a[1], a[2], a[3], ... , a[n]

The most obvious way to search for a given *item* in array a is to compare *item* with each element of a one by one. That is, first we test whether a[1]=*item*, and then we test whether a[2]=*item*, and so on. This method, which traverses array a sequentially to locate *item*, is called *linear search* or *sequential search*.

If you are just checking for any occurrence of the *item*, you can stop the search after the first matching *item* is found. If you have a thousand-element array, and you find a match in the first element, there is no need to continue searching the remaining 999 values.

Here is a program to find a value in our sample array:

```
/*program 10-9.cpp*/
#include<stdio.h>
#include<conio.h>

void main(void)
{
  clrscr();
  int array[10];
  int index, num, found;

   /*load the array with values */
  for(index=0; index < 10; index++)
     {
      printf("Enter any number %d: ", index);
      scanf("%d", &array[index]);
     }
  printf("\nEnter the value to search for: ");
  scanf("%d", &num);
```

CamScanner

```
    for(index=0; index < 10; index++)
       {
       if(num==array[index])
        {
          found=1;
          break;
        }
       }
    if(found==1)
       printf("\nThe number is in position: %d", index);
    else
       puts("\nThe number is not in the list");

 getch();
 }
```

The program assumes you have declared an integer variable named found. The variable is used as a flag; it indicates that a match has been found, and so we can stop looking at the remaining array elements.

The linear searching method works well for small arrays or for unsorted arrays. However, for large arrays linear searching is inefficient.

Example 10.4

Array can be loaded using do-while loops and linear searching can be performed effectively using while loop instead of for loop. Consider program 10-10:

```
/*program 10-10.cpp*/
#include<stdio.h>
#include<conio.h>

void main(void)
{
  clrscr();
  int array[10];
  int index, num, found, spot;

  /*load the array with values */
  index=0;
  do
  {
     printf("Enter any number %d: (555 to quit): ", index);
     scanf("%d", &array[index]);
     index++;
  }
while(index< 10 && array[index-1] != 555);
printf("\nEnter the value to search for: ");
scanf("%d", &num);
```

CamScanner

```
    index=0;
    found=0;
    while(!found && index < 10 && array[index]!=555)
    {
       if(array[index]==num)
       {
        found=1;
        spot=index;
       }
       else
        index++;
    }
    if(!found)
       puts("\nThe number is not in the list");
    else
       printf("\nThe number is in position: %d", spot);

    getch();
}
```

Here we use `found` as a unary operator. The expression

```
while(!found)
```

means "while not found". This statement is true when `found` has a zero value. Since we assign `found` an initial value of 0 outside the loop, the condition is true for the first repetition, and the program examines the first array element (number 0) for a match.

When a match is located, the `if` condition assigns `found` the value of 1. This means that "while not found" is no longer true, and the loop ends. The final value of the variable `index` will contain the element number in which the match was located.

The final `if` condition prints an appropriate message depending on whether the match was found.

Notice that the `while` statement contains three conditions with `AND` logic operators. The loop will repeat as long as a match has not been found, the end of the array has not been reached, and the value 555 is not located.

Passing an Array to a Function

You can pass an *individual* array element or the entire array to a function. Passing an individual array element to a function is like passing a variable. You list the array name, followed by a subscript reference to the desired element. The program 10-11 computes the sum of the first three elements in an array (*i.e.* `sum = x[0]+x[1]+x[2]`) and show its output.

```
/* program 10-11.cpp */
#include<stdio.h>
#include<conio.h>
```

CamScanner

```
int sum_elements(int a, int b, int c);    /* Function prototype */
void main(void)
{
  clrscr();
  int x[10], total;
  x[0] = 10;          /* Assign values to the first three elements */
  x[1] = 20;
  x[2] = 30;

  total = sum_elements(x[0], x[1], x[2]); /* Compute the sum */
  printf("\nFor values: %d %d %d",x[0], x[1], x[2]);
                                          /* Output the results */
  printf(" the sum = %d", total);
  getch();                                /* End of main function */
  }

int sum_elements(int a, int b, int c)     /* Function header */
{
  int sum;
  sum = a+b+c;
  return(sum);                            /* End of sum elements */
}
```

When you pass a variable other than an array, C makes a copy of the data and places it in a memory location associated with the receiving variable. There are two copies of the data, and changing the variable in the receiving function does not change the original variable.

When you pass an array, you are actually passing the address of the array. C does not make a copy of the entire array but merely assigns the same address area, and thus its data, to a second array name. Program 10-12 shows how to pass an array as an argument. Notice that only the array name is used in the calling argument, without a subscript.

```
/* program 10-12.cpp */
#include<stdio.h>
#include<conio.h>
int large(int[], int);                    /* Function prototype */
void main(void)
{
  clrscr();
  int list[5];
  int index, num;

  /*load the list with values */
  for(index=0; index < 5; index++)
     {
      printf("Enter a number for position %d: ", index);
      scanf("%d", &list[index]);
     }
```

CamScanner

```
    num = large(list, index-1);          /* get largest number */
    printf("\n\nLargest number is %d", num);     /*print it */
    getch();
}

int large(int list[], int index)
{
  int dex, large;
  large = list[0];                    /* Assume 1st element largest */
  for(dex=1; dex <= index; dex++)     /* Check remaining elements */
     if(list[dex] > large)            /* If one bigger, */
       large = list[dex];             /* make it the largest */
  return(large);
}
```

Because you are actually passing the address of the array, you do not have to specify the number of elements in the receiving argument; just use empty brackets to indicate an array.

Passing an array by its address rather than duplicating it saves memory, but it can have some side effects. Changing an array element in the receiving function will also change the value of the passed array.

Multi-Dimensional Array

An array of arrays is a multi-dimensional array. In other words, multi-dimensional array is an array with two or more dimensions. As we can visualise one-dimensional array as a contiguous list, we usually visualise a two-dimensional array as a matrix or table. You can also create higher-dimensional array. But higher-dimensional arrays are more complicated.

A two-dimensional array is declared with two pairs of square brackets. For example, the following are all valid array declarations.

```
int grades[20][5];        /*20 students, each having 5 grades*/
float rainfall[12][31]; /*daily rain for on year*/
double table [50][100]; /*spreadsheet with 50 rows, 100 columns*/
char puzzle[25][25];     /*crossword puzzle*/
```

In each case the declaration indicates the maximum number of rows followed by the maximum number of columns. As with one-dimensional arrays, the indexing is zero-based. For the declaration above, the last element you can reference in the grades array is `grades[19][4]`.

You can initialise two-dimensional arrays when you declare them. The initialiser list can be arranged by row, with each row enclosed in braces or arranged linearly, as shown.

```
int x[3][2] = { {84, 54},                          /* Method 1 */
                {43, -12},
                {59, 93} };

int x[3][2] = {84, 54, 43, -12, 59, 93},           /* Method 2 */
```

CamScanner

which results in the assignment

84	54
43	−12
59	93

Regardless of the style you use, make sure that the data are listed in row-by-row order because that is how the data are inserted into the array. As with one-dimensional arrays, if not enough data values are specified, the remaining elements of the array are set to 0.

A three-dimensional array is declared with three pairs of square brackets. For example, the following are all valid array declarations.

```
float records[A][B][C];
int enroll[course][campus][year];
int enroll[6][4][3];
double table[50][60][70];
```

The second and third definitions of arrays are similar except that the array size is defined by constants in the third array. In the third array first subscript may take on values from 0 to 5; the second, from 0 to 3; and the third, from 0 to 2. A total of 6×4×3, or 72, type `int` values may be stored in the array `enroll`. All three subscripts must be specified in each reference to array `enroll` in order to access a single number (*e.g.*, `enroll[2][4][1]`). This array can be visualised in the form of three-dimensional table as shown in Fig. 10.2.

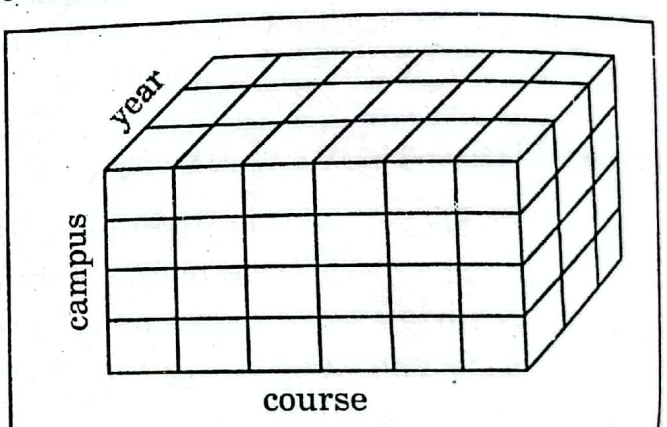


Fig. 10.2. A three-dimensional array

Example 10.5

Consider an elementary problem of *matrix addition*: adding the corresponding elements of two arrays and storing the sums in a third array. All three arrays must have the same size and data type. We generally use a doubly nested `for` loop to add matrices. The first (outer) loop supplies the *row* index and the second (inner) loop supplies the *column* index. Assume that the input matrices are named `a` and `b` and the resulting output matrix is named `c`. Here is a program:

```
/* program 10-13.cpp */
#include<stdio.h>
#include<conio.h>
#define ROWS 3
#define COLS 2

void main(void)
{
  clrscr();
```

CamScanner

```
   int a[ROWS][COLS] = { {23,17},{14,34}, {3,19} };
   int b[ROWS][COLS] = { {12,16},{42,26}, {63,20} };
   int c[ROWS][COLS];
   int row, col;

   for(row=0; row < ROWS; row++)      /* perform matrix addition */
      for(col=0; col < COLS; col++)
       c[row][col]=a[row][col]+b[row][col];
   printf("\nMatrix 1 + Matrix 2 = \n");   /* output a heading */
   for(row=0; row < ROWS; row++)      /* output resulting matrix */
      {
      printf("\nRow #%d:    ", row+1);
      for(col=0; col < COLS; col++)
       printf("%4d", c[row][col]);
      }
   getch();
}
```

Use a desk check to verify that the output matrix is indeed the sum of the two input matrices.

Example 10.6

The transpose of matrix is formed by interchanging the rows and columns of the original matrix. For instance,

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

then its transpose is

$$A = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

and if the matrix has equal number of rows and columns, it is called a square matrix. The program 10-14 reads a square matrix and prints its transpose.

```
/* program 10-14.cpp */
#include<stdio.h>
#include<conio.h>

void main(void)
{
  clrscr();
  int a[10][10];
  int n,i,j,holder;

  printf("\nEnter the size of square matrix (maximum 10): ");
  scanf("%d", &n);
```

CamScanner

```
   printf("Enter each row in one line:\n");
   for(i=0; i < n; i++)
      {
      for(j=0; j < n; j++)
        scanf("%d", &a[i][j]);
      }
   for(i=0; i < n-1; i++)
      for(j=i+1; j < n; j++)
      {
       holder=a[i][j];
       a[i][j]=a[j][i];
       a[j][i]=holder;
      }
   printf("\nThe Transpose Matrix is\n");
   for(i=0; i < n; i++)
      {
      for(j=0; j < n; j++)
       printf("%3d", a[i][j]);
      }
   getch();
}
```

CamScanner

Exercise

10.1. What is an array?

10.2. Can an array store more than one type of variable?

10.3. How do you declare an array?

10.4. What is the difference in meaning between `x3` and `x[3]`?

10.5. The simple variable `x` can take on different values, such as 5 and 12. Why is `x` not called an array?

Ans. The value of any simple variable can change overtime. Yet at any given time, only one value can be associated with a simple variable. There is a one-to-one correspondence between its name and its current value. By contrast, an array has a one-to-many correspondence between its name and its current values.

10.6. For the declaration

```
char num[7];
```

how many memory cells are allocated for data storage? What type of data can be stored there? How does one refer to the initial array element? To the final array element?

10.7. What is the relationship between the value of the subscript and the value of the array element?

10.8. If an array is declared as `float x[10]`, does that mean that the indices are also floating point?

Ans. No! The data type of an array is the *kind* of data that can be stored in the individual cells of the arrays. The index of any array is *always* an integer in the range of `0` to `(array size - 1)`.

10.9. Initialise a five-element array to the values 10, −43, 32, 93, 76.

10.10. Declare one array for storing the square roots of the integers from 0 through 10 and a second array for storing the cubes of the same integers.

10.11. Write a program that reads n integers and prints the smallest along with its index in the list.

10.12. What happens if an array is referenced outside of its bounds?

Ans. Out-of-bounds references are allowed but they are not checked and the results are unpredictable! As the programmer you are responsible for ensuring that all arrays are correctly referenced.

10.13. If an array has been defined this way:

```
float cost[MAX];
```

is the following a good way to read values into all the elements of the array?

```
for(j=0; j <= MAX; j++)
  scanf("%f", cost[j]);
```

10.14. How do we set all the cells of a 100-element array to 1?

CamScanner

Ans. Assume that the array j has been declared, use a for loop:

```
for(index=0; index < 100; index++)
   j[index]=1;
```

10.15. How can you change array data?

Ans. As with simple variables, you can change array data with an input statement such as scanf or with an assignment statement as in list[4]=9; As the programmer you must be sure to supply a well-defined subscript for the array.

10.16. Is this a correct way to initialise a one-dimensional array?

```
int array = {1, 2, 3, 4};
```

10.17. Write a program that reads two integer arrays, x and y, having 4 elements each and prints the sum of the products as given below.

```
sum = x[1]×y[1] + x[2]×y[2] + x[3]×y[3] + x[4]×y[4]
```

10.18. Explain what is wrong with this program?

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

void main(void)
{
  clrscr();
  int count[10];
  int index;
  for(index = 0; index < 100; index++)
     {
      printf("Enter a number %d: ", index);
      scanf("%d", &count[index]);
     }
.
.
.
}
```

10.19. A problem calls for a list of numbers to be input. The user decides how many numbers, as shown below:

```
int size, list[size];
printf("\nEnter the size of the list");
scanf("%d", &size);
```

What is wrong with this program segment?

Ans. The size of a declared array must be a constant. Because the value of size is not fixed at compile time, the array is not described correctly and the declaration generates a syntax error. We have to estimate a reasonable maximum size for the array and use that number in the array declaration.

10.20. Write a program that reads ten numbers entered by the user and reports if any of them match.

10.21. Change the sorting program shown in program 10-8 so that it sorts data of type float.

10.22. What are the two- and three-dimensional arrays? How do you declare them?

CamScanner

Objectives and MCQs

10.1. An array is a collection of variables of
(a) different data types scattered throughout memory.
(b) the same data type scattered throughout memory.
(c) the same data type placed next to each other in memory.
(d) different data types placed next to each other in memory.

10.2. A simple variable differs from an array variable because a simple variable
(a) can have only one value during the entire execution of a program.
(b) can have many values at one time during the execution of a program.
(c) can have many values but only one value at any given time.
(d) has a many-to-one relationship between the variable and its associated data.

10.3. Individual elements of an array are referred to by means of an index or subscript.

10.4. Is this a correct array declaration?

```
int max(7);
```

10.5. Which element of the array does this expression reference?

```
list[2]
```

10.6. What is the difference between the 5s in these two expressions?

```
int temps[5];
temps[5] = 84;
```

(a) first is particular element, second is type.
(b) first is size, second is particular element.
(c) first is particular element, second is array size.
(d) both specify array elements.

10.7. The following declares a `float` array whose elements are numbered 0 through 10:
(a) `float rainfall[10];`
(b) `float rainfall[11];`
(c) `float rainfall[9];`
(d) `float rainfall[0:10];`

10.8. What does this combination of statements do?

```
#define MAXSIZE 10
int list[MAXSIZE];
```

(a) make `MAXSIZE` a subscript.
(b) makes `MAXSIZE` a variable of type `float`.
(c) makes `list[]` an array of type `MAXSIZE`.
(d) makes `list[]` an array of size `MAXSIZE`.

CamScanner

10.9. The following sets all cells of a five-element array to the value of 10?

(a) `int x = {10, 10, 10, 10, 10};`

(b) `int x[] = {5*10};`

(c) `int x[5] = {5*10};`

(d) `int x[] = {10, 10, 10, 10, 10};`

10.10. What will happen if you try to put so many elements into an array when you initialise it that the size of the array is exceeded?

(a) nothing.

(b) possible system malfunction.

(c) error message form the compiler.

(d) other data may be overwritten.

10.11. What will happen if you put too few elements in an array when you initialise it?

(a) nothing.

(b) possible system malfunction.

(c) error message from the compiler.

(d) unused elements will be filled with 0s or garbage.

10.12. What will happen if you assign a value to an element of an array whose subscript exceeds the size of the array?

(a) the element will be set to 0.

(b) nothing, it is done all the time.

(c) other data may be overwritten.

(d) possible system malfunction.

10.13. A one-dimensional array is sometimes called a contiguous list.

10.14. When you pass an array as an argument to a function, what is actually passed?

(a) the address of the array.

(b) the values of the elements in the array.

(c) the address of the first element in the array.

(d) the number of elements in the array.

10.15. True or false: A function operates on an integer array passed to it as an argument by placing the values of that array into a separate place in memory know only to the function.

10.16. True or false: Individual array elements cannot be passed easily to a function.

10.17. True or false: To pass a whole array to a function, you specify the array name.

10.18. True or false: Address of a floating-point variable is always a whole number.

10.19. To pass an individual element of an array to a function, you must specify

(a) the data type of the array, its name, and the element number.

(b) the name of the array and the element number as two separate arguments.

(c) the name of the array and the element number as one argument.

(d) just the element number.

10.20. Assuming the array declaration `float values[MAX_VALUE];` which statement will pass the whole array to function `calc_max`?

CamScanner

(a) `calc_max(float values[]);`

(b) `calc_max(values[MAX_VALUE]);`

(c) `calc_max(float values[MAX_VALUE]);`

(d) `calc_max(values);`

10.21. True or false: Multi-dimensional arrays are sometimes called arrays of arrays.

10.22. True or false: Up to three dimensions for an array are allowed in C.

10.23. True or false: Multi-dimensional arrays allow multiple data types.

10.24. True or false: In a two-dimensional array, the first dimension always refers to the row and the second dimension always refers to the column.

10.25. Two-dimensional arrays are sometimes called matrices or tables.

10.26. The following declares a two-dimensional double array with four rows and three columns.

(a) `double x[3,4];`

(b) `double x[4,3];`

(c) `double x[3][4];`

(d) `double x[4][3];`

10.27. Can you initialise a two-dimensional array this way?

```
int array[3][3] = { {1, 2, 3},
                    {4, 5, 6},
                    {7, 8, 9} };
```

10.28. The following initialises a two-dimensional array to all ones?

(a) `int x[3][2]={7*1};`

(b) `int x[3][2]={1,1,1,1,1,1};`

(c) `int x[3][2]={1};`

(d) `int x[3][2]={{1,1}},{{1,1}},{{1,1}};`

10.29. Which statement initialises the array so that it looks like the pictures?

7	7	6	6	6
7	6	6	6	6

(a) `x[5][2]={{7,7}, {7,6}, {6,6}, {6,6}, {6,6}};`

(b) `x[2][5]=({7,7,6,6,6}, {7,6,6,6,6});`

(c) `x[5][2]={{7,7,6,6,6}, {7,6,6,6,6}};`

(d) `x[2][5]={{7,7,6,6,6}, {7,6,6,6}};`

CamScanner

CHAPTER 11

STRINGS

A string is an array of characters. It is used in programming for storing and manipulating text, such as words, names and sentences. When you declare a string, you give it a name and specify the maximum number of characters it can hold. The last array element, however, is reserved for the null terminator (\0), so you need to declare the array as one element larger than the maximum number of actual characters.

We have already used strings with the `printf` and `scanf` function, and strings can also be used with array variables as discussed in chapter 6. Although a string data type is not defined in C, we can think of a `char` array as a string. *Turbo C++* let us input and output a character array as one entity—a string. But each character is really a separate array element, as shown in Fig. 11.1. For instance, you can input a string and then display its individual characters, like this:

```
/* program 11-1.cpp */
#include<stdio.h>
#include<conio.h>

void main(void)
{
  clrscr();
  char name[10];
  int index;
  printf("Enter your name: ");
  scanf("%s", name);
  for(index = 0; index < 10; index++)
      printf("%c\n", name[index]);

  getch();
}
```

CamScanner

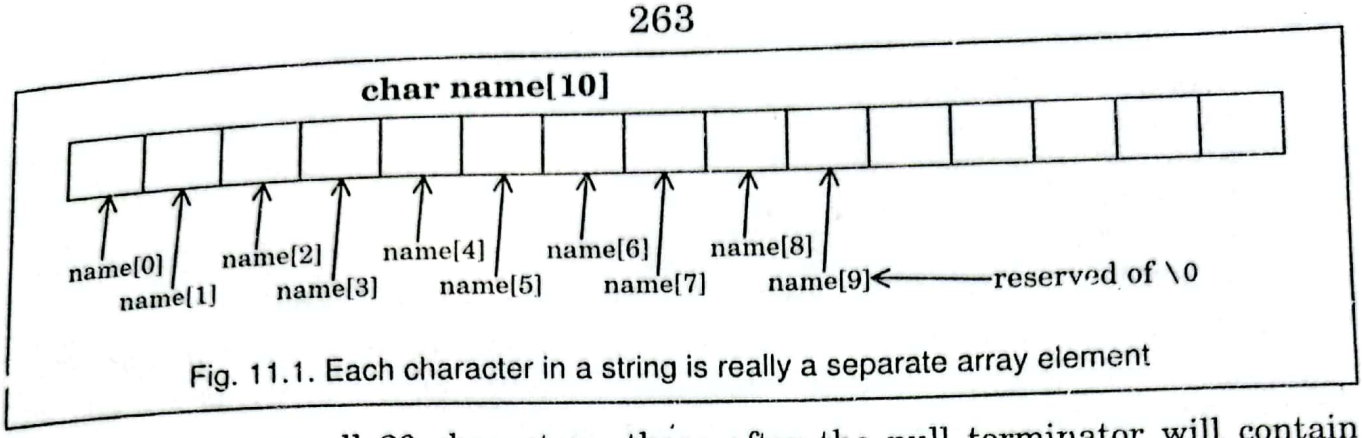


Fig. 11.1. Each character in a string is really a separate array element

If you do not enter all 20 characters, those after the null terminator will contain unknown values. Do not overflow the array that holds the string.

String Initialisation

Like the arrays, strings can also be initialised. Since a string is an array of characters, we can initialise one in exactly that way. For example:

```
char message[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

You can also have a simpler choice by giving the following declaration:

```
char message[6] = "Hello";
```

A slightly more simper initialisation is also allowed:

```
char message[] = "Hello";
```

In the absence of an explicit size constant, sufficient storage is allocated to hold the specified string, including the `'\0'` character.

String Input/Output

The easiest way to input strings is by using the *Turbo C++* library function `gets` (for *get string*). The `gets` library function takes one argument, the string variable in which the data will be stored:

```
char name[51];
printf("Enter your name: ");
gets(name);
```

Notice that the string variable is declared as 51 characters so that the full 50-character line including the null byte can be stored.

The `gets` library function has an associated function for output called `puts` (for *put string*). The `puts` function also takes one string argument. (This is explained in greater detail in Chapter 6.) Unlike `printf`, `puts` automatically advances the output to the following line after displaying it.

You can also use the `scanf` function for string input by means of the `%s` format specifier. Note these two restrictions: First, the input string cannot have blanks because `scanf` interprets a blank to mean the end of a particular input value. As a result, you cannot conveniently input a person's first and last name with `scanf` unless you specify two string variables or use two `scanf` functions.

CamScanner

```
char first[21], last[21];
printf("\nEnter your first name: ");
scanf("%s", first);
printf("\nEnter your last name: ");
scanf("%s", last);
printf("\nYour full name is: %s %s", first, last);
```

Second, notice that the & (*address of*) operator is *not* used in the `scanf` function. This is because the name of a string variable, like the name of any array, is actually an address and is already in the correct form for `scanf`.

Built-in String Functions

Turbo C++ compiler includes special functions for working with strings. While you can perform the same functions by writing your own instructions, the built-in functions are more efficient. Table 11.1 lists some typical string functions from `<string.h>` found in *Turbo C++* libraries.

Table 11.1. Some Typical String Functions

Function	Description
`strcat`	Adds the character of one string to the end of another.
`strchr`	Returns the position of a specified character in the string.
`strcmp`	Compares two strings alphabetically.
`strcmpi`	Compares two strings; not case sensitive.
`strcpy`	Copies one string, or string constant, to another.
`strncpy`	Copies specified characters from one string to another.
`strcspn`	Returns the position of a character in the string from a specified character set.
`strdup`	Duplicates a string.
`strlen`	Calculates the string length.
`strlwr`	Converts a string to lowercase.
`strncat`	Appends specified characters from one string to another.
`strncmp`	Compares specified characters of two strings.
`strcmpi`	Compares two strings, not case sensitive.
`strncpy`	Copies specified characters from one string to another.
`strnset`	Changes specified characters in a string to another character.
`strrev`	Reverses the characters in a string.
`strstr`	Finds one string within another.
`strupr`	Converts a string to uppercase.

As an example, we will look at some string-handling functions and see how you can perform them yourself.

CamScanner

1. Comparing two Strings: `strcmp(string1, string2)`

The strcmp function compares `string1` to `string2` and returns an integer value to show the status of the comparison. If the function returns a 0, then the two strings are identical. A value of less than 0 shows that `string1` is (according to its alphabetic ordering) less than `string2`. A value greater than 0 shows that `string1` is greater than `string2`.

```
char string1[] = "Syed Asif";
char string2[] = "Syed Asif";
char string3[] = "Muhammad Asif";
char string4[] = "Muhammad Iqbal";
printf("%d%d%d", strcmp(string1, string2),
                 strcmp(string1, string3),
                 strcmp(string1, string4));
                                  /* Output 0, >0, <0 */
```

Note that when the strings are not equal your concern is only whether the result is positive or negative and not the exact value of the function.

2. Determining String Length; `strlen(string)`

The length of a string is not necessarily the same as the size of the array. For instance, you may declare a 40-element array called name but enter a name of less than 40 characters. *Turbo C++* compilers use the `strlen()` function to determine the actual number of characters in the string, as shown here:

```
char string1[41];
char string2[] = "Syed Asif";
printf("%d %d", strlen(string1), strlen(string2));
                                  /* Outputs 0, 9 */
```

3. Combining Strings; `strcat(string1, string2)`

The `strcat` function *concatenates* or joins `string1` and `string2`. A copy of `string2` is put at the end of `string1`. Make sure that `string1` is long enough to hold the resulting string.

If `string1` is not long enough to hold the result, the extra characters will be copied into the adjacent memory locations following those associated with `string1` The extra characters will overwrite any legitimate data, causing unpredictable results!

```
char string1[41] = "abc";
char string2[] = "def";
strcat(string1, string2);
puts(string1);                    /* Output "abcdef" */
```

4. Assigning Strings; `strcpy(string1, string2)`

The strcpy function copies `string2` into `string1`. Again, make sure that `string1` is long enough to hold the resulting string.

```
char string1[41];
char string2[] = "xyz";
strcpy(string1, string2);
puts(string1);                    /* Output "xyz" */
```

CamScanner

5. Searching String; strchr(string, ch)

The strchr function searches string for the first occurrence of ch. We might expect this function to return the index of the search character in the string. Instead the function returns the *memory location* (address) of the character. Significantly, the address is set to the predefined value NULL if the character is *not* in the string.

```
char string[9] = "Syed Asif";
char search    = 'i';
if (strchr(string, search) != NULL)
   printf("\nSearch character found");
else
   printf("\nSearch character not found");
```

As with strcmp, your only concern is whether or not the character was found and not where in the string the character resides.

Example 11.1

This program request input of two strings, then demonstrates the four string functions with them.

```
/* program 11-2.cpp */
#include<stdio.h>
#include<conio.h>
#include<string.h>

void main(void)
{
  clrscr();
  char str1[80], str2[80];
  int i;

  printf("Enter first string: ");
  gets(str1);
  printf("Enter second string: ");
  gets(str2);

  /*see how long the strings are*/
  printf("\n\n");
  printf("%s is %d chars long\n", str1, strlen(str1));
  printf("%s is %d chars long\n", str2, strlen(str2));

  /*compare the strings*/
  i = strcmp(str1, str2);
  if(i==0)
     printf("The strings are equal\n");
  else if(i<0)
     printf("%s is less than %s\n", str1, str2);
  else
     printf("%s is greater than %s\n", str1, str2);
```

CamScanner

```
    /*concaterate str2 to end of str1 if there is enough room*/
    if(strlen(str1)+strlen(str2) < 80)
       {
        strcat(str1, str2);
        printf("%s\n", str1);
       }

    /*copy str2 to str1*/
    strcpy(str1, str2);
    printf("%s %s\n", str1, str2);

    getch();
}
```

One of the outputs will look like this:

```
Enter first string: Syed Asif
Enter second string: Muhammad Iqbal

Syed Asif is 9 chars long
Muhammad Iqbal is 14 chars long
Syed Asif is greater than Muhammad Iqbal
Syed AsifMuhammad Iqbal
Muhammad Iqbal Muhammad Iqbal
```

Example 11.2

This program request input of a string, then prints it diagonally.

```
/* program 11-3.cpp */
#include<stdio.h>
#include<conio.h>
#include<string.h>

void main(void)
{
  clrscr();
  char name[30];
  int index;

  printf("Enter your name: ");
  gets(name);
  for(index=0; index<strlen(name); index++)
  {
     gotoxy(index+5, index+5);
     printf("%c", name[index]);
  }

  getch();
}
```

CamScanner

The output of the program will be:

```
Enter your name: Syed Asif

    S
     y
      e
       d

          A
           s
            i
             f
```

The size of the string is provided by the function `strlen()`, which returns the length of the value of its string argument, not counting the `'\0'` character.

The `gotoxy()` function moves the cursor to certain specific location (`x, y`). This function works under `<conio.h>` header file. In the function `x` denote the column number and `y` denote the row number. The function in the above program moves the cursor on the next column and next row each time the loop is executed. This prints the entered name diagonally.

Array of Strings

You can have an array of strings, just as you can have an array of any other data type. But an array of strings is really an array of arrays of characters. An array whose elements are arrays is called a two-dimensional array.

You can picture a two-dimensional array as a spread sheet with rows and columns. You must define such an array with two subscripts—one specifies the number of rows in the spreadsheet; the other represents the number of columns. The following instruction declares an array of 5 rows and 20 columns, with 100 integer variables in all:

```
int table[5][20];
```

Visualize each element as an integer value in its own cell of the 5×20 spreadsheet. The element `table[0][0]` is in the top-left cell, and element `table[0][1]` is in the cell to its right.

When you declare an array of strings, you must also use two subscripts. The first subscript represents the maximum number of strings in the array; the second represents the maximum length of each string. So if you declare

```
char names[5][20];
```

you can have five names up to 19 characters each.

To initialise the array of names, consider the following declaration:

```
char names[5][20] = {"Abbas", "Qasim", "Azam", "Umar", "Abbas"};
```

The names in quotes are already a one-dimensional array, therefore, we do not need braces around each name as we did for two-dimensional numeric array. We do need braces around all the names because this is an array of strings.

CamScanner

Example 11.3

The program in this example asks you to type in your name. When you do, it checks your name against a master list to see if your name is in the list or not. It displays the message accordingly.

```
/* program 11-4.cpp */
#include<stdio.h>
#include<conio.h>
#include<string.h>

void main(void)
{
  clrscr();
  int index, enter;
  char name[20],
       list[5][20] = {"Abbas", "Qasim", "Azam", "Umar", "Abbas"};

  printf("Enter your name: ");
  gets(name);                                          /*get name*/
  for(index = 0; index < 5; index++)                   /*go thru list*/
     if(strcmp(&list[index][0], name)==0)              /*if match*/
      enter = 1;                                       /*set flag*/
     if(enter == 1)                                    /*if flag set*/
      printf("\nYour Name is in the list");            /*one response*/
     else                                              /*otherwise*/
      printf("\nYour Name is not in the list");        /*different one*/

  getch();
}
```

The run of the program shows:

1st Run

```
Enter your name: Arif
Your Name is not in the list
```

2nd Run

```
Enter your name: Umar
Your Name is in the list
```

You will notice in the program, that we used a function `strcmp` which is available in the header file `<string.h>`. Accordingly `<string.h>` file is included in the beginning of the program. This function will compare the string entered from the keyboard with the strings already available in the program. If the comparison is true, then the message *"Your Name is in the list"* will be displayed otherwise message *"Your Name is not in the list"* will be displayed.

Example 11.4

One common use of strings is to support a *command-based interface*. Unlike a menu, which allows the user to make a selection, a command-based interface displays a

CamScanner

prompting message, waits for the user to enter a command, and then does what the command requests. Many operating systems use command-line interfaces, for example. The following program is similar to the program 7-19. It allows the user to add, subtract, multiply, or divide, but does not use a menu. Instead, it uses a command-line interface.

```
/* program 11-5.cpp */
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<stdlib.h>

void main(void)
{
  clrscr();
  char command[20], temp[20];
  int i,j;

  printf("Operation? ");
  gets(command);

  printf("Enter first number: ");
  gets(temp);
  i = atoi(temp);

  printf("Enter second number: ");
  gets(temp);
  j = atoi(temp);

  if(!strcmp(command, "add"))          /*now perform operation*/
     printf("Answer is %d\n", i+j);
  else if(!strcmp(command, "subtract"))
     printf("Answer is %d\n", i-j);
  else if(!strcmp(command, "divide"))
     {
     if(j <= 0)
     printf("Second number should be greater than zero.");
     else
     printf("Answer is %d\n", i/j);
     }
  else if(!strcmp(command, "multiply"))
     printf("Answer is %d\n", i*j);
  else
     printf("Unknown Command.");

  getch();
}
```

CamScanner

One of the outputs will look like this:

```
Operation? multiply
Enter first number: 32
Enter second number: 5
Answer is 160
```

Notice that this example also introduces another of C's standard library functions: `atoi()`. The `atoi()` function returns the integer equivalent of the number represented by its string argument. For example, `atoi("50")` returns the value 50. The reason that `scanf()` is not used to read the numbers is because it is incompatible with `gets()`. The `atoi()` function uses the header file `<stdlib.h>`.

CamScanner

Exercise

11.1. Is `'C'` the same as `"C"`?

Ans. No! `'C'` is a character constant and is not equal to the string constant `"C"`.

11.2. Where is the `'\0'` character in a string?

Ans. *Turbo C++* supplies `'\0'` if you provide space for it in the array. The `'\0'` is important for string processing because it indicates the logical endpoint of a given string.

11.3. How do you compare two strings?

11.4. How do you assign a value to a string variable?

11.5. Assuming the following initialisation:

```
char string[] = "Pakistan";
```

how would you refer to the string `"tan"` (the last three letters of the string)?

11.6. What is the wrong ~~with~~ the following declaration?

```
name = "Azeem";
```

11.7. Output the third character from a 40-character string.

11.8. Output the third string from a 40-string array of strings.

11.9. Copy the alphabetically smaller of two strings into the third position of a string array.

11.10. Copy the fifth string from an array of strings to a single string variable.

11.11. Copy the seventh string from an array of strings to the ninth string of a string array.

11.12. Write a program to find whether a character string is a palindrome. A palindrome reads the same whether it is read from left to right or right to left [ABBA is a palindrome].

11.13. What expression would you use to find the length of the string name?

11.14. Write a program to input word and then print number of vowels, with appropriate message. For example if you enter word *Computer* it should print following message:

```
Number of vowels in word Computer is 3.
```

CamScanner

Objectives and MCQs

11.1. A string is
(a) a list of character.
(b) a collection of characters.
(c) an array of characters.
(d) an exaltation of characters.

11.2. True or false: *Turbo C++* does not distinguish between `'C'` and `"C"`.

11.3. What is the following expression?
`"Karachi\n"`
(a) a string variable.
(b) a string array.
(c) a string constant.
(d) a string of characters.

11.4. True or false: Strings are stored as arrays of characters.

11.5. The string `"I like programming"` needs 19 memory cells.

11.6. String functions are accessible through the `<string.h>` header.

11.7. An array of strings is actually a character array.

11.8. Which is more appropriate for reading in a multiword string?
(a) `gets()`
(b) `printf()`
(c) `scanf()`
(d) `puts()`

11.9. The statement `strlen(string)`
(a) returns the length of the string including the null character.
(b) returns the length of the string excluding the null character.
(c) returns the length of the string as specified in its declaration.
(d) returns the declared length of the string minus 1.

11.10. The statement `strcpy(string1, string2)`
(a) adds `string2` to the end of `string1`.
(b) adds `string1` to the end of `string2`.
(c) overwrites `string1` with `string2`.
(d) overwrites `string2` with `string1`.

11.11. The statement `strcmp("Syed", "Asif")`
(a) is not legal because `"Syed"` and `"Asif"` are not string variables.
(b) returns the value 0 because the strings are not equal.
(c) returns the value −1 because the strings are not equal.
(d) returns a positive value because `"Syed"` is greater than `"Asif"`.

CamScanner

11.12. True or false: The `gets` function is accessed through `<string.h>`.

11.13. The statement `strchr("Asif", 'i')`

(a) returns 0 because `'i'` is contained in `"Asif"`.

(b) returns a negative value because `'i'` is contained in `"Asif"`.

(c) returns a positive value because `'i'` is contained in `"Asif"`.

(d) is a syntax error because `"Asif"` is not a string variable.

11.14. What does this program fragment display?

```
char x[80] = "gorilla";
char y[80] = "giraffe";
strcpy(x, y);
printf("%s %s\n", x, y);
```

(a) gorilla giraffe

(b) giraffegorilla gorilla

(c) gorilla gorilla

(d) giraffe giraffe

11.15. What does this program fragment display?

```
char x[80] = "gorilla";
char y[80] = "giraffe";
strcat(x, y);
printf("%s %s\n", x, y);
```

(a) gorillagiraffe giraffe

(b) giraffegorilla gorilla

(c) gorilla gorilla

(d) giraffe giraffe

CamScanner

CHAPTER 15

DATA MANAGEMENT SYSTEMS

Manual data collection and examination are time-consuming and prone to a high degree of error. Today, rather than searching through piles of paper to locate a file or to find specific data in several files (see Fig. 15.1), we have database software to perform these tasks very efficiently. Instead of storing data in paper files, database software stores data electronically on a disk. Instead of requiring an army to search through a file cabinet for one or more files, database software makes it easy to search through all files almost instantaneously. Instead of having to scan an entire file for specific data, database software will do it for you.

A *database* is a collection of related files within an organisation. An organisation may have more than one database consisting of related files. A database usually includes more than one file, but may contain only one. The fields within a database can be joined or otherwise interrelated for query purposes.

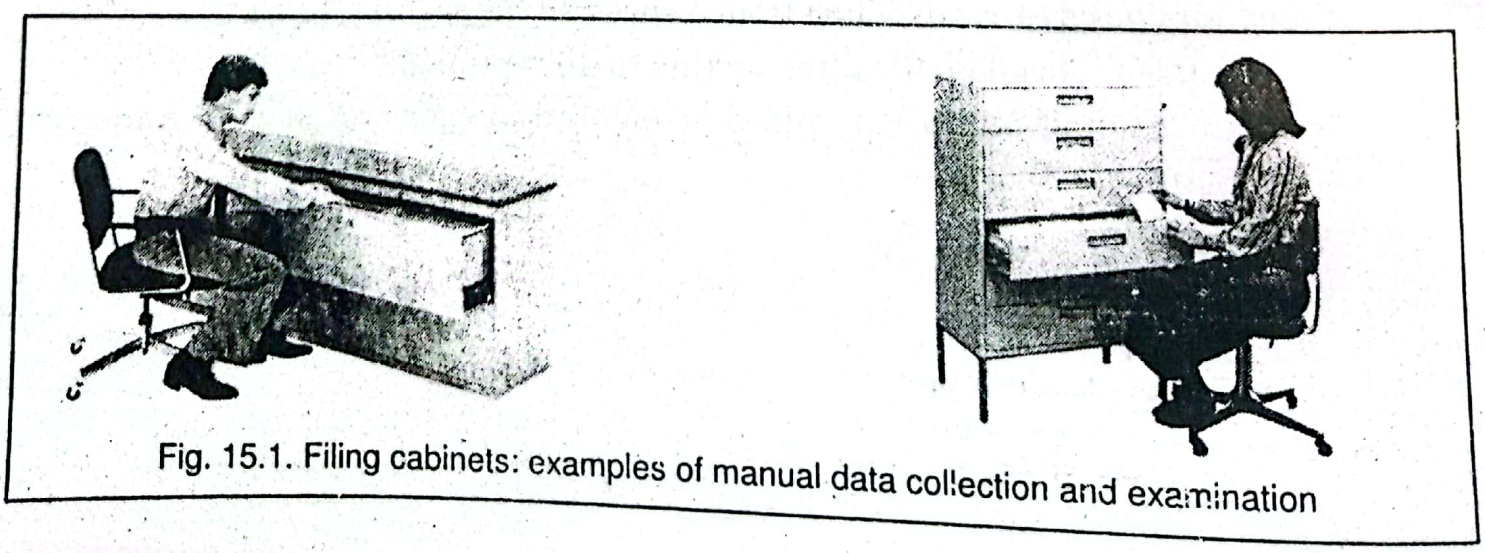


Fig. 15.1. Filing cabinets: examples of manual data collection and examination

CamScanner

As an example, a school may have a database containing the names and authors of all the books in its library, a short description of each book, a list of books checked out, a list of books on order, and a list of students who are eligible to borrow books. Each list is to be in a separate file. This composite group of files makes up the library's database.

The same school might have another database containing information about all its employees, including salaries, dates hired, home addresses, year-to-date payments, year-to-date deductions and any other information needed to implement the school's payroll application. It is likely that the year-to-date payroll and tax-deduction information would be in one file, while the rest of the relevant employee information would be in another file. Having the two files each containing different payroll information is more efficient because several different functional groups can access employee data—accounting and payroll, as well as the personnel department—and they each may need different data. If, instead, each record contained *all* employee data, the file would be very long, and its retrieval time extensive. Such files would also be far more difficult to access in a multi-user environment.

How does the database management system actually work? To understand this, first we need to consider some basic definitions:

Character

The basic building block of data is a *character*. A character—sometimes called byte—consists of letters (A, B, C, ..., Z or a, b, c, ..., z), numeric digits (0, 1, 2, ..., 9) or special characters (+, −, /, *, #, $. ...).

Entity

Any object or event about which someone chooses to collect data is an *entity*. An entity may be a person, place, event, object, or concept in a real world that we wish to represent in database.

Some examples of each of these types of entities follow:

Person:	employee, student, patient, customer.
Place:	state, region, country.
Object:	machine, building, country.
Event:	machine breakdown, electricity failure.

Entity can be physical or conceptual, *e.g.* student is a physical entity while dreams are conceptual entities.

Field or Attribute

A *field* is an item consisting of one or more logically related characters. It is some characteristic or quality of an entity that is treated as a single unit in data processing. For example, if we are processing employees data of a company, we may have an employee code field, an employee name field, an employee pay field, a tax-rate-deduction field etc. A field is sometimes called an *attribute* or a *data element*.

So each column of the table represents a field, which is a type of data item. The most commonly used field types are text fields, numeric field, currency field, logical field, date field, and memo field etc. (see table 15.1), but other specialised types are necessary in some situation.

CamScanner

Table 15.1. Field Description

Text	Text (character) fields store data that are not manipulated arithmetically. You can, however, sort, index, or compare on the fields.
Numeric	Numeric (number) fields, which store integer numbers and numbers that contain decimal points, are those that must be arithmetically manipulated.
Currency	Many database packages have a special field type for numeric data that are to be output in a currency format, with dollar signs, decimal points, and commas.
Logical	Logical (Yes/No) fields often store a single character of data—for instance, a "Y" (for "yes") and an "N" (for "no").
Date	Date fields store dates, provided that they are in the format DD/MM/YY (such as 29/07/1999) or something very similar. You can sort, index, and subtract date fields.
Memo	Memo fields are used to store comment information. Tney cannot be arithmetically manipulated or compared, but they can be edited and output.

The Key Field

An important concept in database is that of the *key field*. It is a field on which records can be sorted in some logical order for fast access. The key field contains a unique identifier. For example, employee records can be sorted on their employee numbers or N.I.C. numbers, both of which can be the key field. The primary characteristic of the key field is that it is unique. Thus, numbers are clearly preferable to names as key fields because there are many people with common names like Mohammad Hussain, Javeed Iqbal, Ashraf Ali, or Mohammad Saad, whose records might be confused.

Primary Key and Secondary Key

In some cases, you want to sort records on the basis of more than one key. For example, let's say that you want to sort your mailing list first by postal code and then alphabetically by the name. In other words, records are to be alphabetised by the name within each postal code. The postal code field is the *primary key* and the name field is the *secondary key*. The primary key is the field you sort on first and the secondary key is the field you sort on second. Virtually all database management systems can sort on the basis of at least two keys. Many database programs let you have as many keys as there are fields in a record.

Another decision you must make before sorting a database is the order. There are two orders in which records can be sorted: *ascending order* and *descending order*. If the sort key contains numeric values, then sorting in ascending order arranges the numbers from smallest to largest. If the sort key contains text, then sorting in ascending order arranges records alphabetically, from A to Z. Descending order goes from largest to smallest for numbers and from Z to A for text. By default, most database programs assume that you want ascending order unless you specify otherwise.

Record

A *record* is a collection of related fields. An employee record would be a collection of fields of one employee. These fields would include the employee code field, an

CamScanner

employee name field, an employee pay field, a tax-rate-deduction field, and so forth. A record is sometimes called an *entity*, the name for any object or concept about which an organisation wishes to store data.

File

A *file* is a collection of related records that are treated as a unit. For example, a collection of all employee records for one company would be an employee file (see Fig. 15.2). Similarly, a collection of all inventory records for a particular company forms an inventory file.

Database Files

EMPLOYEES FILE

L_Name	F_Name	Code No.
Khan	Imran	440-125
Bashir	Babar	328-486
Durani	Anis	183-839

PAYROLL FILE

Emp. code	Net pay	Deduction
2001-7376	7800	370
2001-2874	9300	460
2001-2837	5950	132

CUSTOMER FILE

C_No.	C_Name	C_Phone
38943	Samad	4628349
24656	Ahmad	9792734
49570	Anis & co.	7362842

ORDER FILE

C_No.	Stock No.	Quantity
83754	12-2455	14
28437	23-3747	43
82743	10-3846	9

Each field is one piece of data

Each record is made up of related fields

Each file is made up of related records

Each database is made up of related files

Fig. 15.2. A database consists of related files; each file in turn contains records which are subdivided into fields of data

Database

A database is a unified collection of related files that can be retrieved and manipulated to produce information. A company database might include files on all past and current employees in all departments. Characters, fields, records, files, and databases form a hierarchy of data storage. Fig. 15.3 summarises the data storage hierarchy used by computer-based processing systems. Characters are combined to make a field, fields are combined to make a record, records are combined to make a file, and files are combined to make a database.

The most common example of a database is a telephone directory.

CamScanner

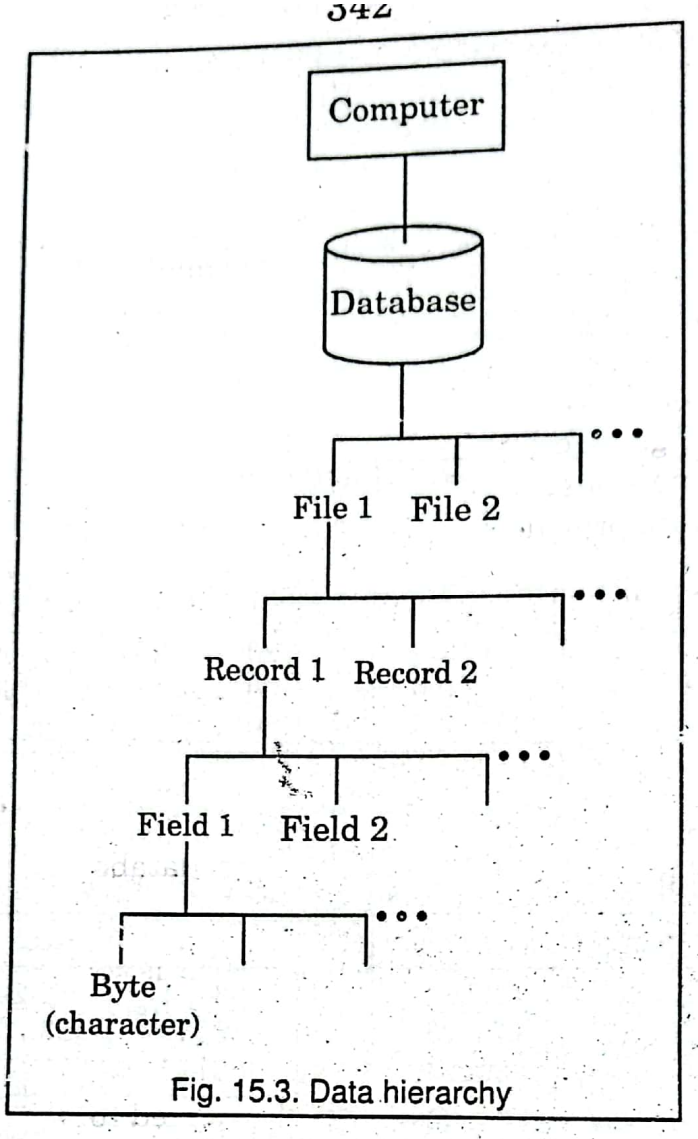


Fig. 15.3. Data hierarchy

Database Management System

A database management system (DBMS) is a software package that allows users to create, maintain, and manipulate a database to produce useful information. The DBMS software represents the interface between the user and the computer's operating system and database. In recent years, database management systems (DBMS) have established themselves as the primary means of data storage for information systems ranging from large commercial transaction processing applications to PC-based desktop applications. Oracle, MS Access and dBASE are examples of database management systems.

Advantages of a DBMS

The advantages of databases and DBMS software are as follow:

Reduction of Data Redundancy: Instead of the same data fields being repeated in different files, in a database the information appears just once. The single biggest advantage of a database is that the *same* information is available to *different* users. Moreover, reduced redundancy lowers the expense of storage media and hardware, because more data can be stored on the media.

CamScanner

Improvement of Data Integrity: Since data are centralised, the data update at only one place reflect throughout the system automatically—thereby avoiding the error that is often introduced when the same update has to be made manually in several independent files.

More Program Independence: With a database management system, the program and file formats are the same, so that one programmer or even several programmers can spend less time to maintain files.

Increased User Productivity: Database management systems are fairly easy to use, so that users can get their requests for information answered without having to resort to technical manipulations. In addition, users do not have to wait for a computer professional to provide what they need.

Increased Security: Although various departments may share data in common, access to specific information can be limited to selected users. Thus, through the use of passwords, a student's financial, medical, and grade information in a university database is made available only to those who have a legitimate need to know.

Disadvantages of a DBMS

Although there are clear advantages to having databases, there are still some disadvantages:

Cost Issue: Installing and maintaining a database is expensive, particularly in a large organisation. In addition, there are costs associated with training people to use it correctly.

Security Issues: Although databases can be structured to restrict access, it's always possible that unauthorised users will get past the safeguards. And when they do, they may have access to *all* the files, not just a few. In addition, if a database is destroyed by fire, earthquake, theft, or hardware or software problems, it could be fatal to an organisation's business activities—unless steps have been taken to regularly make backup copies of the files and store them elsewhere.

Privacy Issues: Databases may hold information they should not and be used for unintended purposes, perhaps intruding on people's privacy. Medical data, for example, may be used inappropriately in evaluating an employee for a job promotion.

New Hardware Required: A DBMS often requires a great deal of memory and secondary storage, and accessing records can be time consuming. Thus, some users find it necessary to upgrade to a bigger, more powerful computer system—with tape or removable disk backup—after acquiring a DBMS.

Objectives of DBMS

The designers of database management systems are faced with a number of challenges. Their software must provide users with concrete and worthwhile benefits to justify the expenses and effort associated with installing it. Ideally, a DBMS should exhibit the following characteristics:

Efficient Use of Resources: A DBMS should function efficiently given the current

CamScanner

computer resources of the installation. It should not require the user to add expensive new hardware or software to make it work, and it should not waste storage space or processing time in the performance of designated tasks.

Speed: A DBMS must run fast enough to provide replies to users' questions within an acceptable time frame. Many systems are now interactive (the user types in a query and expects an immediate response), so speed is a primary objective.

Compatibility: A DBMS must interface smoothly with existing hardware, software, and data if it is to be installed successfully. If an organisation has to buy a lot of new computer equipment or new software or have all its data re-entered, it has probably chosen the wrong DBMS. Although some conversion of data and software is often required, it should be kept at a minimum.

Efficient Update of Data: A DBMS must allow for the quick and efficient addition and deletion of data records and categories. It must also be flexible enough to handle changes to or rearrangement of the data in its database.

Accessibility: DBMS users should find it convenient to interrogate the database. Most systems have an easy-to-learn query language that allows users to ask a wide variety of questions about the data without having to prepare specially designed application programs. For example, a sales analyst in a company's home office who was using Artificial Intelligence Corporation's query language REFLECT could key in a question such as "WHICH SELLERS HAVE THE LOWEST PERCENT OF QUOTA IN THE WESTERN REGION?" instead of a series of program codes.

Data Integrity: Since the database is the one place the information is stored, a very important quality of a DBMS is how well it safeguards data integrity. In case something goes wrong (a power failure, for example), the DBMS should have backup measures to ensure that valuable data remain intact. Further more, the DBMS should prevent users from making potentially disastrous mistakes. For example, requiring users to reconfirm data deletions can help to avert accidental erasures.

Privacy and Security: One of the major concerns about databases is the privacy and security of the information they contain. Medical records, bank records, credit files, and military information are but a few examples of the types of sensitive or confidential material kept in databases. A major objectives of any DBMS is to limit access to authorised users. Features such as passwords and access codes are often utilised to restrict unauthorised individuals from changing or viewing data.

Components of a Database Management System

To create, gain access to, and modify the database, the special software known as the *database management system* is required. A DBMS may have a number of components including (1) query language (2) data dictionary (3) utilities (4) report generator (5) access security, and (6) data recovery. We will consider each component in detail.

Query Language

A query language is an easy-to-use computer language for making queries to a

CamScanner

database and for retrieving information from a database. Typically, the query is in the form of a sentence or near-English typed command, using such basic words as SELECT, DELETE, or MODIFY. There are several different query languages, each with its own vocabulary and procedures. One of the most popular is *Structured Query Language*, or *SQL*.

Data definition language (DDL) have facilities for describing the structure of fields, records, files and program accessing rights. Whereas *data manipulation language* (DML) is used for accessing and updating data records and files contained within the database. The SQL contains all the features of both the data definition language and the data manipulation language. Also the American National Standards Institute's (ANSI) recommended language for relational database definition and manipulation is SQL. It is a common standard for describing and querying databases and is available with may commercial relational database products, including DB2, SQL/DS, Oracle, R:BASE, Microsoft Access, and FoxPro.

An example of an SQL query is as follows:

```
SELECT    PRODUCT-NUMBER, PRODUCT-NAME
FROM      PRODUCT
WHERE     PRICE < 1000.00
```

This query selects all records in the product file for products that cost less than Rs. 1000.00 and displays the selected records according to product number and name—for example, like this:

```
B-41      Bookshelf
E-39      Table
H-284     Chair
```

You can learn to use a query language with just several hours of instruction and practice. With this skill, you can then prepare a special report based on data from many subject areas in a few minutes instead of days or weeks.

One feature of most query languages is *query by example*. Often a user will seek information in a database by describing a procedure for finding it. However, *query by example (QBE)* make it easy for users to construct database queries without having to remember a language syntax. The user asks for information in a database by using a sample record to define the qualifications he or she wants for selected records.

Data Dictionary

Many database management systems include a *data dictionary,* which is a software component that stores the data definitions of the structure of data used in the database, just as an English dictionary contains information about the rules for using words that make up the English language. A data dictionary also maintains data integrity because as data is entered, the dictionary checks it to make sure that it conforms to the rules defined during data definition, such as field name, field size, type of data (text, numeric, date, and so on). The data dictionary may also help protect the security of the database by indicating who has right to gain access to it.

CamScanner

Utilities

The DBMS *utilities* are programs that enables users to maintain the database. This includes such tasks as:

1. Creating and maintaining the data dictionary.
2. Removing records flagged for deletion. (Most DBMSs have built-in protection schemes to prevent users from accidentally deleting records. To delete unwanted records from the database, the user must first mark, or "flag", them for deletion and then give the command to actually remove the flagged records.)
3. Monitoring performance.
4. Preventing data corruption when multiple users attempt to access the same database simultaneously.

Report Generator

The report generator aspect of DBMS software simplifies the process of generating an on-screen or printed-out report. You can specify the format of the report in advance—row headings, column headings, page headers, and so on. With a report generator, even non-experts can create attractive, readable reports on short notice.

Access Security

Access security is a feature of DBMS that allow database administrators (DBA) to specify different access privileges for different users of a DBMS. For instance, one kind of user might be allowed only to retrieve (view) data, whereas another might have the right to update data and delete records. The purpose of this security feature, of course, is to protect the database from unauthorised access and sabotage.

System Recovery

Some advanced database management systems have a system recovery feature, which enables the DBA to recover contents of the database in the event of a hardware or software failure. For instance, the feature may recover transactions that appear to have been lost since the last time the system was backed up.

The Basic Database Models

A database model is simply a model of the various ways in which the data within a database may be structured, *i.e.* its *logical structure*, and *not* the way that it is stored on any particular storage medium. Computer scientists have devoted whole careers to developing database theories, and programmers have written hundreds of database management packages. All database systems are characterised by the way they structure, organise, and manipulate data items and by the relationships among the items. These design elements are reflected differently in the various data models, of which three most popular models of organisation are:

1. Hierarchical database model.
2. Network database model.
3. Relational database model.

Which of these three is the best is a debatable issue. In fact, it depends on the situation, and, in practice most systems combine features from each of the basic approaches.

CamScanner

These three models have evolved gradually as users and computer specialists have gained experience in using database management systems. They differ in cost of implementation, speed, degree of data redundancy, ease with which they can satisfy information requirements, and ease with which they can be updated.

In use since the late 1960s, the hierarchical and network database models were first developed and used principally on mainframe computers. The concepts behind the relational database model were pioneered in the early 1970s. The relational database model, which takes advantage of large-capacity direct access storage devices, was developed for mainframe and minicomputer systems but is now also used on microcomputers. The relational database model is rapidly replacing the hierarchical and network models.

Now we will consider the three database models.

The Hierarchical Database Model

In the hierarchical database model, data is organised into related groups much like a family tree, with lower-level records subordinate to higher-level records. A lower-level record is called a *child*, and a higher-level record is called a *parent*. The parent record at the top of the database is called the *root record*.

Like families in real life, a parent in a hierarchical database may have more than one child, but a child always has only one parent. This is called a *one-to-many*

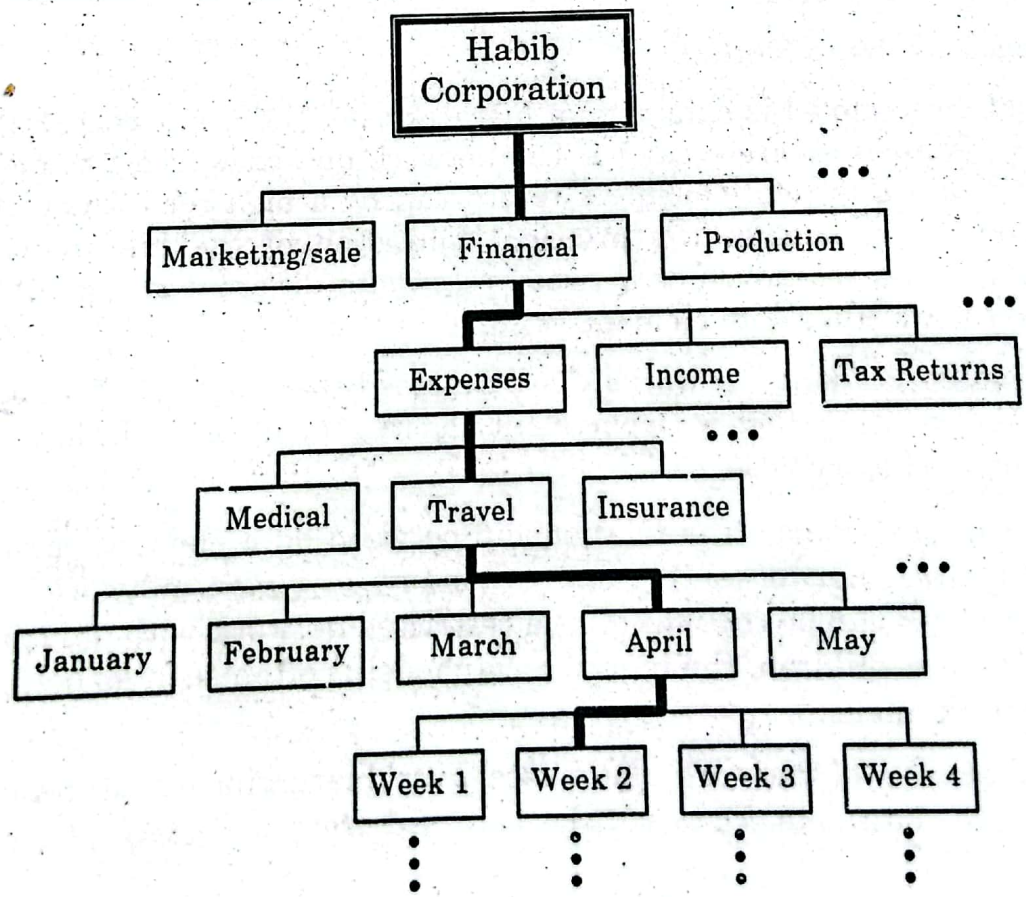


Fig. 15.4. Hierarchical database model for Habib Corporation. In this database model, which looks somewhat like a family tree, a parent record can have many child records, but each child record can have only one parent record. The root record is the topmost record.

CamScanner

relationship. To find a particular record, you have to start at the top with a parent and trace down the chart to the child. There is no connection between separate branches in this type of model. See, for example, the route in the Habib Corporation illustration for locating a record of travel expenses for the second week in April (see Fig. 15.4). The hierarchical database model works best with data that fall naturally into some hierarchy; that is, groups of data items that can be arranged according to some ranking, as in the organisation chart of a company.

Data accessed via a hierarchical database must be sequential. For example, accessing a student's record may require the database to move through university, college, and departmental data before accessing a specific student. By today's standards, this is cumbersome and has two distinct disadvantages. First, it is difficult to locate records in different areas, or sub-groups, of the database. In the example above, a search for a second student's record would again begin at the university level. This is an inefficient and time-consuming method for searching records in different groups. The second major disadvantage occurs when adding new fields. Hierarchical databases require a complete restructuring each time a new field is added—again, an inefficient and time-consuming process.

While accessing data from multiple groups and redefining an entire database can be time-consuming, once data is in place, searching for one specific record is very fast with a hierarchical database. The top-down organisation helps the database eliminate the need for searching through all records. Instead, the database searches only specified groups of data.

The Network Database Model

A network database model is similar to a hierarchical DBMS, but each child record can have more than one parent record. The network database terms "member" and "owner" to indicate this relationship. A record can be a member of more than one owner. This model overcomes the principal limitation of the hierarchical model because it establishes relationships between records in different groups. Any record can be related to any other data element.

Fig. 15.5 shows the Student B's "owners" are courses Broadcasting 310 and Indo-Pak History 104. The "owner" History Department has two "members"—Indo-Pak History 104 and European History 204.

In a network database model, the relationship between data elements can be either many-to-one (*simple* networks) or many-to-many (*complex* networks). Since a hierarchical database can be thought of as a restricted network with only single links between parents and children, the hierarchical model is often seen as being a special case of the network model.

The major advantage of network databases over hierarchical databases is their access to multiple data sets. Data can be accessed from a variety of sources with speed and ease.

The disadvantage is that the network databases, as do hierarchical databases, requires every relationship and link to be predefined. Adding any new record requires a complete redefinition of the database.

CamScanner

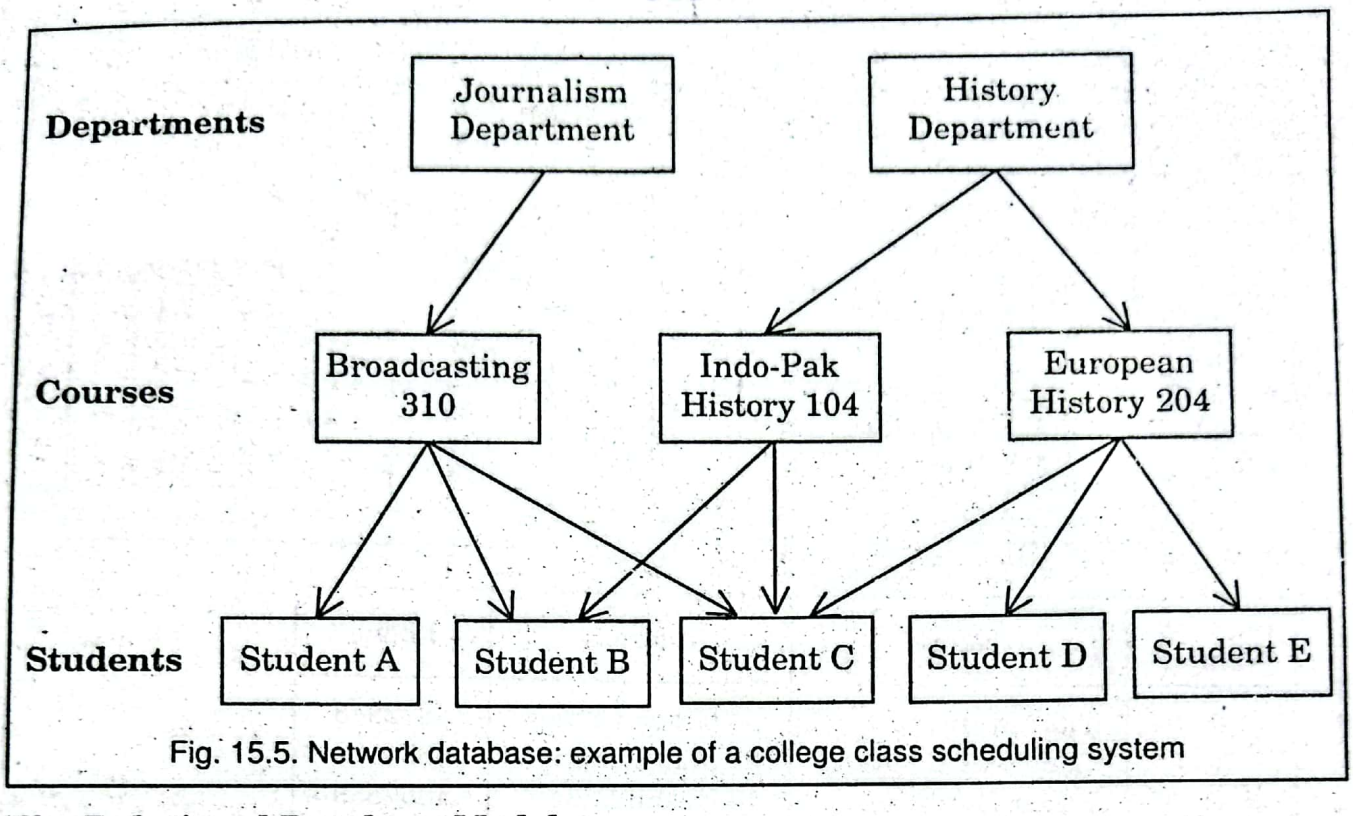


Fig. 15.5. Network database: example of a college class scheduling system

The Relational Database Model

The relational database model was developed by Dr. Edgar F. Codd, a mathematician working at IBM, in the early 1970s, as a result of recognised shortcomings of hierarchical and network DBMSs. Moreover, Codd supplied a collection of general purpose operations upon relations, called the *relational algebra,* which allow the designer to work on one or two relations to produce another relation.

Relational database model is the most commonly used type of database model for all levels of computers, from micros to mainframes. Even though the *network* and *hierarchical* database models are still *extensively* used, at some time in the future it is most likely that all database models will be relational—at least until somebody comes up with a better idea.

The relational database relates, or connects, data in different files through the use of a key field, or common data element. In this arrangement relational databases allow you to manipulate or control more than one file at one time. What makes this multiple file control possible is the organisational technique employed by the database program. Relational databases organise data into tables called *relations* in mathematical terms. Relations are subdivided into rows and columns. The rows represent data records, and the columns represent fields within those records.

In schools, relational databases are very popular. Instead of keeping all student data in one giant database, several databases are used. For example, there may be a database for academic grades, another for attendance, another for extra-curricular activities, and so forth.

In Fig. 15.6, we have organised the data about parts, shipments, and suppliers into three tables according to the relational model. The significance of this model is not

CamScanner

PARTS TABLE

Part Number	Name	Cost (Rs.)
123	½-inch screw	.5
234	Circuit board	125
345	Coiled cord	45
456	Keyboard casing	200
567	Key set	95

SHIPMENTS TABLE

Part Number	Supplier Number	Quantity
123	1	2000
567	2	4000
456	3	2300
345	4	5400
234	5	2100
124	6	1000
789	7	3850
678	8	3400

SUPPLIERS TABLE

Supplier Number	Name	Address	Phone
1	Pak Hardware	34 Ground Street	5872843
2	Computer Company	21 Mall Road	3854732
3	Badar Hardware	14 Kashif Lane	4857839
4	Dataline Wiring	45 Silver Street	3945729
5	Electro Electronics	76 Green Street	3452722
6	General Hardware	95 Millat Road	5892147
7	Sadat Bolts	85 Park Street	2549575
8	Hard Circuits	56 Main Street	8224868

Fig. 15.6. The Relational Model

that data are arranged in tables. The really important thing is that relationships between records are implied by the data values stored in common fields. For example, each part listed in the PARTS table is associated with its supplier number in the SHIPMENTS table. That is, the relationship between part and supplier number is implied by the common field, Part Number. Part number 456 is also a field in a record in the SHIPMENTS table. This shows that its supplier is number 3, and from the SUPPLIERS table we see that this number refers to Badar Hardware.

Relational databases are popular because the data within each of the separate files can be combined to produce a wide range of queries and reports. The table structure makes this all possible. When the table structure of one file has one or more of the same field attributes as another file, called a *common field*, the two files can link together.

One of the major advantages of relational databases is that there are no parents or children. This means that relationships among records in different tables are not predetermined by the structure of the database. Instead, they are specified by the user when data are retrieved. Consequently, relational database systems provide a

CamScanner

high degree of flexibility and data independence. There is no need to know in advance what questions users might ask about information in the database. Data files can be designed and set up and then perhaps used for purposes that had not been considered. Relational databases also offer the advantage of adding new fields to any file without having to convert or rearrange data links. Other types of database organisations require precise definitions for linking files.

Relational databases also offer the advantage of allowing changes made to one record in one file to change a record automatically in another file. For example, suppose one file contains a student's name, roll number, and percentage marks. Another file contains that student's name, roll number, and marks of individual subjects (physics, chemistry, mathematics, Islamic studies, etc.). Changing marks of mathematics in the second database can cause the percentage marks in the first database to change.

A disadvantage is that some searches can be time consuming. Nevertheless, the relational model has become popular for microcomputer DBMSs, such as Paradox or Access.

CamScanner

Exercise

15.1. Why is a database management system important to many organisations?
15.2. What is the difference between a primary key and a secondary key?
15.3. List some of the advantages and disadvantages of database system.
15.4. What is a database management system?
15.5. What characteristics should a good DBMS exhibits?
15.6. Evaluate the three major data models. In what situations would one be better than the others, and why?
15.7. What are the advantages of organising a database according to a relational model as opposed to a network model?
15.8. For each of the following types of database management systems, describe how data relate to one another:
(a) Relational DBMS
(b) Hierarchical DBMS
(c) Network DBMS
15.9. Describe, using both your own words and an example, how a relational database works.
15.10. What kinds of things should a DBMS report generator be able to do?
15.11. Name at least five advantages and four disadvantages of database management system.
15.12. What is a query language and why is it a valuable aspect of DBMS software?
15.13. What does the data dictionary in a DBMS provide?
15.14. Refer to the two tables in the sample relational database, and answer the following questions:

Employee Table

Emp. Name	Location	Department	Salary
M. Amir	Hyderabad	Accounting	5800
Bashir Ahmad	Karachi	Sales	7000
Maqbool Siddiqui	Kotri	Sales	4500
Sohail Aslam	Khairpur	Accounting	6200
Pervaiz Malik	Rohri	Accounting	7300
Baqir Ali	Khairpur	Sales	5600

Office Table

Location	Manager
Khairpur	Faheemuddin
Kotri	M. Saad
Rohri	M. Ali
Hyderabad	Qasim Abbas

(a) What is the average salary of an employee in the Sales department?
(b) Which employees does M. Ali supervise?
(c) What is the average salary under Faheemuddin's supervision?
(d) In which of the preceding questions did you have to relate data in both tables to get an answer? Through what fields did you relate tables?

CamScanner

15.15. Referring to the table given below, answer the following with respect to the relational database.

(a) What are the name, address, and telephone number of the owner of restaurant R3?

(b) Which restaurants have annual sales of 450000 or less? Who owns the restaurants in question?

(c) What initial fee was charged for restaurant R5? What are the royalty and advertising percentages in effect for this restaurant?

(d) How much does restaurant R3 owe to the corporation in royalty and advertising fees for the coming year? (answer in rupees, as opposed to percentage.)

(e) Which contract type has the highest initial fee? What is that amount?

(f) Which restaurant has the highest annual sales? How much?

(g) Which franchisees do not live in Karachi?

(h) What is the total paid to the corporation in initial fees for the five existing restaurants?

(i) Who owns the restaurant with the highest annual sales?

(a) Restaurant File

Restaurant Number	Street Address	City	Province	Annual Sales	Franchisee Number	Contract Type
R1	34 Ground Street	Karachi	Sindh	600000	F1	C1
R2	21 Mall Road	Hyderabad	Sindh	450000	F2	C1
R3	14 Kashif Lane	Kotri	Sindh	250000	F1	C2
R4	45 Silver Street	Lahore	Punjab	975000	F3	C1
R5	76 Green Street	Khairpur	Sindh	50000	F4	C3

(b) Franchisee File

Franchisee Number	Franchisee Name	Telephone	Street Address	City	Province
F1	Kabir Ahmad	3274623	95 Millat Road	Karachi	Sindh
F2	Arif Hussain	9876728	45 Peak Street	Karachi	Sindh
F3	Qazi Samad	3848189	85 Park Street	Lahore	Punjab
F4	Munawar Ali	3846219	56 Main Street	Peshawar	NWFP

(c) Contract File

Contract Type	Term (years)	Initial Fee	Royalty %	Advertising %
C1	99	250000	2%	2%
C2	5	50000	4%	3%
C3	10	75000	3%	3%

CamScanner

15.16. Consider the relation scheme containing book data for a bookstore:

```
BOOK (TITLE, AUTHOR, ISBN, PUBLISHER, PUB_DATE, CITY,
QTY_ON_HAND)
```

(a) Write out the table for an instance of this relation.

(b) Identify a primary key, and a secondary key, writing out any assumptions you need to make to justify your choice.

Objectives and MCQs

Choose the answer that best completes the statement.

15.1. Which of the following hardware components is the most important to the operation of a database management system?

(a) high-resolution video display
(b) printer
(c) high speed, large capacity disk
(d) plotter
(e) mouse

15.2. Which of the following is not function of a DBMS?

(a) providing security to database
(b) providing facilities for back up and recovery of data
(c) providing facilities to generate code for a given set of specifications
(d) none of the above

15.3. Database management systems are intended to:

(a) eliminate data redundancy
(b) establish relationships among records in different files
(c) manage file access
(d) maintain data integrity
(e) all of the above

15.4. Which of the following can help speed up data retrieval?

(a) program file
(b) key field
(c) offline storage
(d) data file
(e) all of the above

CamScanner

15.5. Which of the following is not characteristic of a relational database model?
(a) tables
(b) treelike structure
(c) complex logical relationships
(d) records
(e) fields

15.6. Which of the following is a type of DBMS software?
(a) database manipulation language
(b) query language
(c) utilities
(d) report writer
(e) all of the above

15.7. A data dictionary doesn't provide information about:
(a) where data is located
(b) the size of the disk storage device
(c) who owns or is responsible for the data
(d) how the data is used
(e) security and privacy limitations

15.8. To retrieve information from a database, users can use a(n) _____, an easy-to-use computer language.
(a) data dictionary
(b) report generator
(c) data mining structure
(d) query language
(e) none of the above

15.9. To extract data from a database and display it or print it to in a preformatted form, users can use a(n) _____.
(a) DBMS utility
(b) QBE
(c) report generator
(d) shared database
(e) none of the above

CamScanner

Objectives

15.1. If you have a collection of related records that forms an integrated base of data that can be accessed by a wide variety of applications programs and user requests, then you have a database.

15.2. A 0 or 1 is called a bit; 8 of these make up a byte, which represents a character.

15.3. Each database record consists of distinct types of data called fields.

15.4. A field is a unit of data consisting of one or more characters.

15.5. A field is sometimes called an attribute or a data element.

15.6. In ascending order, the levels of data stored in a database are character, field, record, file, and database.

15.7. A key field is a particular field chosen to uniquely identify a record so that it can be easily retrieved and processed.

15.8. Sequential access storage is mainly used on magnetic tape.

15.9. The aspect of DBMS software that simplifies the process of generating reports is called a report writer.

15.10. A special file in the DBMS called the data dictionary maintains standard definitions of all data items within the scope of the database.

15.11. Privacy is the right of individuals to not reveal information about themselves.

15.12. Ensuring backup and recovery of a database is not one of the functions of a database administrator. (true/false)

15.13. There can be more than one root in a hierarchical model. (true/false)

15.14. Relational database management systems are the type of DBMSs typically found on microcomputers.

15.15. The most sophisticated database model that allows complex logical relationships among records in many different files is the relational database model.

15.16. dBASE III, dBASE IV, and Paradox are popular microcomputer DBMS packages. (true/false)

15.17. SQL is an acronym for Structured Query Language.

15.18. A query language allows the user to phrase requests for information from a database in very flexible fashion.

15.19. If an element of data in a database needs to be changed, it must be changed in *all* the files in order for data integrity to be maintained.

15.20. Data redundancy means that an element of data (field) is repeated in many different files in a database. (true/false)

15.21. Match the following:

(a) Attribute	(i) A collection of related records
(b) File	(ii) Root
(c) Hierarchical database	(iii) One column of a relation
(d) Data	(iv) Raw facts

CamScanner

BASICS OF MS ACCESS 2000

Databases are large collections of related data stored together to provide useful information about related topics. For example, most people keep a list of the names, addresses, and phone numbers of the people they contact frequently.

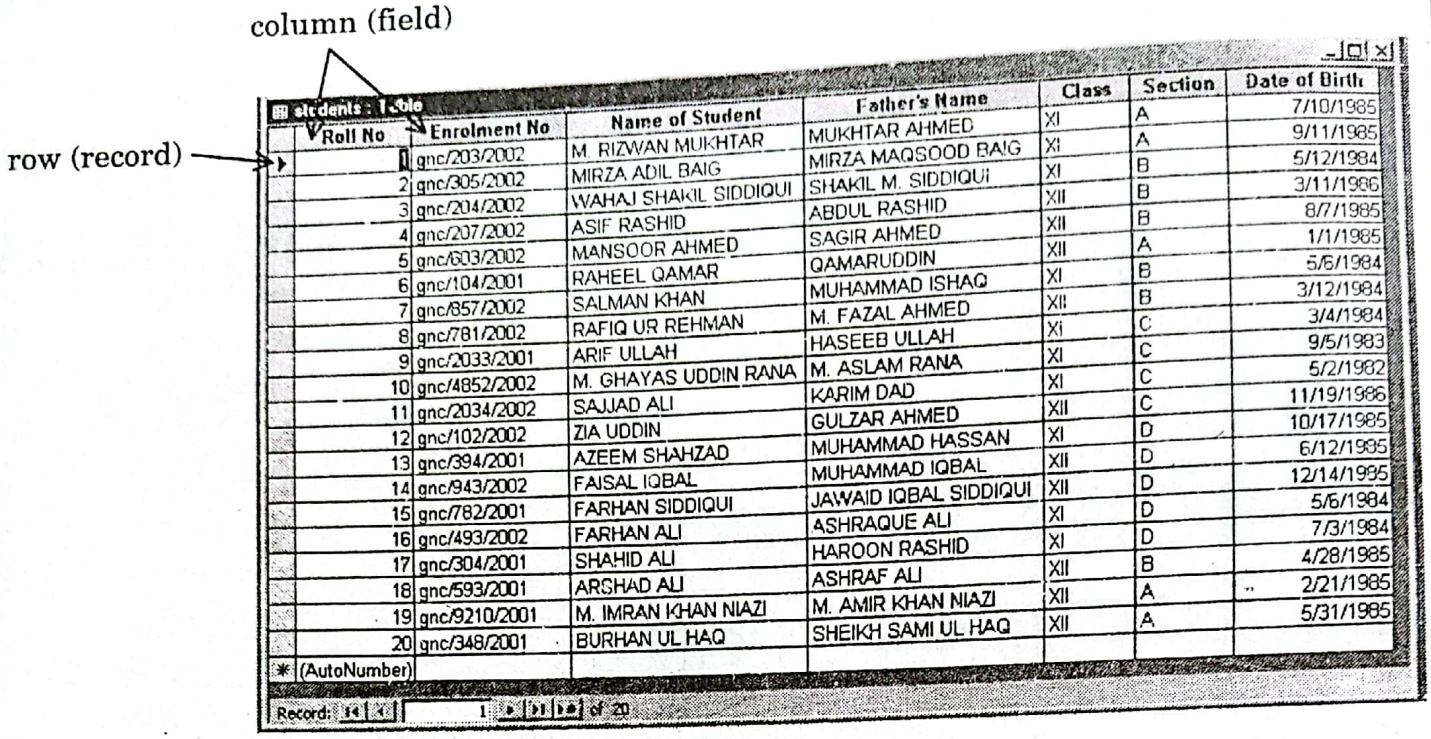
Microsoft Access is a powerful Windows-based relational database management system (RDBMS) for organising and storing large volumes of data in a relatively small space so that you can find data more efficiently. Each database consists of one or more tables; a table is a collection of data about a particular subject organised in a row-and-column format. Fig. 16.1 shows the *students* table (a sample application provided on CD-ROM). This database serves the information needs of a college that enrolled twenty students.

In a table, a *field* is a category of information, such as roll no. or enrolment no. in Fig. 16.1. A *record* is a collection of related fields such as a student with roll no. 1 and enrolment no. gnc/203/2002 named M. Rizwan Mukhtar father's named Mukhtar Ahmed, class XI, section A, whose date of birth is October 7, 1985.

Using Access, you can quickly view only those records meeting your search criteria and limit display to selected fields.

In addition to tables, Microsoft Access databases contain other components, each of which is called an object. A *database object* is a component of the database that gives it functionality. Each database object belongs to a category of objects known as a *class*. For example, a table for storing phone list information is a specific instance of a table object. Microsoft Access has seven classes of database objects. A table, a query, a form, a report, a page, a macro, and a module are all classes of database objects. The purpose of each object is listed in table 16.1.

CamScanner



Roll No	Enrolment No	Name of Student	Father's Name	Class	Section	Date of Birth
1	gnc/203/2002	M. RIZWAN MUKHTAR	MUKHTAR AHMED	XI	A	7/10/1985
2	gnc/305/2002	MIRZA ADIL BAIG	MIRZA MAQSOOD BAIG	XI	A	9/11/1985
3	gnc/204/2002	WAHAJ SHAKIL SIDDIQUI	SHAKIL M. SIDDIQUI	XI	B	5/12/1984
4	gnc/207/2002	ASIF RASHID	ABDUL RASHID	XII	B	3/11/1986
5	gnc/603/2002	MANSOOR AHMED	SAGIR AHMED	XII	B	8/7/1985
6	gnc/104/2001	RAHEEL QAMAR	QAMARUDDIN	XII	A	1/1/1985
7	gnc/857/2002	SALMAN KHAN	MUHAMMAD ISHAQ	XI	B	5/6/1984
8	gnc/781/2002	RAFIQ UR REHMAN	M. FAZAL AHMED	XII	B	3/12/1984
9	gnc/2033/2001	ARIF ULLAH	HASEEB ULLAH	XI	C	3/4/1984
10	gnc/4852/2002	M. GHAYAS UDDIN RANA	M. ASLAM RANA	XI	C	9/5/1983
11	gnc/2034/2002	SAJJAD ALI	KARIM DAD	XI	C	5/2/1982
12	gnc/102/2002	ZIA UDDIN	GULZAR AHMED	XII	C	11/19/1986
13	gnc/394/2001	AZEEM SHAHZAD	MUHAMMAD HASSAN	XI	D	10/17/1985
14	gnc/943/2002	FAISAL IQBAL	MUHAMMAD IQBAL	XII	D	6/12/1995
15	gnc/782/2001	FARHAN SIDDIQUI	JAWAID IQBAL SIDDIQUI	XII	D	12/14/1995
16	gnc/493/2002	FARHAN ALI	ASHRAQUE ALI	XI	D	5/6/1984
17	gnc/304/2001	SHAHID ALI	HAROON RASHID	XI	D	7/3/1984
18	gnc/593/2001	ARSHAD ALI	ASHRAF ALI	XII	B	4/28/1985
19	gnc/9210/2001	M. IMRAN KHAN NIAZI	M. AMIR KHAN NIAZI	XII	A	2/21/1985
20	gnc/348/2001	BURHAN UL HAQ	SHEIKH SAMI UL HAQ	XII	A	5/31/1985
(AutoNumber)						

Record: 1 of 20

Fig. 16.1. A sample table named "students"

Table 16.1. Seven Classes of Database Objects

Object Class	Purpose
Table	An organized collection of rows and columns used to store field data.
Query	An object that is used to view, change, or organize data.
Form	A graphical object that displays data from a table or a query in an easy-to-use format.
Report	An object used to present data in a printed format.
Pages	Data Access Pages are HTML files designed in Access and formatted to display in a Web browser.
Macro	A set of one or more actions that are used to automate common tasks such as opening a form or printing a report.
Module	A collection of Visual Basic for Applications programming components that are stored together as a unit.

In Microsoft Access, the database objects you create are stored in a single database file with a **.mdb** file name extension.

If you are thinking that such a powerful program must be difficult to learn, you will find that numerous wizards make it possible for you to create and use a sophisticated database system with little knowledge. A *wizard* asks you questions and builds an object such as a database, table, query, form, or report based on your answers. For example, *Database Wizard* builds a complete application including tables, forms, and reports.

CamScanner

Launching Microsoft Access and Creating a New Database

As with each of the Office applications, you can launch Microsoft Access in a variety of ways from the Windows desktop. In the steps that follow, you will use the Start button to launch Access and create a new database.

TASK 1: To Launch Microsoft Access and Create a New Database

1. Click the Start button.
2. Click Programs from the Start menu and click Microsoft Access, as shown in Fig. 16.2.

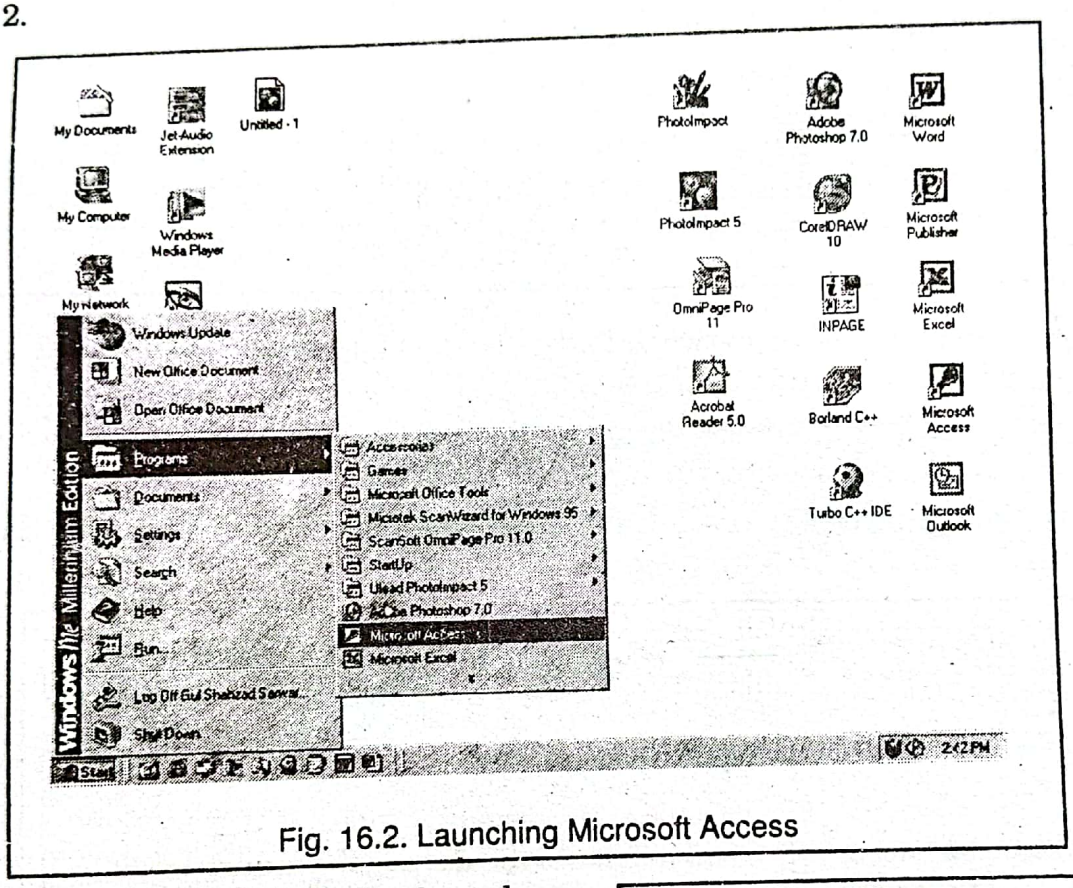


Fig. 16.2. Launching Microsoft Access

3. The Microsoft Access dialog box shown in Fig. 16.3 appears. You have three options: opening an existing file (database) using the list of recently used databases, creating a new database using the database wizard, or creating a blank database.
4. Select the Blank Access database option button and click OK. The File New Database dialog box shown in Fig. 16.4 appears. Notice that the dialog box specifies a default location and database name. The folders appearing to the left may vary, depending on your settings.

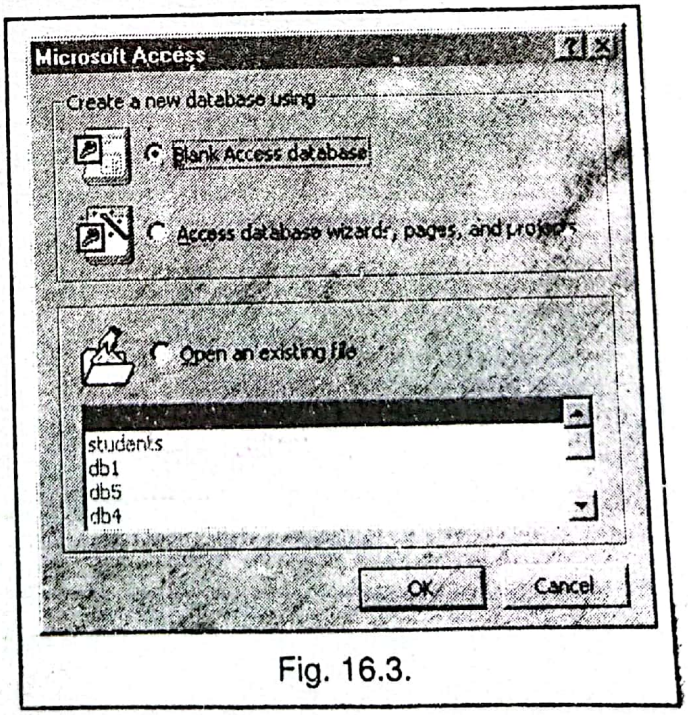


Fig. 16.3.

CamScanner

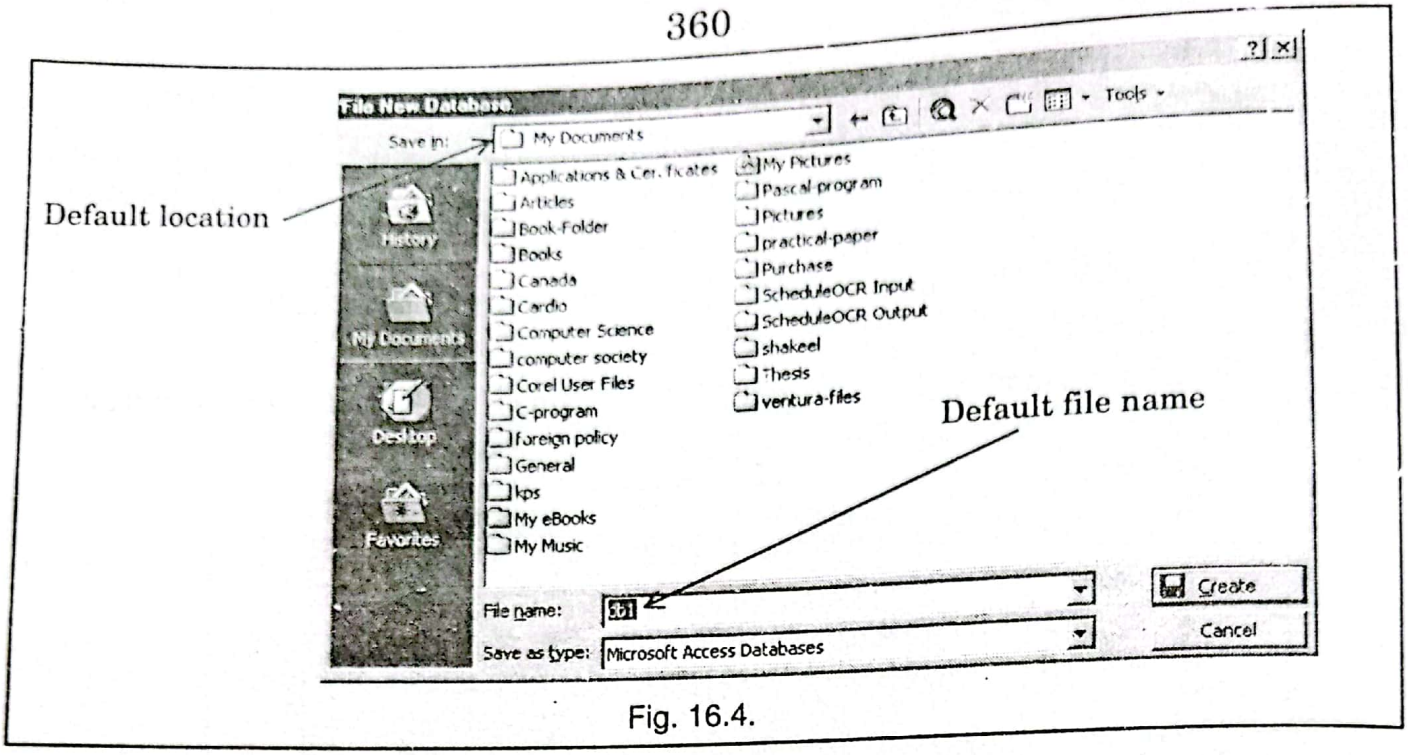


Fig. 16.4.

5. Select your floppy disk or network folder or keep the same folder as the storage location, and type `college` as the file name.

   TIP When you name your database file, Access will automatically add the `.mdb` extension to the file name.

6. Click the Create button. Access creates a new database and displays the database window shown in Fig. 16.5.

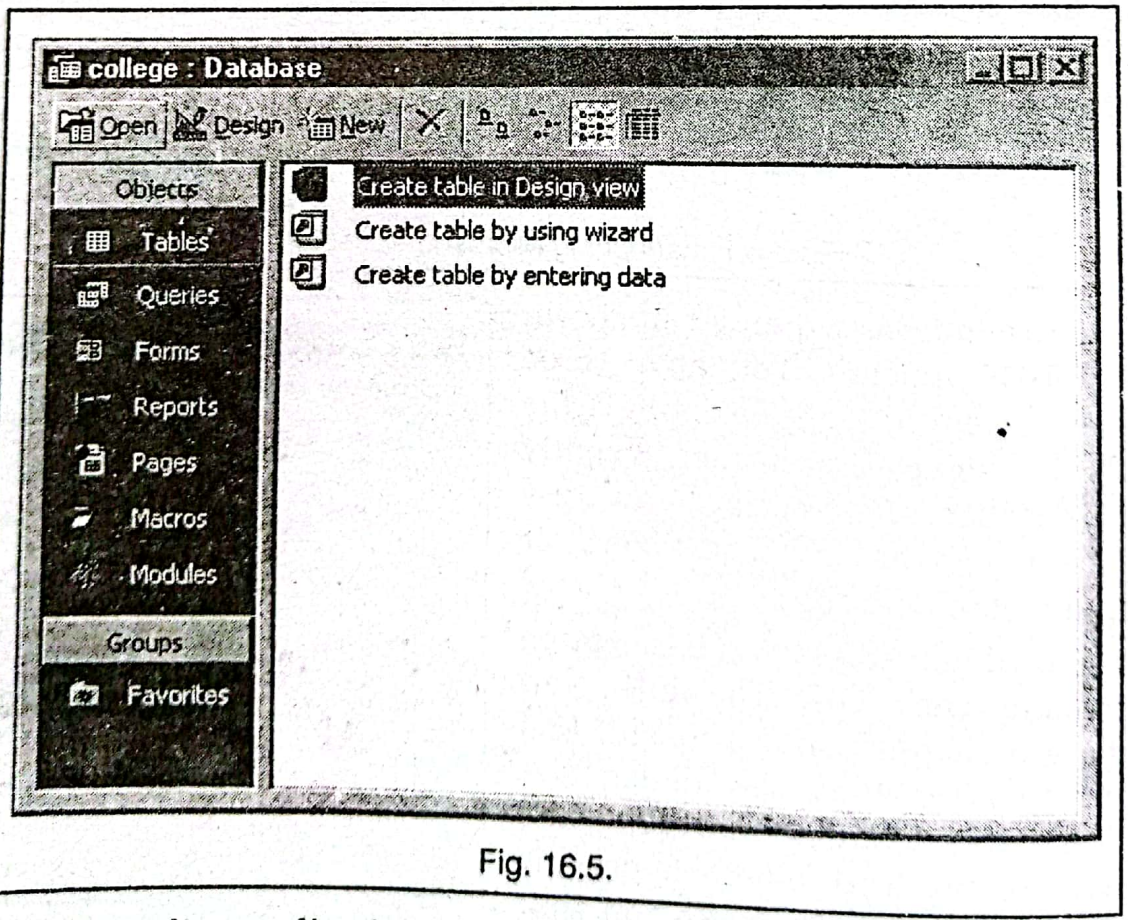


Fig. 16.5.

TIP Unlike other applications in the Office suite, only one database file can be open at a time.

CamScanner

Identifying Microsoft Access Screen Elements

The Microsoft Access screen features a number of elements found in other Office applications as well as several elements unique to Microsoft Access, as you can see in Fig. 16.6.

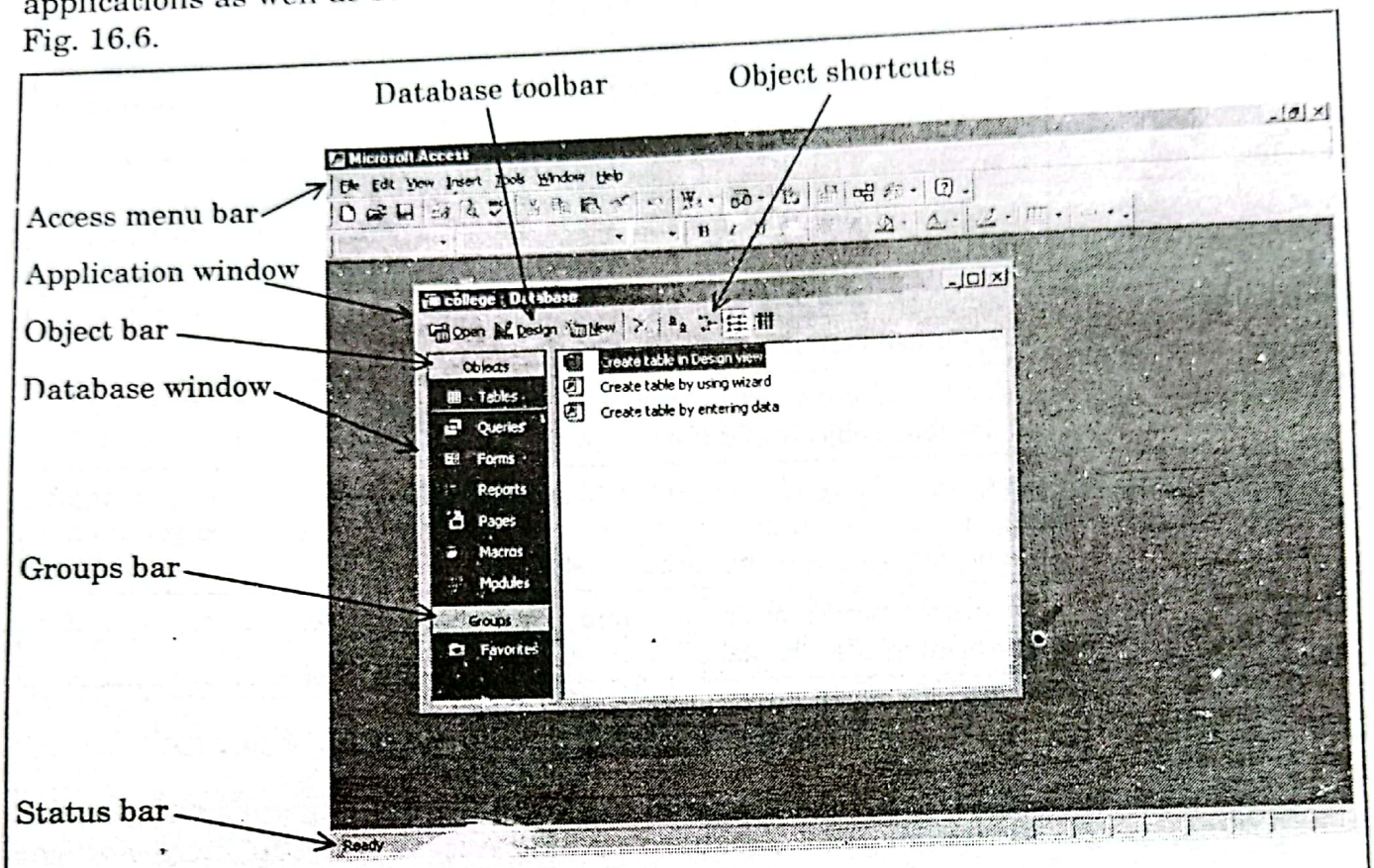
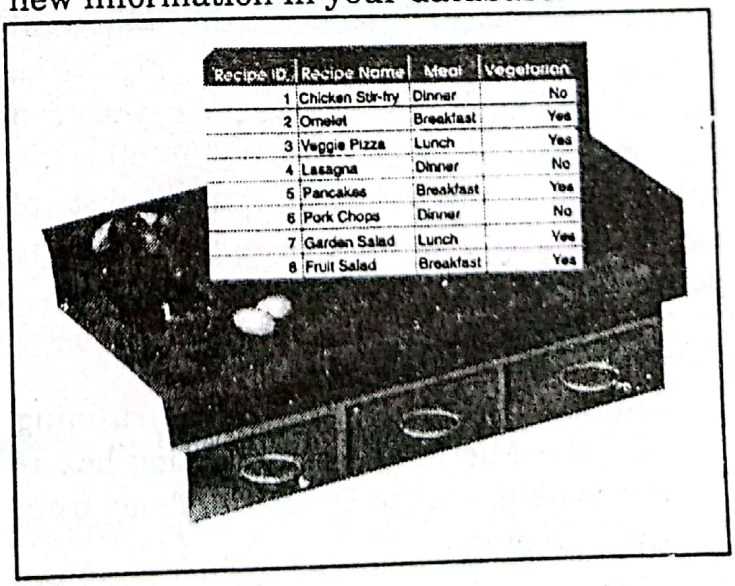


Fig. 16.6

TIP Notice that the object buttons in the database window appear in the same order as the seven classes of database objects listed in the steps for designing a database in table 16.1. Since tables are the primary repositories for the data in a database, the Tables button is listed first. A table stores a collection of information about a specific topic, such as a list of recipes. You can create a table to store new information in your database.

The most striking feature of the Access interface is the database window. As you create database objects, they will be displayed in the appropriate page for a specific object selection on the *Objects bar*. You will also notice that the application window contains only one toolbar. You will use this in the same way as other toolbars in the Windows environment. Table 16.2 identifies the purpose of each screen element.



CamScanner

Table 16.2. Purpose of Screen Elements

Screen Element	Purpose
Access menu bar	Provides access to the commands used to perform tasks.
Application title bar	Identifies the current application and contains control menu items for maximizing, minimizing, and restoring the application window and exiting the application.
Database toolbar	Provides shortcuts to the most common database tasks.
Database window	Displays object tabs for the seven kinds of database objects you can create.
Objects bar	Display the database objects you create in the Objects bar, which has an easy-to-use vertical orientation.
Groups bar	Click the Groups bar to view your groups, which can contain shortcuts to database objects of different types.
Object wizards and shortcuts	In the database window, quickly create a new database object by using a wizard, or open a new database object in Design view. Once you create an object, open it using its object shortcut.
Status bar	Displays program status and instructions and information for performing specific tasks.

Closing Your Database File and Exiting Microsoft Access

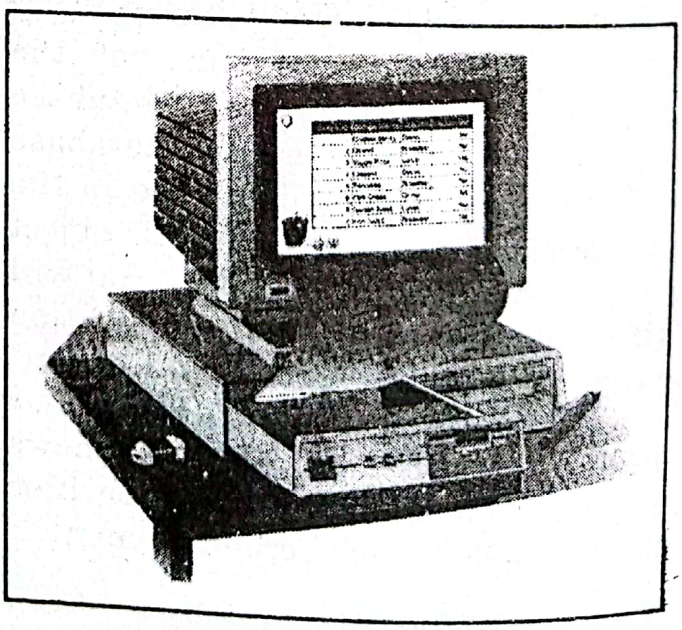
When you finish working on a database, you can save or discard your changes and then continue developing another database or exit Access. As you have not yet created any database objects, there is nothing to update in the file. Any time you close a database that you have modified, it is updated automatically.

TASK 2: To Close Your Database File and Exit Microsoft Access

1. Click the Close button (⊠) of the database window to close the database file or choose Exit from the File menu.

Opening an Access Database

When you launch Access, the Microsoft Access . dialog box shown in Fig. 16.8 appears. From this dialog box you can either create a new database or open an existing one. Remember that a list of recently opened databases appears at the bottom of the Microsoft Access dialog box.



TASK 3: To Open a Database

1. Launch Access. If Access is running and the Microsoft Access dialog box is not visible, choose Open Database from the File menu.
2. With the "Open an existing file" option

CamScanner

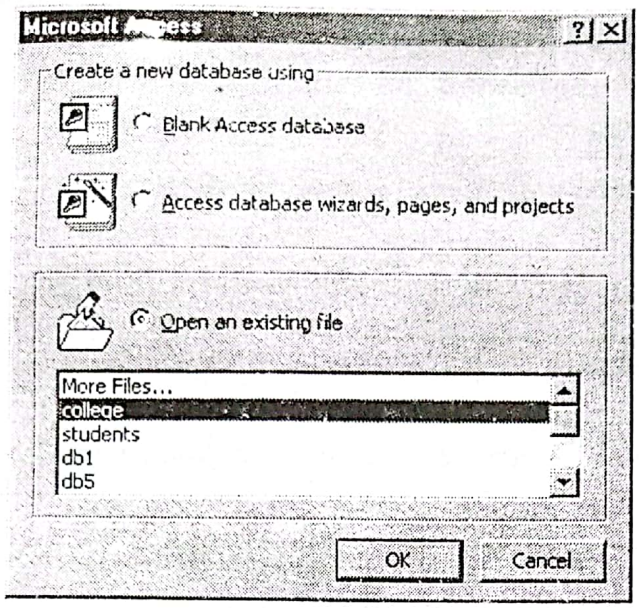


Fig. 16.8

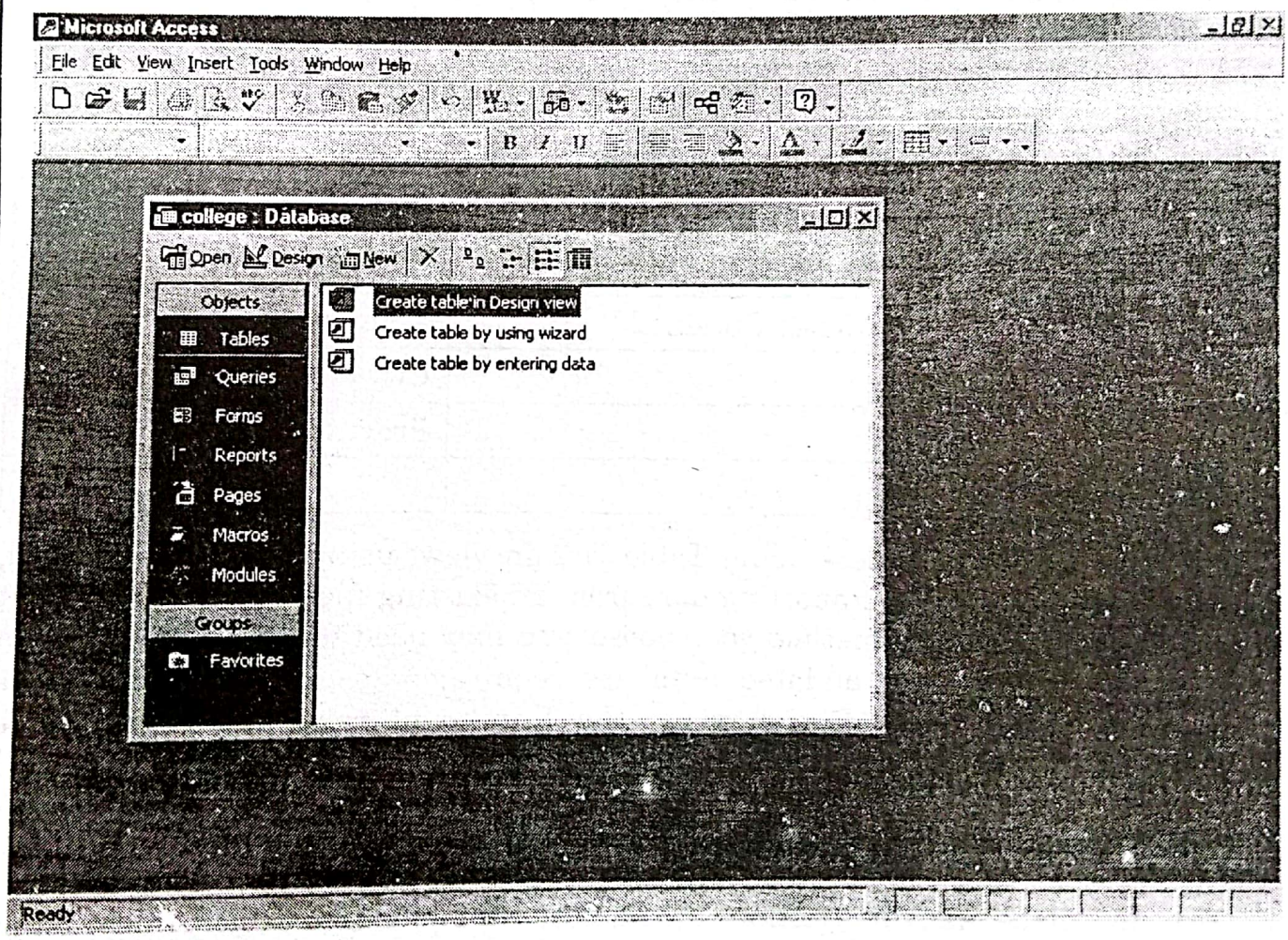


Fig. 16.9

selected, double-click `college` in the list at the bottom of the Microsoft Access dialog box.

TIP If the file you want doesn't appear on the dialog box list, select More Files. From the open dialog box that appears, select the folder that has the database you want to open and then double-click the database filename.

CamScanner

Creating Tables

Designing a table would involve the following steps:

1. Selecting fields or data items that would fit into each of the tables.
2. Specifying data type of each data items.
3. Providing a field size for the field, where possible.
4. Enforcing constraints on the required field.

The names of each field in the table, the type of data the field will contain, field size, constraints and other properties comprise the *structure* of the table. Each table in a database should also have a *primary key*, which is a field that is unique for each record in the database, and never contains an empty or null value. A table is a two-dimensional grid consisting of named fields. Tables are the primary repositories fo data in an Access database.

The first step in defining the table structure is determining the appropriate fields. Table 16.3 displays the structure for each of the tables you will create in this project. The field designated as the primary key is underlined.

Table 16.3. Structure of a Table

Students Table	**Enrolment Table**	**Address Table**
Roll No	Enrolment No	Roll No
Name of Student	Roll No	Phone Number
Father's Name	Marks Obtained	Address
Class		City
Section		Province
Date of Birth		

You can create tables in Access using Table Design view, using the Table Wizard, Entering data directly, or by importing data from an existing file such as a Microsoft Excel file. Regardless of the method you choose, you may need to modify the table in the future to accommodate updated database requirements. You will create the Students table using Table Design view, the Enrolment table using the Table Wizard, and the Address table by Entering data directly.

Naming Fields and Determining Data Types

Access does not place many restrictions for naming the fields in a table. A field name can be up to 64 characters in length and may include any combination of letters, numbers, spaces, and special characters except a full stop (.), an exclamation point (!), an accent grave (`), and brackets ([ ]).

To store your table data in the most efficient manner, Access supports different data types. A *data type* is a characteristic of how data is stored in a database. Depending upon what kind of data your fields will contain, various data types are appropriate. For example, a text data type is required to store name and address information. If you

CamScanner

need to perform calculations involving monetary units, you will need to use the currency data type. Table 16.4 summarizes the ten data types you may use for table data.

Table 16.4. Access Data Types

Data Type	Description
Text	Any combination of alphabetic and numeric characters such as names, addresses, and phone numbers. The text data type holds a maximum of 255 characters. This is the default data type.
Memo	Used for long text entries exceeding 255 characters. Holds up to 64 kilobytes of data in a random format.
Number	Numeric values such as inventory quantity or the number of items ordered. Numeric data can be used in calculations.
Date/Time	Date and time values for the years 100 through 9999.
Currency	Currency values and numeric data used in mathematical calculations involving data with one to four decimal places.
AutoNumber	A unique and sequential number is created as records are added to a table. AutoNumber data cannot be changed, edited, or deleted.
Yes/No	Single-character data in a Yes/No (Boolean) format. This data type displays a checkbox.
OLE object	Embedded or linked objects such as a Microsoft Excel spreadsheet, a Microsoft Word document, graphics, or sounds.
Hyperlink	Text or combinations of text and numbers stored as text and used as a hyperlink address. This data type is used to link to Web pages or other documents.
Lookup Wizard	A field that allows you to choose a value from another table or from a list of values.

When you create a table using a wizard, Access may determine the data types for you. When using Table Design view, you must specify the data types. When you create a table by importing data into Access, it will determine the appropriate data type.

Creating a Table using Table Design View

A common method for creating tables in Access is by using Table Design view. To create a table by this method, enter the names and properties for each field in the table. Once this process is complete, save the table structure.

TASK 4: To Create the Students Table Using Table Design View

1. Click the Tables button in the Objects bar, if it is not currently active.
2. Select the Create table in Design view shortcut in the database window and click the Open button, as shown in Fig. 16.10.

TIP You can also double-click this shortcut to open it.

CamScanner

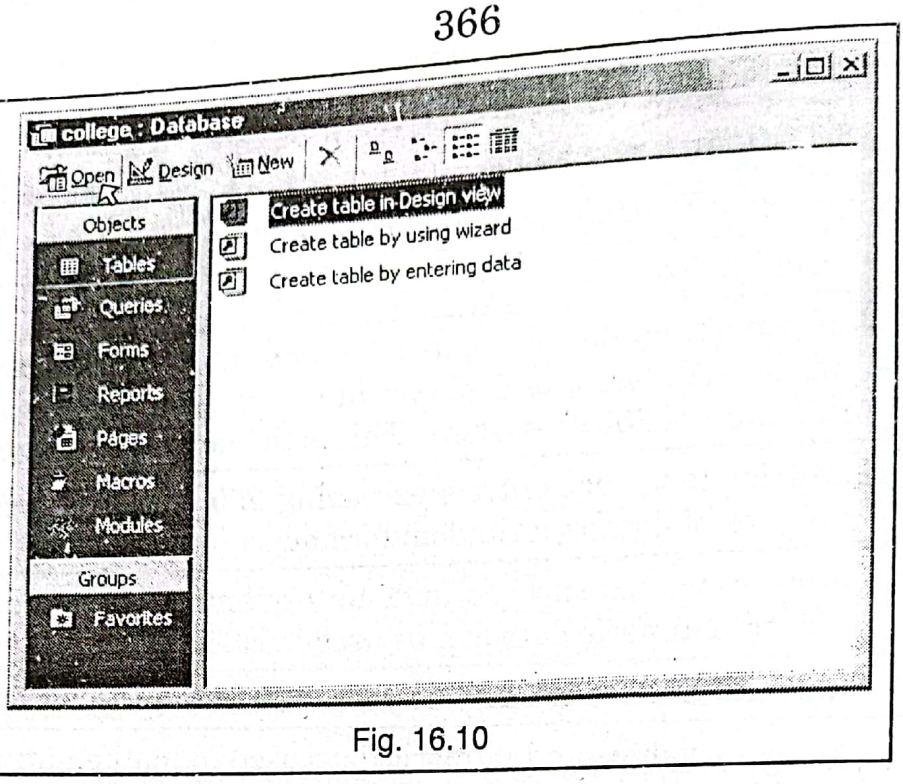


Fig. 16.10

3. Access opens a new table in Table Design view, which consists of the two panes displayed in Fig. 16.11.

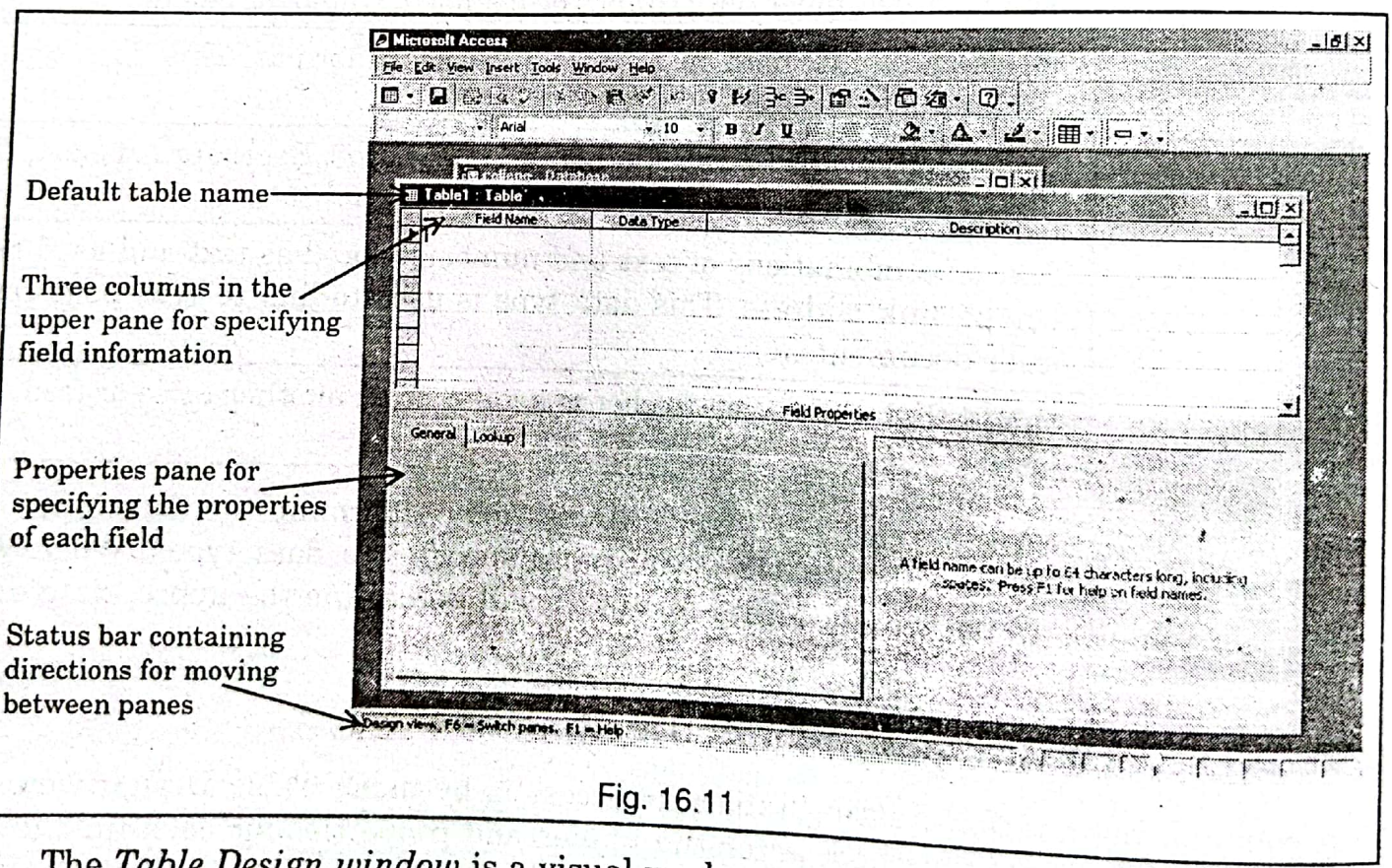


Fig. 16.11

The *Table Design window* is a visual workspace where you can enter information about each field in your table. Table Design View always displays this window.

4. The insertion point will be in the left column of the first row of the upper pane. Type *Roll No* in the left column of the first row and press Tab to move the insertion point into the Data Type column of the upper pane.

5. Click the drop-down list Data Type button in the Data Type column, and select AutoNumber as the data type for this field, as shown in Figure 16.12

CamScanner

TIP In Access, the AutoNumber data type is often used as a primary key, since it always contains a unique, non-null value.

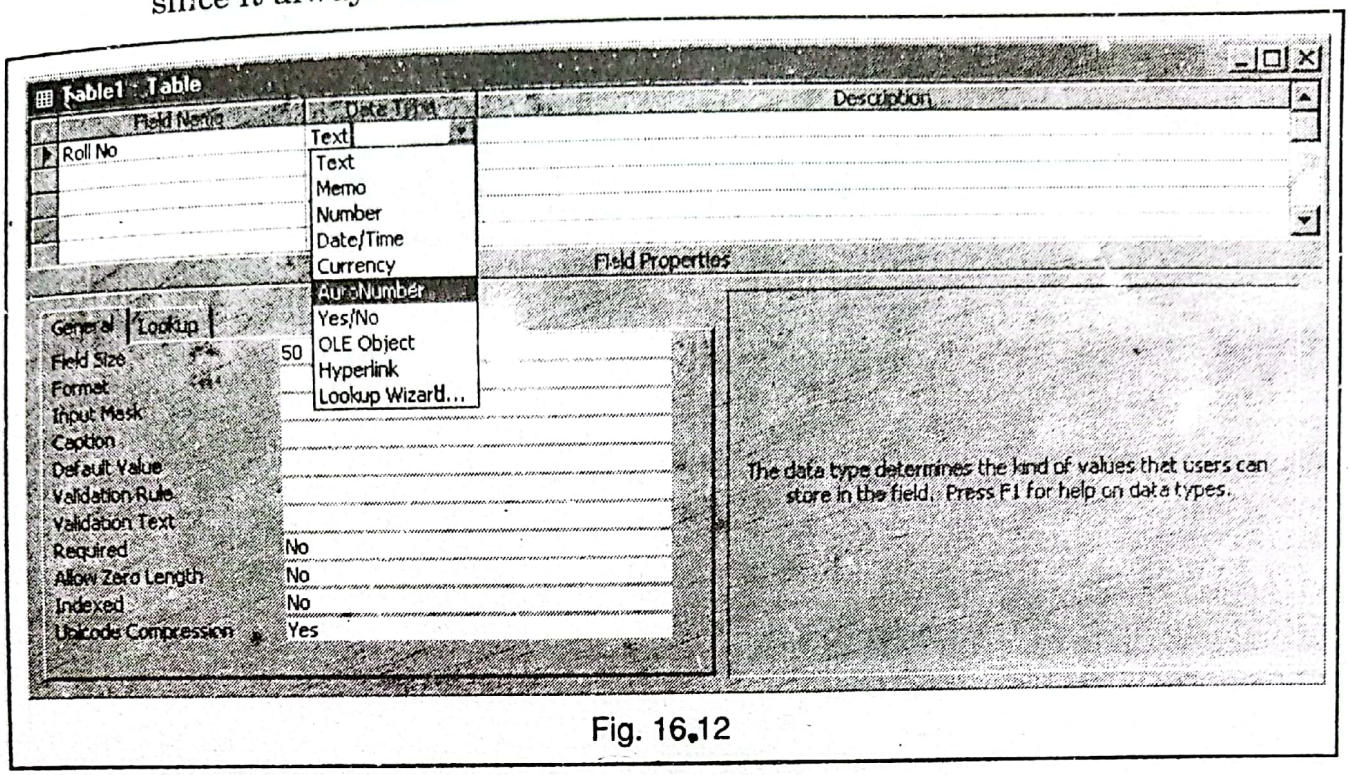


Fig. 16.12

6. Press Tab to move the insertion point into the Description column. Type *primary key*.

   TIP Although Access does not require anything in the Description column, you should consider documenting information about the table's structure for future reference.

7. Click the Primary Key button on the database toolbar to make this field the primary key.

   TIP An icon representing this field as the primary key appears to the left of the field name in the upper pane of the Table Design window.

8. Press Tab once to move the insertion point into the Field Name column for the second row. Type Enrolment No as the field name.
9. Press Tab to move the insertion point into the Data Type column. Press F6 once to activate the Properties pane for the current field. You will notice that the data type is set to text by default, and that the insertion point appears immediately after the default field size entry.
10. The default field size is selected. Type 35 as the field size for the field.
11. Press F6 to move back to the upper pane. Keep repeating the same procedure to define remaining table structure.
12. Click the Save button on the database toolbar to save the table structure.
13. Type students as the table name in the Save As dialog box and click OK, as shown in Fig. 16.13.

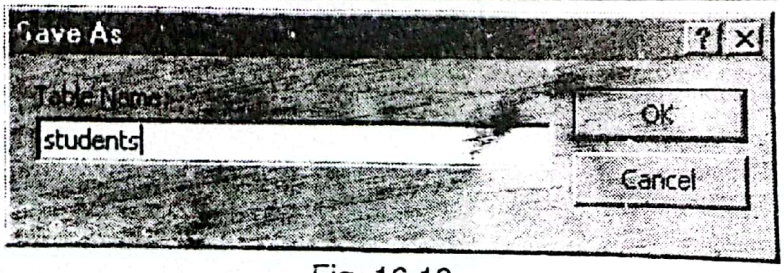


Fig. 16.13

CamScanner

14. Click the Close ☒ button in the Table Design window to close the table. The table name now appears in the database window, as shown in Fig. 16.14.

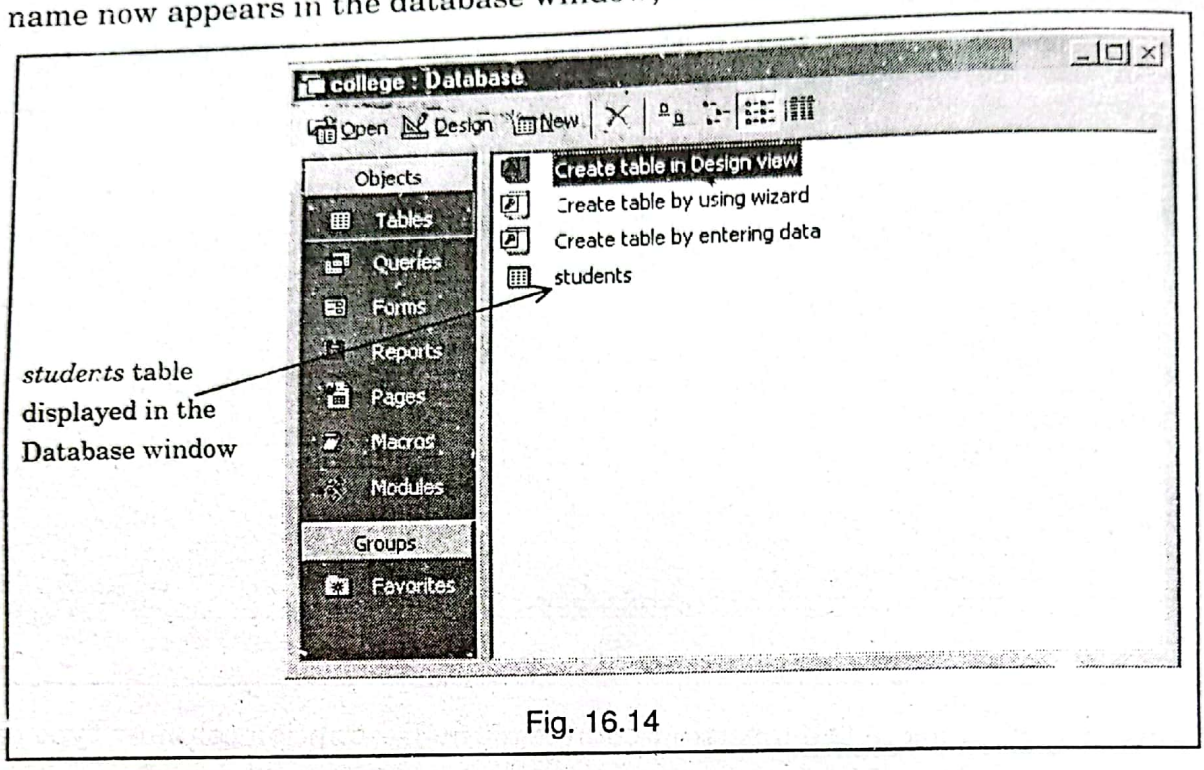


Fig. 16.14

Creating a Table Using the Table Wizard

You have successfully created a table using Table Design view. Access includes additional options for creating tables other than Table Design view. In this task you will create a table using the *Table Wizard*, which walks you through the table design process.

TASK 5: To Create the Address Table Using the Table Wizard

1. Select the "Create table by using wizard" shortcut as shown in Fig. 16.15, and click the Open Open button. The Table Wizard shown in Fig. 16.16 appears.

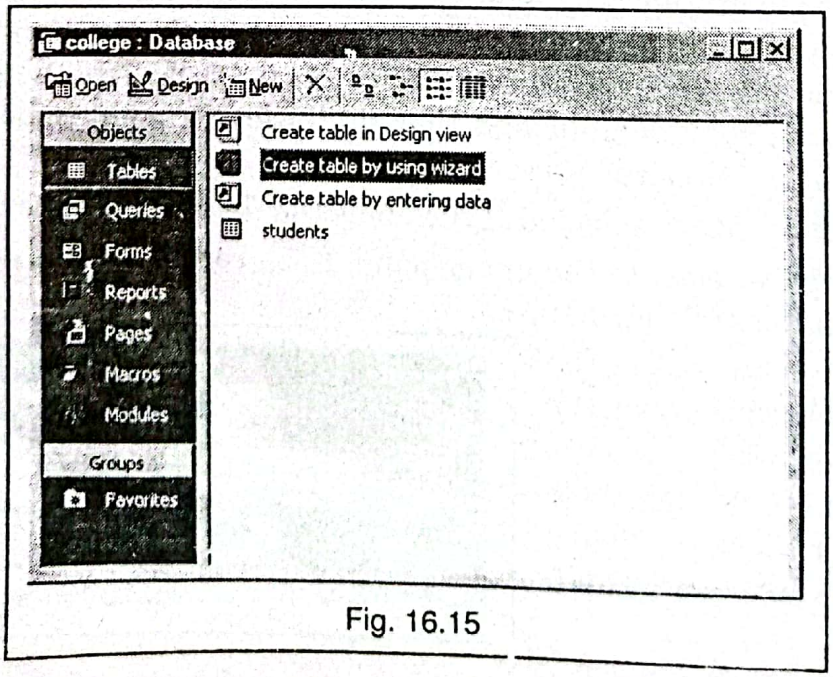


Fig. 16.15

CamScanner

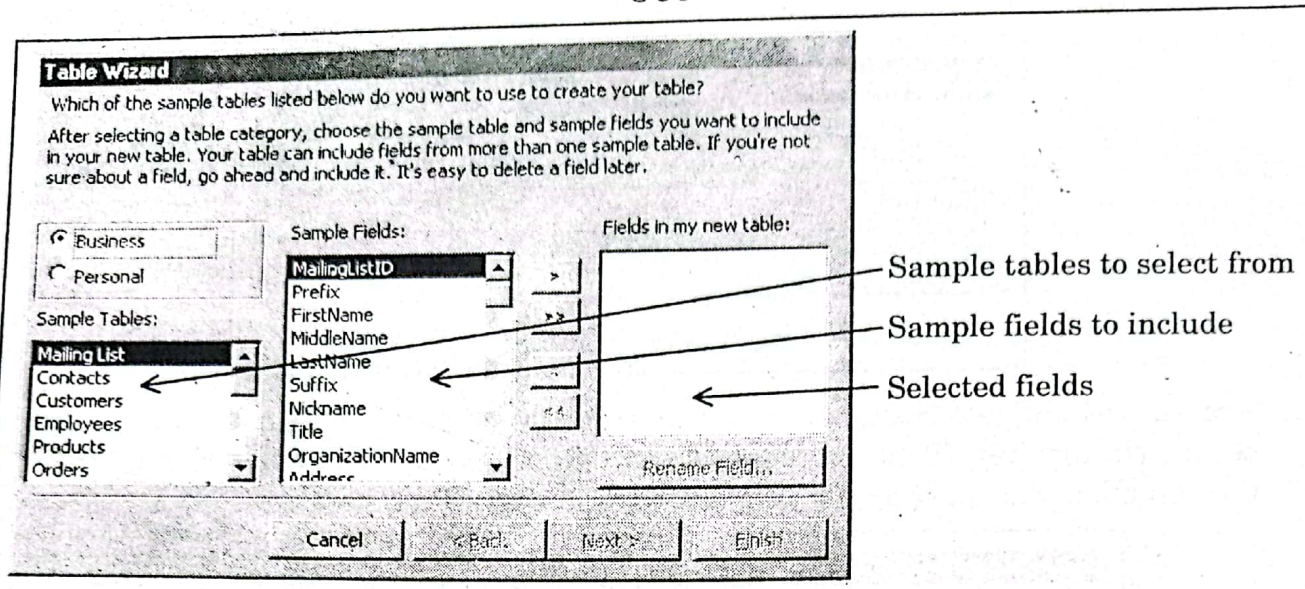


Fig. 16.16

2. There are two options: Business and Personal. Click one of these options to specify whether the table is for business or personal use (○ changes to ◉). Note that the available sample tables depend on this selection.
3. Click the sample table that best describes the table you want to create. Use the vertical scroll bar in the Sample Tables to see all tables. Select the Students table.
4. The Table Wizard now displays fields appropriate to your table. Click the Add button > to move the StudentID field from the Sample Fields to Field in my new table. The field is added, as shown in Fig. 16.17.

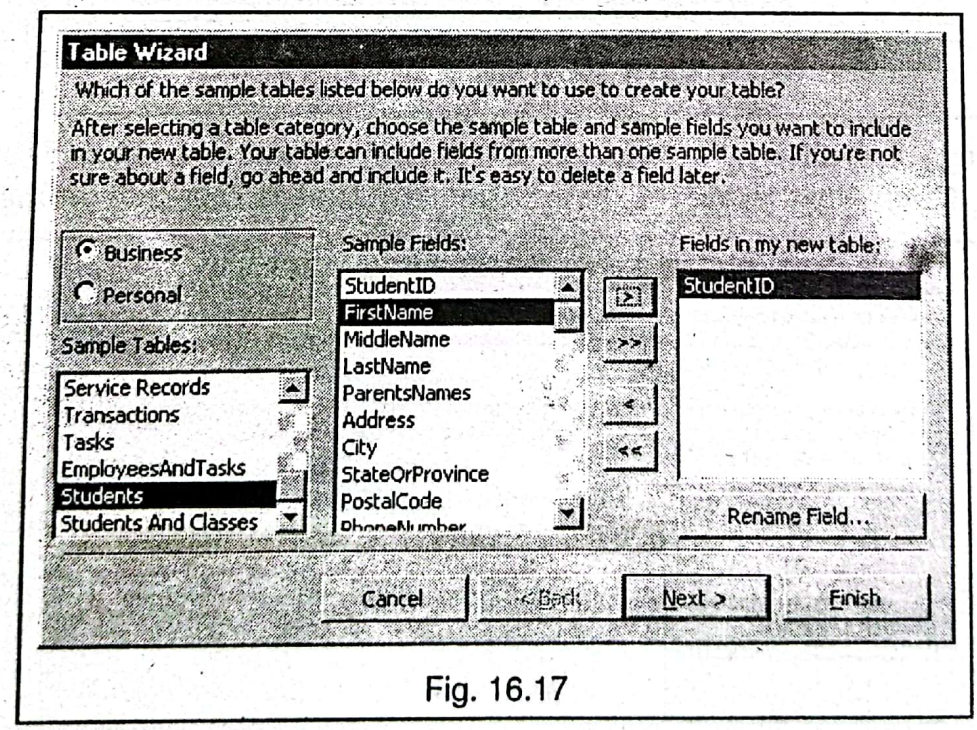


Fig. 16.17

5. Copy the Phone Number, Address, City and State or Province fields to the Fields in my new table. Click the StudentID field and then click the Rename field. Rename StudentID field to Roll No field as shown in Fig. 16.18. Similarly rename State or Province field to Province. Click Next to continue.

CamScanner

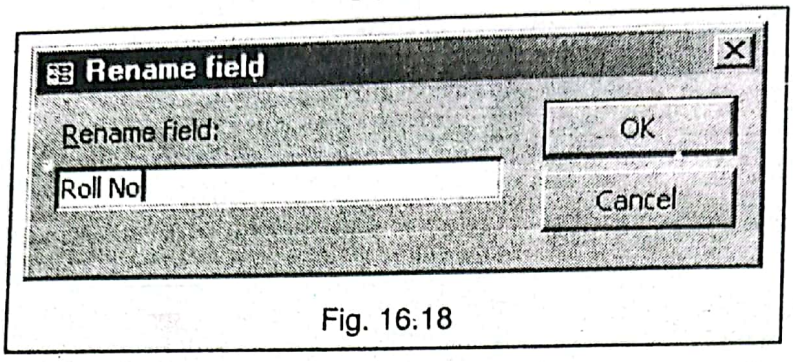


Fig. 16.18

6. Access will suggest Students as a name for this table. It will also request that it set a primary key. Type Address as the name. To have Access set a primary key for you click yes, as shown in Fig. 16.19, and click Next.

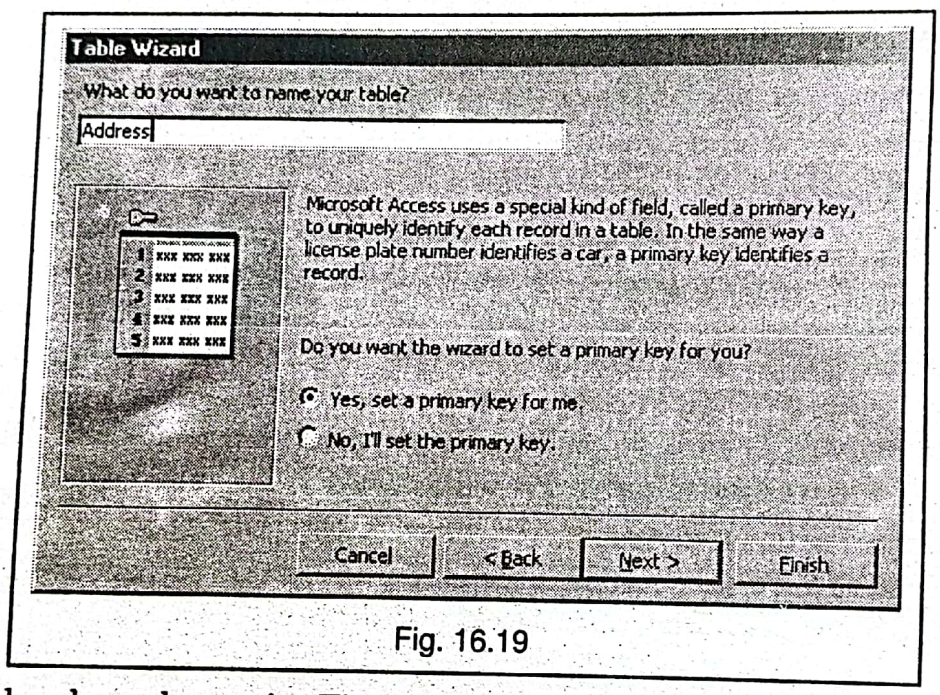


Fig. 16.19

7. The dialog box shown in Fig. 16.20 appears if another table exists in the database. Click Next to accept the relationship that Access has defined.

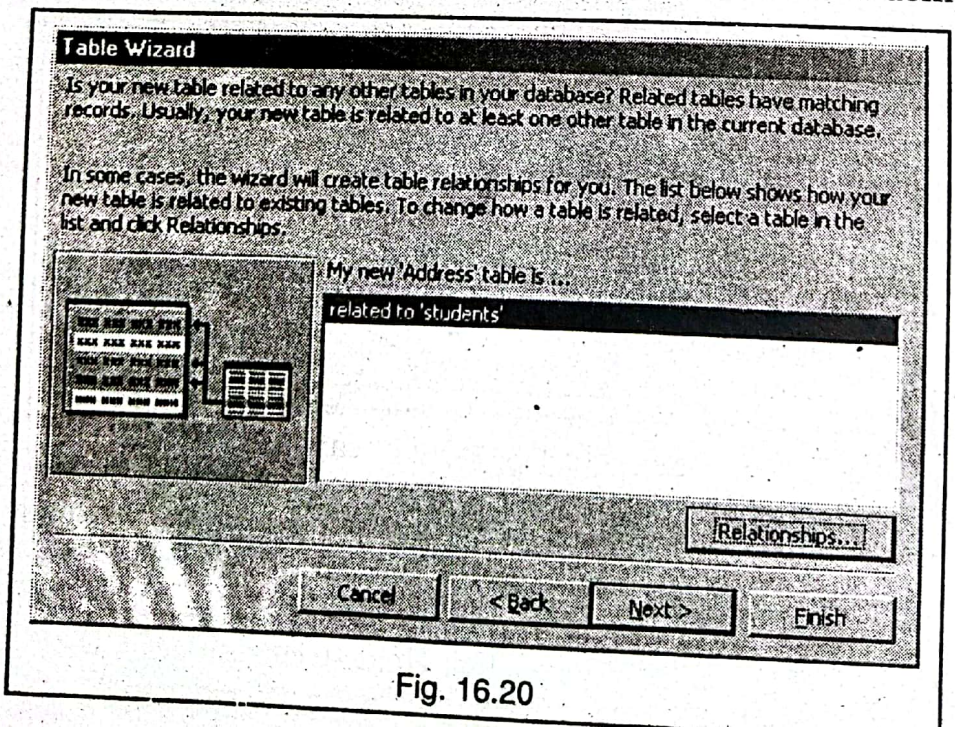


Fig. 16.20

CamScanner

TIP When creating a table, the wizard will show you how the new table relates to all the other tables in your database. Related tables have matching records.

8. The Table Wizard now allows you to select a view for the table once it is created. Click what you want to do. Select the option to enter data directly into the table and click Finish as shown in Fig. 16.21 to create the table.

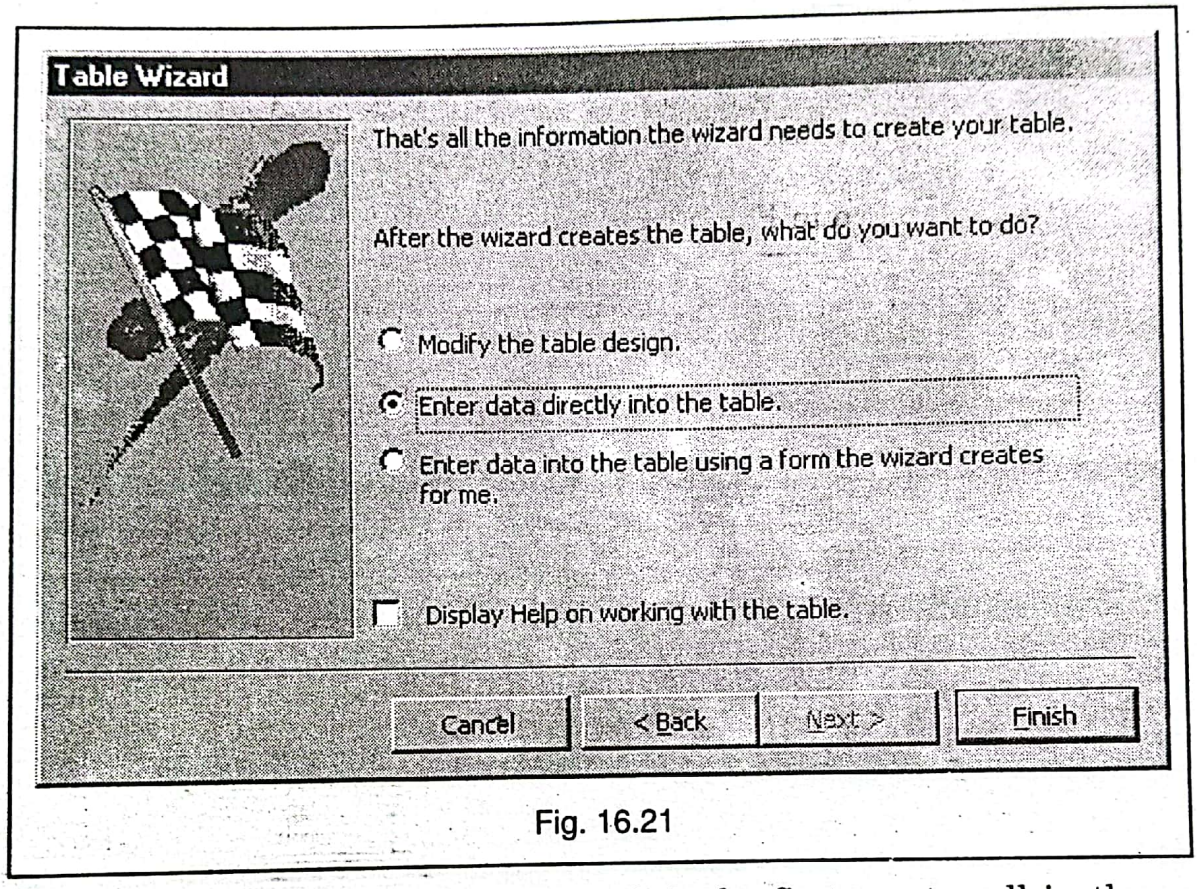


Fig. 16.21

9. To enter the data for the first record, click the first empty cell in the row. The type the data that corresponds to the field. Press ↵ on your keyboard to enter the data and move to the next cell. Then type the data that corresponds to the next field. Repeat this step until you finish entering all the data for the record.

TIP You can change the properties of a field at any time using Table Design window. Here the format control code of Phone Number has been modified.

10. Click the Save button to update the table.
11. Click the Close ☒ button to return to the database window.

Creating a Table by Entering Data Directly

An easy method for creating tables in Access is by entering data directly into access table. To create a table by this method, rename the fields and enter data directly to each field in the table. Once this process is complete, save the table structure.

TASK 6: To Create the Enrolment Table by Entering Data Directly

1. Select the "Create table by entering data" shortcut as shown in Fig. 16.22, and click the Open button.

CamScanner

4. To enter the data for the first record, click the first empty cell in the row. Then type the data that corresponds to the field. Press ↵ on your keyboard to enter the data and move to the next cell. Then type the data that corresponds to the next field. Repeat this step until you finish entering all the data for the record. Repeat this process for each record. Access automatically saves each record you enter.

   TIP Access removes the rows and columns that do not contain data.

5. To save the table click . The Save As dialog box appears as shown in Fig. 16.25. Type Enrolment as the name of the table and click OK.

Table1 : Table

Enrolment No	Roll No	Marks Obtained
gnc/203/2002	1	84
gnc/305/2002	2	36
gnc/204/2002	3	94
gnc/207/2002	4	34
gnc/603/2002	5	76
gnc/104/2001	6	47
gnc/857/2002	7	93
gnc/781/2002	8	56
gnc/2033/2001	9	50
gnc/4852/2002	10	28
gnc/2034/2002	11	69
gnc/102/2002	12	56
gnc/394/2001	13	88
gnc/943/2002	14	79
gnc/782/2001	15	48
gnc/493/2002	16	41
gnc/304/2001	17	85
gnc/593/2001	18	37
gnc/9210/2001	19	39
gnc/348/2001	20	45

Record: 22 of 31

Fig. 16.24

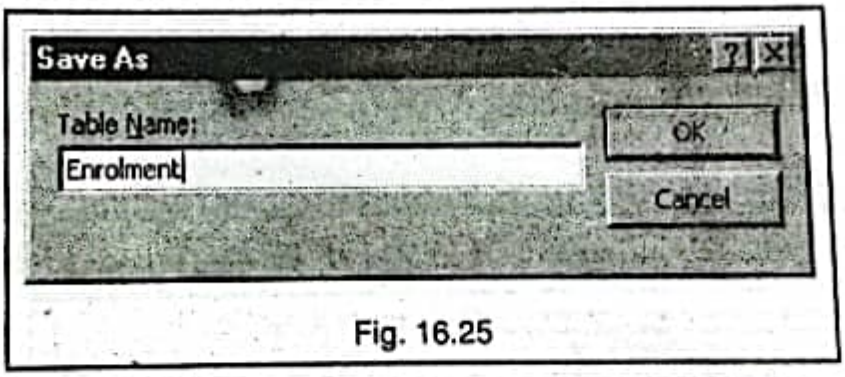


Fig. 16.25

6. A dialog box as shown in Fig. 16.26 appears stating that the table does not have a primary key. If you select Yes, Access adds an ID field to the table to serve as the primary key. The ID field will number each record you add to the table. As our primary key is Roll No so click No.

   TIP You can later create the primary key. To do so, refer to TASK 4.

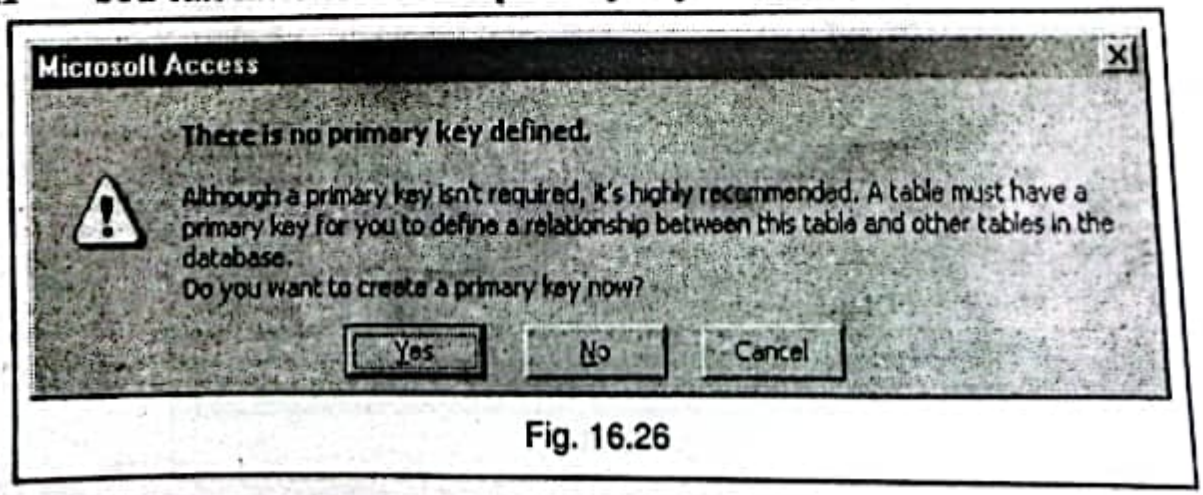


Fig. 16.26

CamScanner

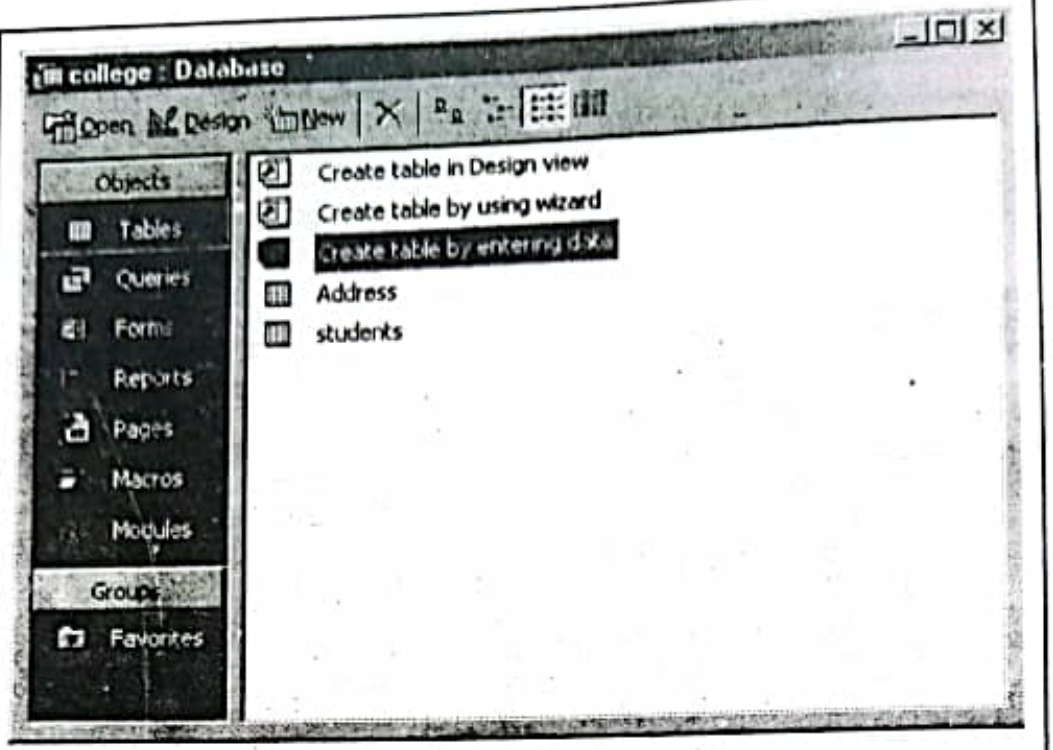


Fig. 16.22

2. A blank table as shown in Fig. 16.23 appears. The top of the blank table displays the field names for each column in the table. To change a filed name, double-click the field name to highlight the name.

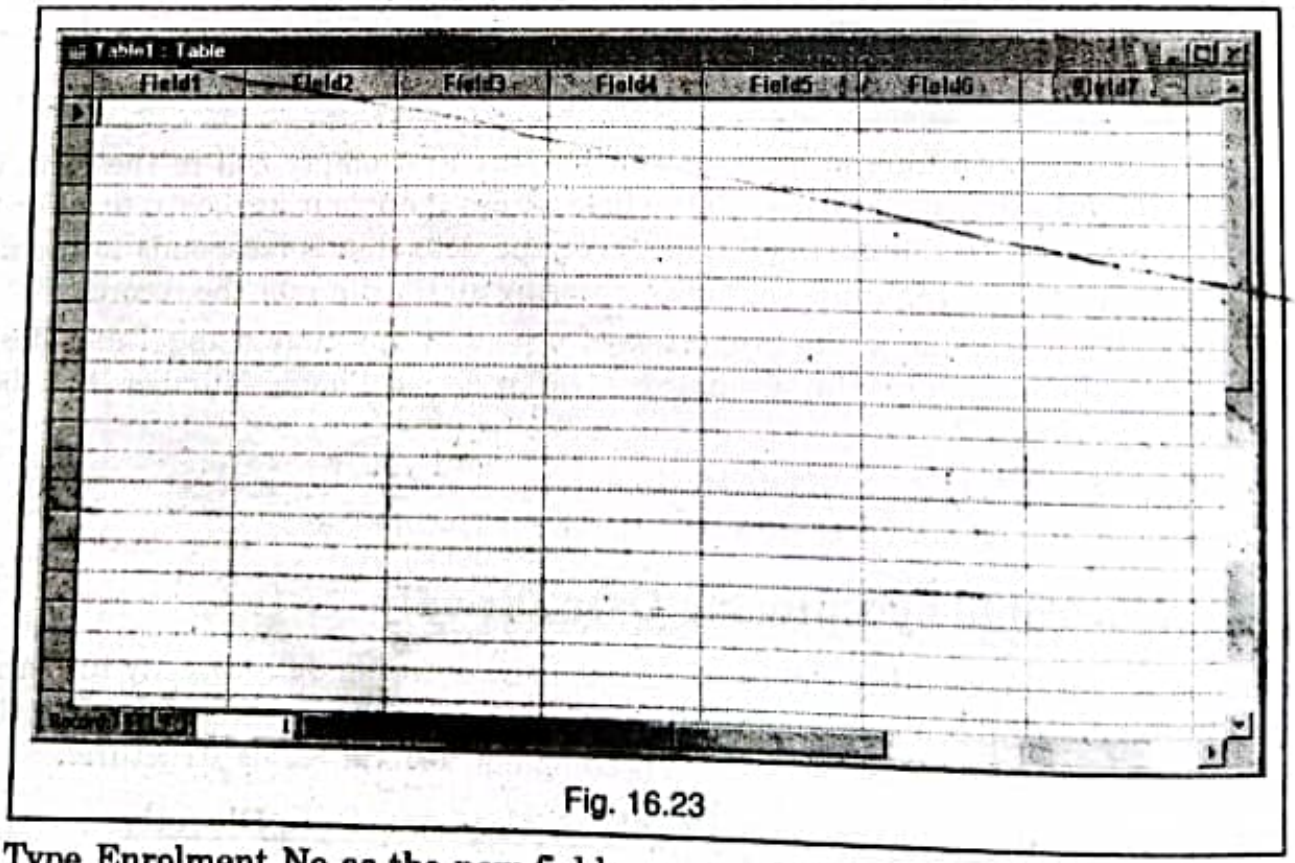


Fig. 16.23

3. Type Enrolment No as the new field name and then press ↵ on your keyboard. Repeat this step for second and third field and type Roll No and Marks Obtained respectively, as shown in Fig. 16.24.

CamScanner

7. Click the Close ☒ button to return to the database window. The table name now appears in the database window, as shown in Fig. 16.27.

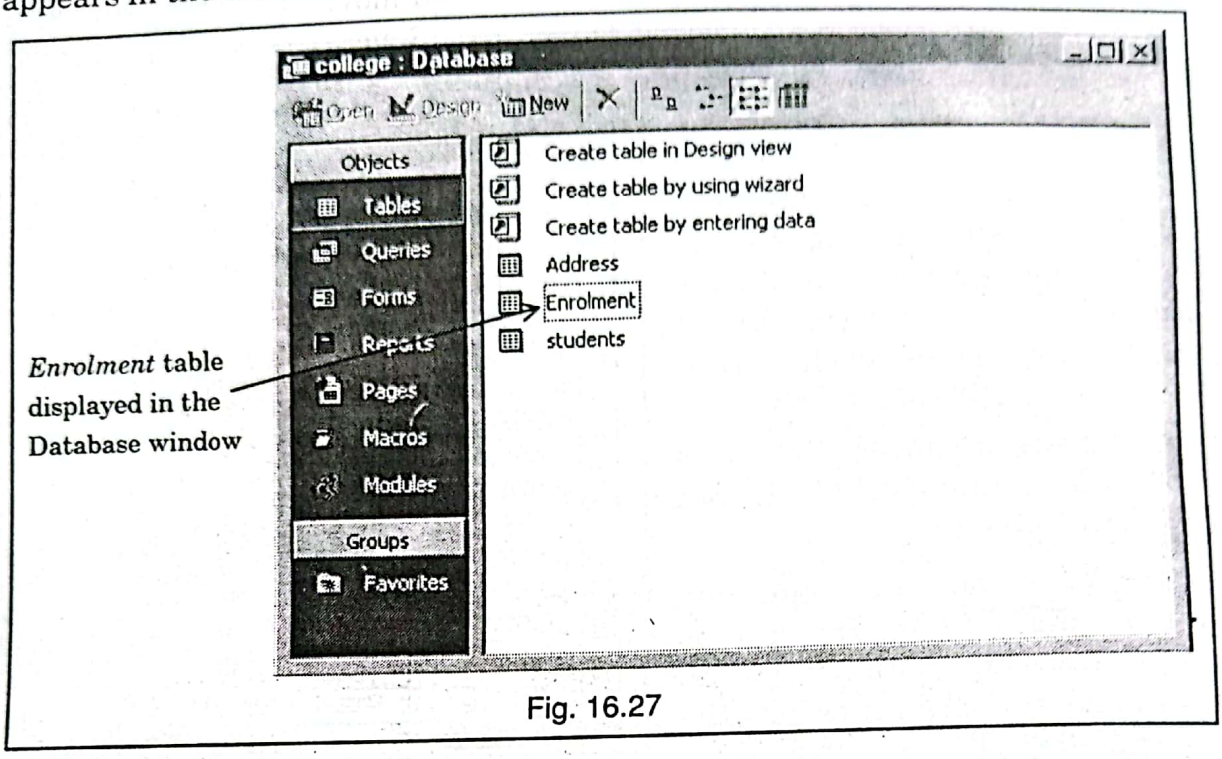


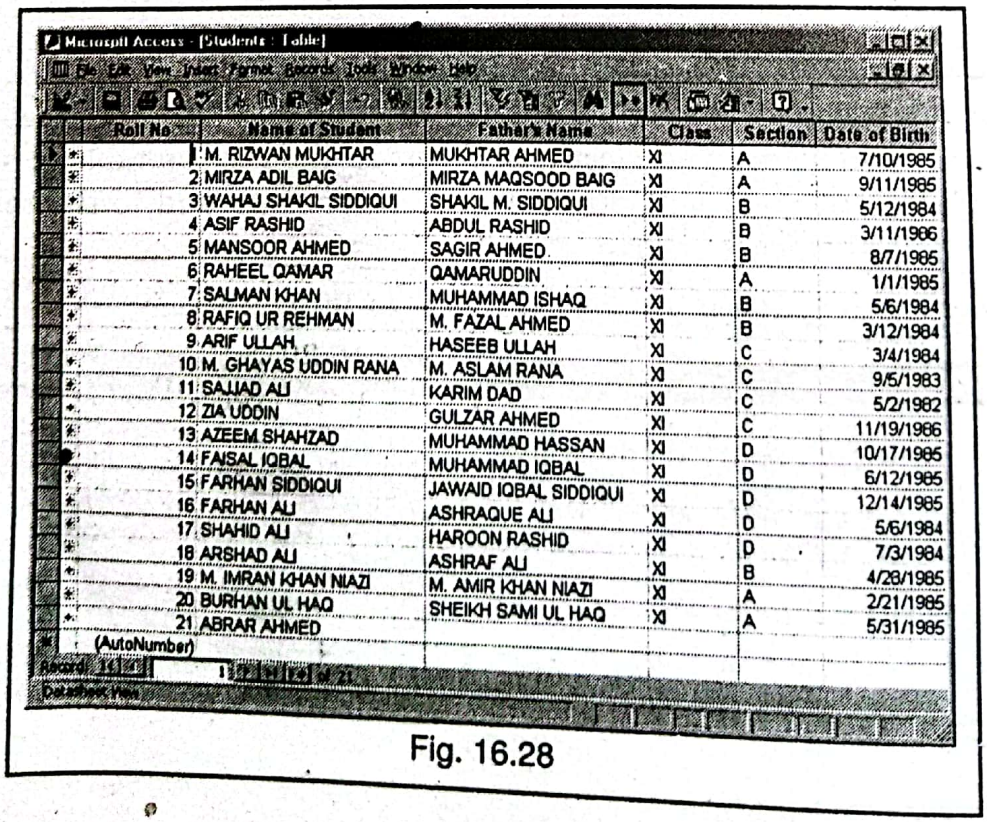
Fig. 16.27

Inserting Records

A common database task is to add or delete records. For example, you may want to add a new student in students table. Records you add appear at the bottom of the database table. You can also create new records by copying existing records and updating field data.

TASK 7: To Insert a Record to Students Table Using Datasheet View

1. Open the table *Students* in the database *college* in Datasheet view shown in Fig. 16.28.



Roll No	Name of Student	Father's Name	Class	Section	Date of Birth
1	M. RIZWAN MUKHTAR	MUKHTAR AHMED	XI	A	7/10/1985
2	MIRZA ADIL BAIG	MIRZA MAQSOOD BAIG	XI	A	9/11/1985
3	WAHAJ SHAKIL SIDDIQUI	SHAKIL M. SIDDIQUI	XI	B	5/12/1984
4	ASIF RASHID	ABDUL RASHID	XI	B	3/11/1986
5	MANSOOR AHMED	SAGIR AHMED	XI	B	8/7/1985
6	RAHEEL QAMAR	QAMARUDDIN	XI	A	1/1/1985
7	SALMAN KHAN	MUHAMMAD ISHAQ	XI	B	5/6/1984
8	RAFIQ UR REHMAN	M. FAZAL AHMED	XI	B	3/12/1984
9	ARIF ULLAH	HASEEB ULLAH	XI	C	3/4/1984
10	M. GHAYAS UDDIN RANA	M. ASLAM RANA	XI	C	9/5/1983
11	SAJJAD ALI	KARIM DAD	XI	C	5/2/1982
12	ZIA UDDIN	GULZAR AHMED	XI	C	11/19/1986
13	AZEEM SHAHZAD	MUHAMMAD HASSAN	XI	D	10/17/1985
14	FAISAL IQBAL	MUHAMMAD IQBAL	XI	D	6/12/1985
15	FARHAN SIDDIQUI	JAWAID IQBAL SIDDIQUI	XI	D	12/14/1985
16	FARHAN ALI	ASHRAQUE ALI	XI	D	5/6/1984
17	SHAHID ALI	HAROON RASHID	XI	D	7/3/1984
18	ARSHAD ALI	ASHRAF ALI	XI	B	4/28/1985
19	M. IMRAN KHAN NIAZI	M. AMIR KHAN NIAZI	XI	A	2/21/1985
20	BURHAN UL HAQ	SHEIKH SAMI UL HAQ	XI	A	5/31/1985
21	ABRAR AHMED				
(AutoNumber)					

Fig. 16.28

CamScanner

2. Click the New Record icon to add a new record to your table.
3. Click the first empty cell in the row. Then type the data that corresponds to the field.
4. Press ↵ on your keyboard to enter the data and move to the next cell. Then type the data that corresponds to the next field. Repeat this step until you finish entering all the data for the record.

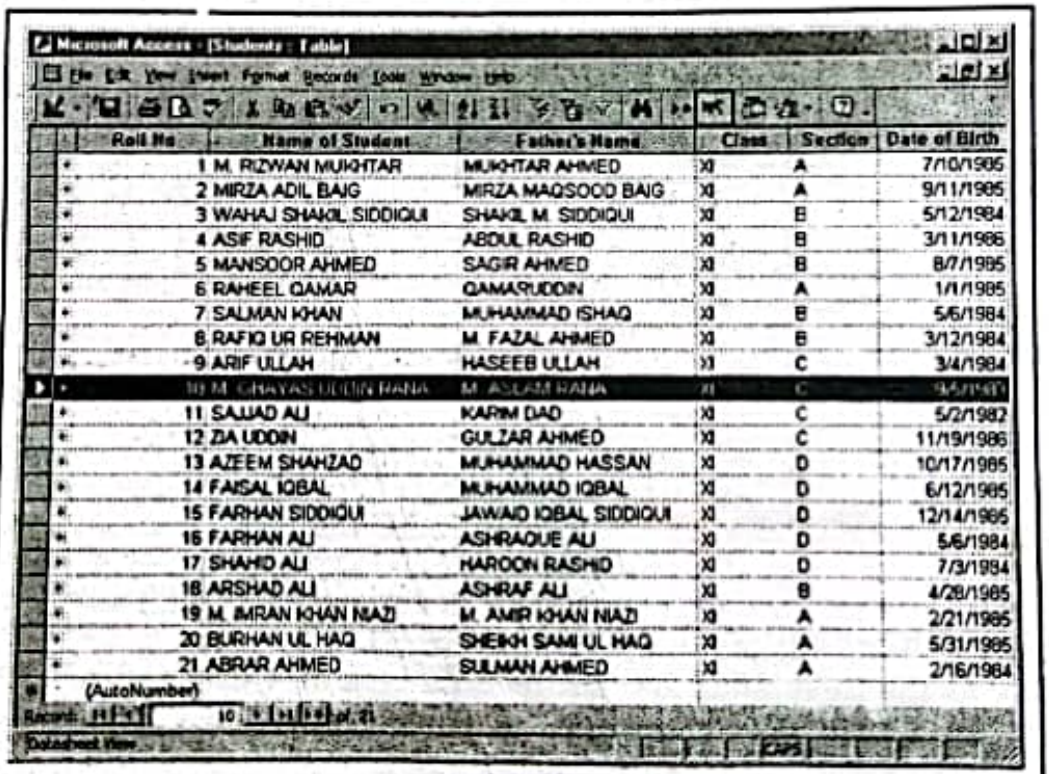
   TIP Access automatically saves each new record you add to the table.

Deleting Records

When records become obsolete or outdate, you should remove the records to keep the database current. In Access, you can select individual records or multiple records and delete them at the same time.

TASK 8: To Delete a Record from Students Table Using Datasheet View

1. Open the table *Students* in the database *college* in Datasheet view.
2. Position the mouse over the area to the left of the record you want to delete (changes to ➔) and then click to select the record as shown in Fig. 16.29.



Roll No	Name of Student	Father's Name	Class	Section	Date of Birth
1	M. RIZWAN MUKHTAR	MUKHTAR AHMED	XI	A	7/10/1985
2	MIRZA ADIL BAIG	MIRZA MAQSOOD BAIG	XI	A	9/11/1985
3	WAHAJ SHAKIL SIDDIQUI	SHAKIL M. SIDDIQUI	XI	B	5/12/1984
4	ASIF RASHID	ABDUL RASHID	XI	B	3/11/1986
5	MANSOOR AHMED	SAGIR AHMED	XI	B	8/7/1985
6	RAHEEL QAMAR	QAMARUDDIN	XI	A	1/1/1985
7	SALMAN KHAN	MUHAMMAD ISHAQ	XI	B	5/6/1984
8	RAFIQ UR REHMAN	M. FAZAL AHMED	XI	B	3/12/1984
9	ARIF ULLAH	HASEEB ULLAH	XI	C	3/4/1984
10	M. GHAYAS UDDIN RANA	M. ASLAM RANA	XI	C	9/4/1985
11	SAJJAD ALI	KARIM DAD	XI	C	5/2/1982
12	ZIA UDDIN	GULZAR AHMED	XI	C	11/19/1986
13	AZEEM SHAHZAD	MUHAMMAD HASSAN	XI	D	10/17/1985
14	FAISAL IQBAL	MUHAMMAD IQBAL	XI	D	6/12/1985
15	FARHAN SIDDIQUI	JAWAID IQBAL SIDDIQUI	XI	D	12/14/1985
16	FARHAN ALI	ASHRAQUE ALI	XI	D	5/6/1984
17	SHAHID ALI	HAROON RASHID	XI	D	7/3/1984
18	ARSHAD ALI	ASHRAF ALI	XI	B	4/28/1985
19	M. IMRAN KHAN NIAZI	M. AMIR KHAN NIAZI	XI	A	2/21/1985
20	BURHAN UL HAQ	SHEIKH SAMI UL HAQ	XI	A	5/31/1985
21	ABRAR AHMED	SULMAN AHMED	XI	A	2/16/1984
	(AutoNumber)				

Fig. 16.29

3. Click to delete the record.
4. A warning dialog box as shown in Fig. 16.30 appears to confirm the deletion. Click Yes to permanently delete the record. The record disappears.

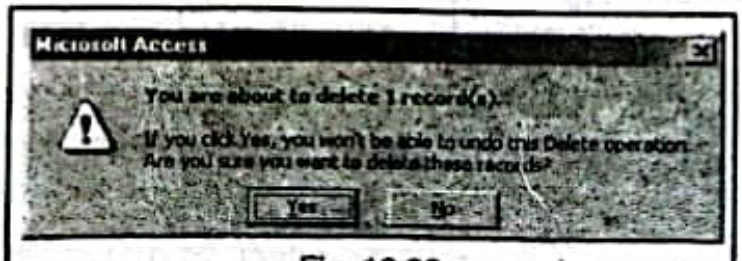


Fig. 16.30

CamScanner

Inserting Fields

You can easily add a field to your table when you want to include an additional category of information.

TASK 9: To Insert a Field to Enrolment Table

1. Open the table *Enrolment* in the database *college*. Click the Table Design view Design button on the toolbar.
2. Right click on the field before which the new field has to be inserted. In this case, we want the field to be inserted before the 'Marks Obtained', so right click on the 'Marks Obtained' field. A Menu as shown in Fig. 16.31 will appear.

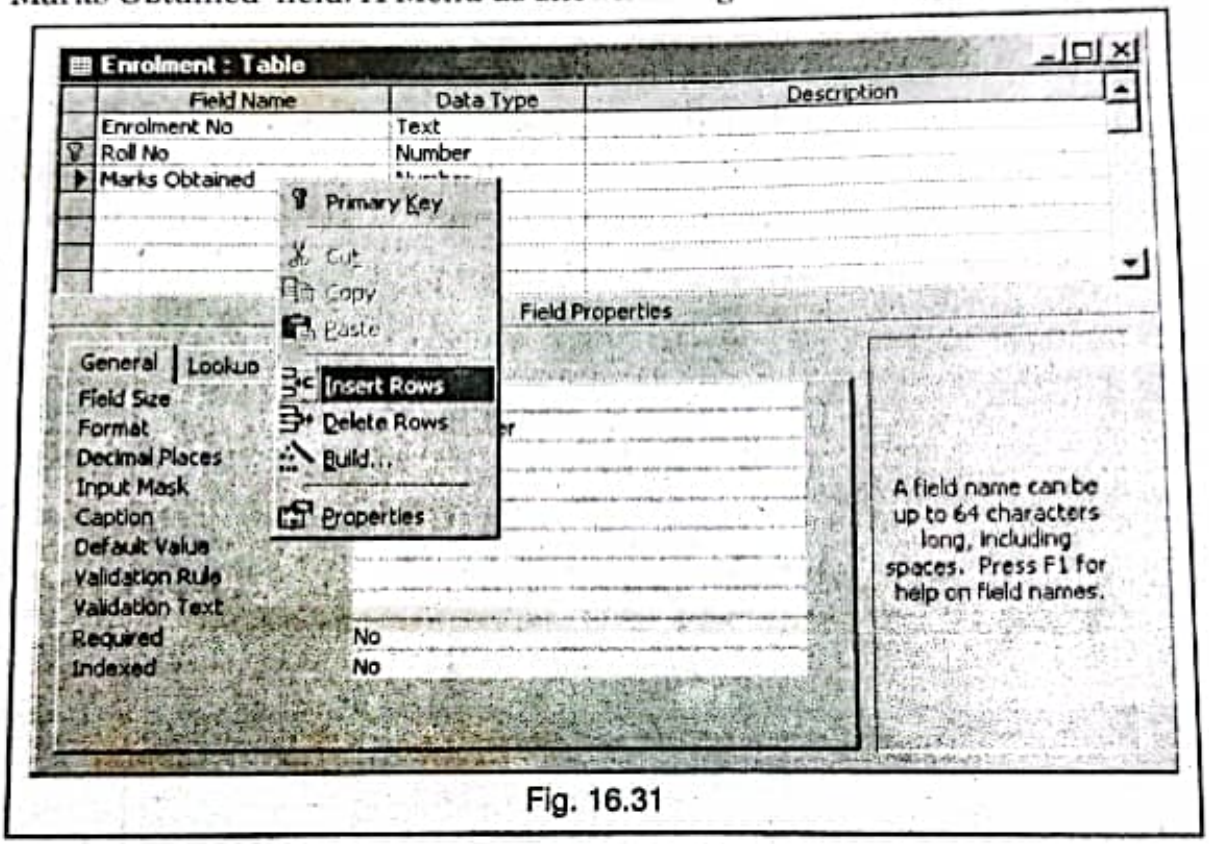


Fig. 16.31

3. Click the Insert Rows button. A blank row appears.
4. Enter 'Dues' as the field name and let the data type be 'Currency'. Click the Save button on the toolbar to save changes to the table design.
5. Click the View button on the toolbar and enter the data in this field.

Deleting Fields

If you no longer need a field, you can delete the field from your table. Once you delete a field from a table you cannot restore the field by clicking the Undo button, so think before you delete.

TASK 10: To Delete a Field from Enrolment Table

1. Open the table *Enrolment* in the database *college*. Click the Table Design view Design button on the toolbar.
2. Right click on the field which you want to delete. In this case, we want to delete

CamScanner

'Marks Obtained' field, so right click on the 'Marks Obtained' field. A Menu as shown in Fig. 16.32 will appear.

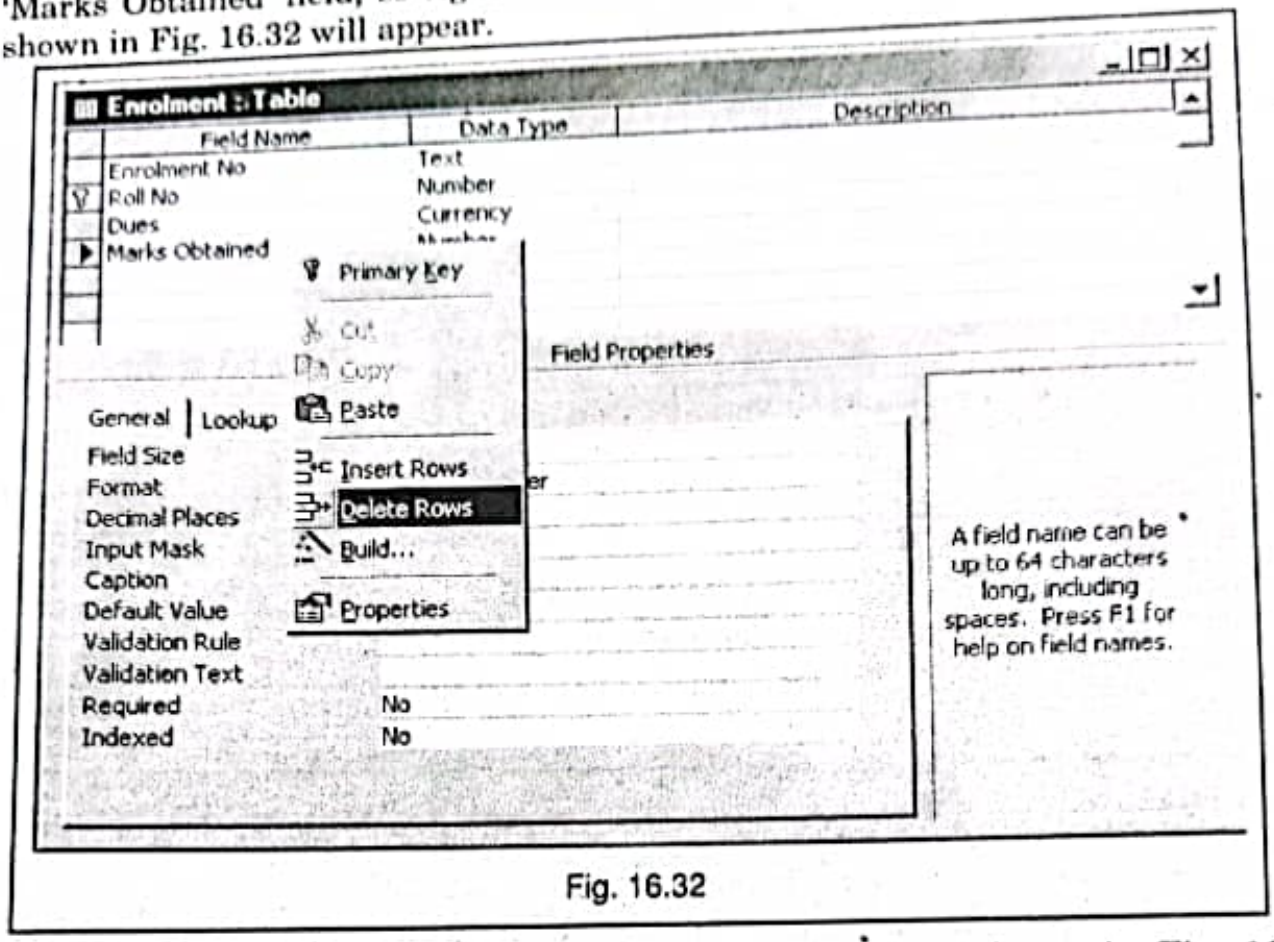


Fig. 16.32

3. Click the Delete Rows button. A warning dialog box as shown in Fig. 16.33 appears to confirm the deletion. Click Yes to permanently delete the field.

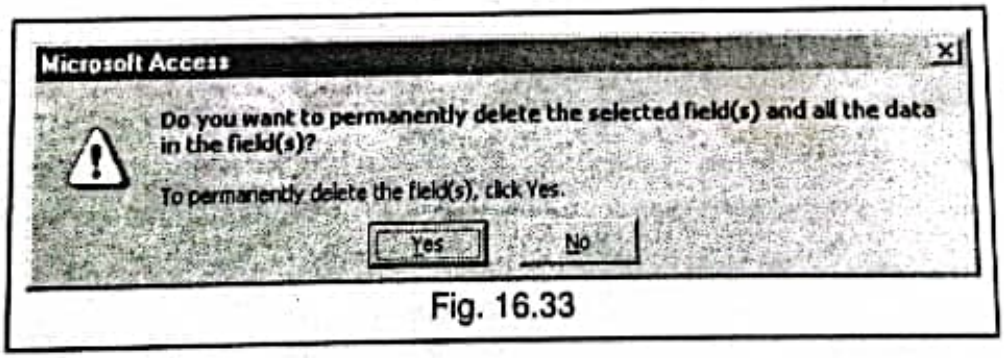


Fig. 16.33

4. Click the Save button on the toolbar to save changes to the table design.
5. Click the View button on the toolbar to see resultant table.

Traversing the Table Records Using Datasheet View

Database designers frequently use Datasheet view to interact with records of a table. Access supports multiple methods for traversing among records. To *traverse* among records is to visit each record of a table exactly once. Datasheet view includes navigation buttons, or traversing buttons, that you can use to move among records. You can use these buttons to move to the first, previous, next, and last records in the table. You can also use the New Record button to add records to the table. Fig. 16.34 shows the navigation buttons appearing in the lower-left corner of the Students table when it is opened in Datasheet view.

CamScanner

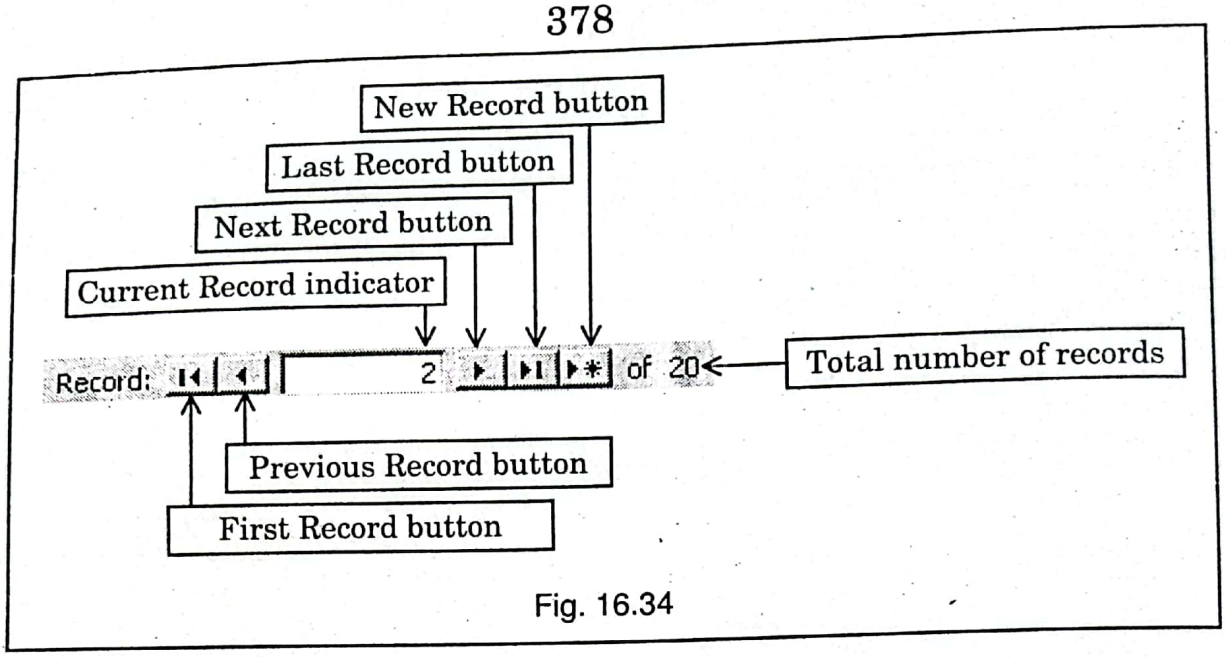


Fig. 16.34

TASK 11: To Traverse among Records in Datasheet View

1. Open the table *Students* in the database *college* in Datasheet view.
2. The datasheet's Record Selector appears next to the first record, as shown in Fig. 16.35.

Record Selector

Students : Table

Roll No	Name of Student	Father's Name	Class	Section	Date of Birth
1	M. RIZWAN MUKHTAR	MUKHTAR AHMED	XI	A	7/10/1985
2	MIRZA ADIL BAIG	MIRZA MAQSOOD BAIG	XI	A	9/11/1985
3	WAHAJ SHAKIL SIDDIQUI	SHAKIL M. SIDDIQUI	XI	B	5/12/1984
4	ASIF RASHID	ABDUL RASHID	XI	B	3/11/1986
5	MANSOOR AHMED	SAGIR AHMED	XI	B	8/7/1985
6	RAHEEL QAMAR	QAMARUDDIN	XI	A	1/1/1985
7	SALMAN KHAN	MUHAMMAD ISHAQ	XI	B	5/6/1984
8	RAFIQ UR REHMAN	M. FAZAL AHMED	XI	B	3/12/1964
9	ARIF ULLAH	HASEEB ULLAH	XI	C	3/4/1984
10	M. GHAYAS UDDIN RANA	M. ASLAM RANA	XI	C	9/5/1983
11	SAJJAD ALI	KARIM DAD	XI	C	5/2/1982
12	ZIA UDDIN	GULZAR AHMED	XI	C	11/19/1986
13	AZEEM SHAHZAD	MUHAMMAD HASSAN	XI	D	10/17/1985
14	FAISAL IQBAL	MUHAMMAD IQBAL	XI	D	6/12/1985
15	FARHAN SIDDIQUI	JAWAID IQBAL SIDDIQUI	XI	D	12/14/1985
16	FARHAN ALI	ASHRAQUE ALI	XI	D	5/6/1984
17	SHAHID ALI	HAROON RASHID	XI	D	7/3/1984
18	ARSHAD ALI	ASHRAF ALI	XI	B	4/28/1985
19	M. IMRAN KHAN NIAZI	M. AMIR KHAN NIAZI	XI	A	2/21/1985
20	BURHAN UL HAQ	SHEIKH SAMI UL HAQ	XI	A	5/31/1985
(AutoNumber)					

Record: 1 of 20

Fig. 16.35

3. Click the Next Record button. The next record in the database is selected. Proceed in the same way to traverse remaining records.
4. Click the Last Record button. The last record in the database is selected.

TIP The Record Selector will at times display these symbols:

CamScanner

Record Selector

Click to select the entire record. Also, can display the following symbols:

	The current record; the record has been saved as it appears.
	A new record that you can enter information in.
	You are editing this record; changes are not yet saved.
	This record is locked by another user; you cannot edit it (Microsoft Access database only).

5. You can also move to a specific record by entering a valid record number in the current record indicator.
6. Double-click inside the Current Record indicator to highlight the value, and type 4 as the record number, as shown in Fig. 16.36. Press ↵. Record 4 will be the current record.

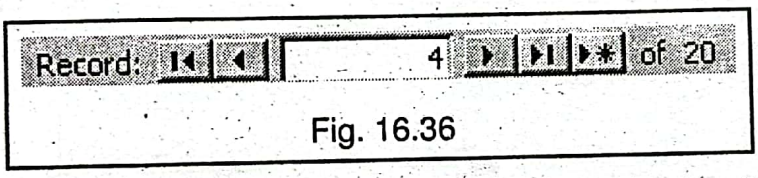


Fig. 16.36

TIP Once a record is current, you may edit its contents by tabbing to the appropriate field in Datasheet view.

7. Close the Students table.

Searching for Records

As with many databases, record data will change over time. This is particularly true for database tables containing name and address information. Although it is not difficult to find a specific record in a table that has a small number of records, when a table contains hundreds or thousands of records, this method becomes impractical.

Fortunately, Access contains search capabilities. Using the *Find* dialog box to search for a specific record and edit, or change, its data is a common database maintenance task. Editing field data is one way of updating records. Updating also includes adding and deleting records. You will need to update the Address table by changing section of Farhan Ali, as he has shifted recently from Section D to Section A.

TASK 12: To Search a Table for a Specific Record

1. Open the table *Students* in the database *college* in Datasheet view.
2. Choose Find from the Edit menu, as shown in Fig. 16.37.
3. The Find and Replace dialog box appears. Type *Farhan Ali* as the search term, select *Students: Table* in the Look in, Drop-down list and select Match *Any Part of Field*, as shown in Fig. 16.38.

CamScanner

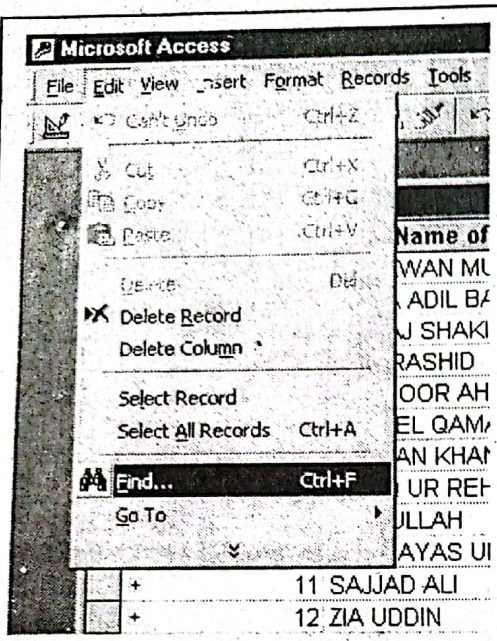


Fig. 16.37

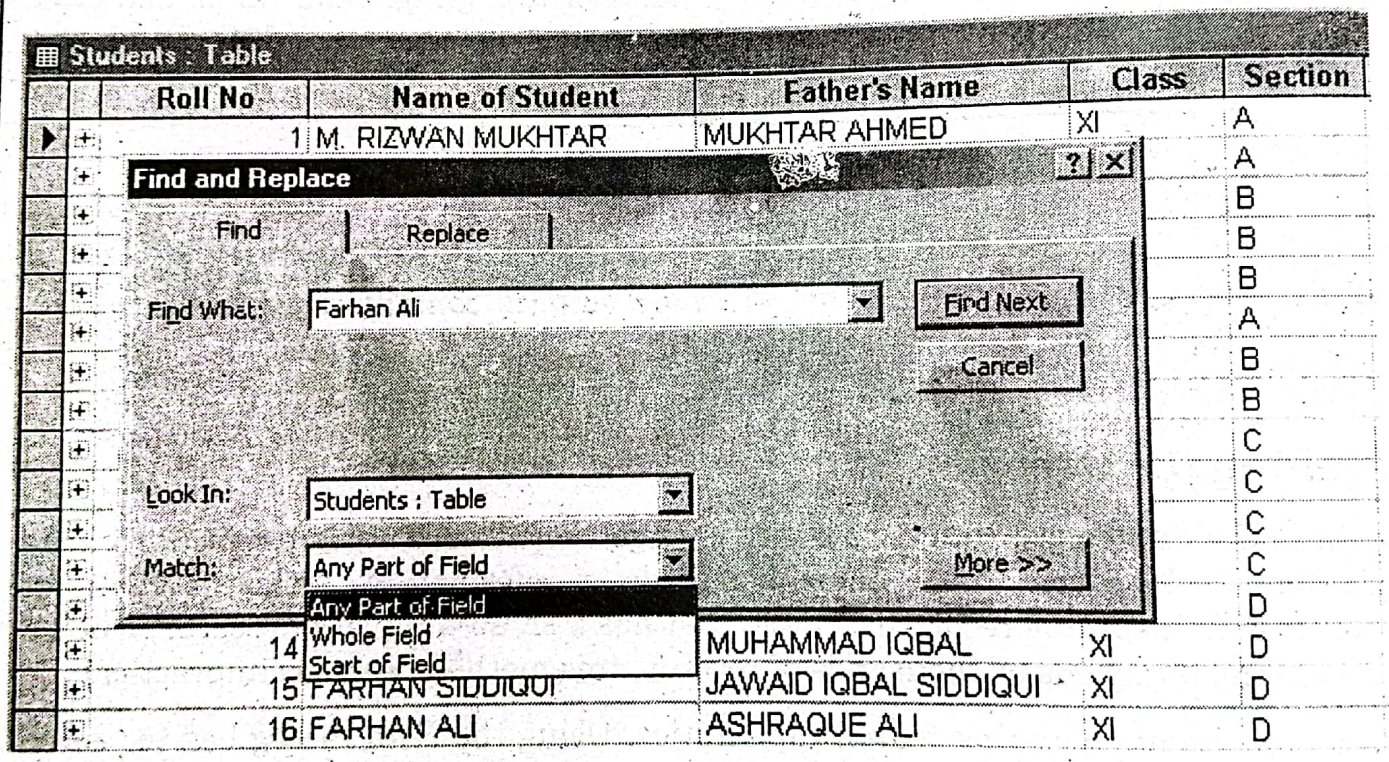


Fig. 16.38

Updating the Records in a Table

Once you find a record using the Find option, you can edit it in the same way you edit any record displayed in a table. By entering a new value, you can replace the selected text with a new entry.

TASK 13: To Update Records

1. In Farhan Ali's record double-click the D in the Section field.
2. Type A as the new section, and press Enter. The record will be updated once you move the focus to another field or record.
3. Click the First Record button.

CamScanner

Sorting Records

Another common database maintenance task is sorting records. When you apply a sort, you reorder the records in a table according to the entries in a specific field. The field you select for a sort specifies the sort criteria. Records can be sorted in either ascending or descending order. When you sort in ascending order, names and terms are sorted from A to Z, and dates and times are sorted from earlier to later. When you sort by descending order, the opposite is true.

When you sort records, Access saves the sort order when you save the datasheet, and reapplies it automatically when you reopen the datasheet.

TASK 14: To Sort Records in a Table

1. Open the table *Students* in the database *college* in Datasheet view.
2. Place the insertion point somewhere in the Name of Student field, or select the entire column by clicking the Name of Student field selector, as shown in Fig. 16.39.

Students : Table

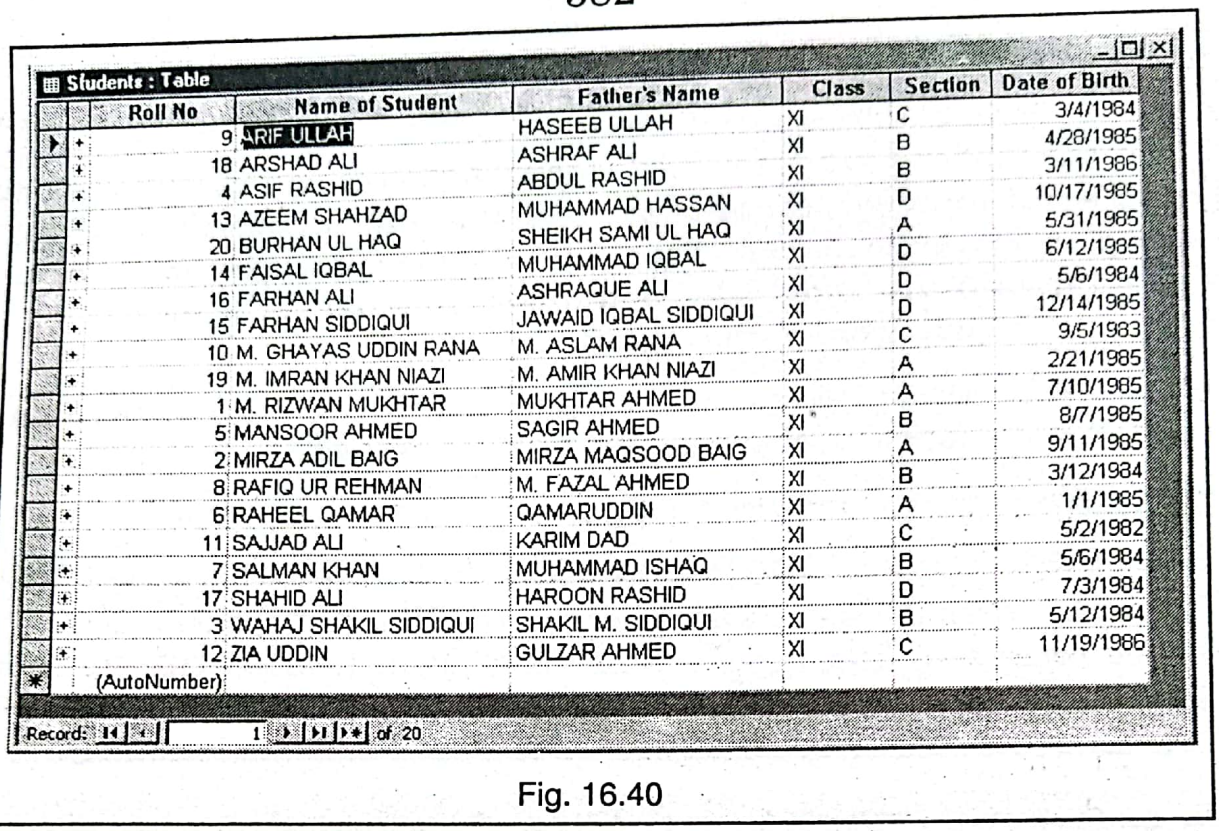
Roll No	Name of Student	Father's Name	Class	Section	Date of Birth
1	M. RIZWAN MUKHTAR	MUKHTAR AHMED	XI	A	7/10/1985
2	MIRZA ADIL BAIG	MIRZA MAQSOOD BAIG	XI	A	9/11/1985
3	WAHAJ SHAKIL SIDDIQUI	SHAKIL M. SIDDIQUI	XI	B	5/12/1984
4	ASIF RASHID	ABDUL RASHID	XI	B	3/11/1986
5	MANSOOR AHMED	SAGIR AHMED	XI	B	8/7/1985
6	RAHEEL QAMAR	QAMARUDDIN	XI	A	1/1/1985
7	SALMAN KHAN	MUHAMMAD ISHAQ	XI	B	5/6/1984
8	RAFIQ UR REHMAN	M. FAZAL AHMED	XI	B	3/12/1984
9	ARIF ULLAH	HASEEB ULLAH	XI	C	3/4/1984
10	M GHAYAS UDDIN RANA	M. ASLAM RANA	XI	C	9/5/1983
11	SAJJAD ALI	KARIM DAD	XI	C	5/2/1982
12	ZIA UDDIN	GULZAR AHMED	XI	C	11/19/1986
13	AZEEM SHAHZAD	MUHAMMAD HASSAN	XI	D	10/17/1985
14	FAISAL IQBAL	MUHAMMAD IQBAL	XI	D	6/12/1985
15	FARHAN SIDDIQUI	JAWAID IQBAL SIDDIQUI	XI	D	12/14/1985
16	FARHAN ALI	ASHRAQUE ALI	XI	D	5/6/1984
17	SHAHID ALI	HAROON RASHID	XI	D	7/3/1984
18	ARSHAD ALI	ASHRAF ALI	XI	B	4/28/1985
19	M IMRAN KHAN NIAZI	M. AMIR KHAN NIAZI	XI	A	2/21/1985
20	BURHAN UL HAQ	SHEIKH SAMI UL HAQ	XI	A	5/31/1985
(AutoNumber)					

Record: 1 of 20

Fig. 16.39

3. Click the Sort Ascending button on the Table Datasheet toolbar.
4. The records are now sorted in ascending order by Name of Student, as shown Fig. 16.40.
5. Close the table.

CamScanner



Students : Table

Roll No	Name of Student	Father's Name	Class	Section	Date of Birth
9	ARIF ULLAH	HASEEB ULLAH	XI	C	3/4/1984
18	ARSHAD ALI	ASHRAF ALI	XI	B	4/28/1985
4	ASIF RASHID	ABDUL RASHID	XI	B	3/11/1986
13	AZEEM SHAHZAD	MUHAMMAD HASSAN	XI	D	10/17/1985
20	BURHAN UL HAQ	SHEIKH SAMI UL HAQ	XI	A	5/31/1985
14	FAISAL IQBAL	MUHAMMAD IQBAL	XI	D	6/12/1985
16	FARHAN ALI	ASHRAQUE ALI	XI	D	5/6/1984
15	FARHAN SIDDIQUI	JAWAID IQBAL SIDDIQUI	XI	D	12/14/1985
10	M. GHAYAS UDDIN RANA	M. ASLAM RANA	XI	C	9/5/1983
19	M. IMRAN KHAN NIAZI	M. AMIR KHAN NIAZI	XI	A	2/21/1985
1	M. RIZWAN MUKHTAR	MUKHTAR AHMED	XI	A	7/10/1985
5	MANSOOR AHMED	SAGIR AHMED	XI	B	8/7/1985
2	MIRZA ADIL BAIG	MIRZA MAQSOOD BAIG	XI	A	9/11/1985
8	RAFIQ UR REHMAN	M. FAZAL AHMED	XI	B	3/12/1984
6	RAHEEL QAMAR	QAMARUDDIN	XI	A	1/1/1985
11	SAJJAD ALI	KARIM DAD	XI	C	5/2/1982
7	SALMAN KHAN	MUHAMMAD ISHAQ	XI	B	5/6/1984
17	SHAHID ALI	HAROON RASHID	XI	D	7/3/1984
3	WAHAJ SHAKIL SIDDIQUI	SHAKIL M. SIDDIQUI	XI	B	5/12/1984
12	ZIA UDDIN	GULZAR AHMED	XI	C	11/19/1986
(AutoNumber)					

Record: 1 of 20

Fig. 16.40

6. The Microsoft Access dialog box shown in Fig. 16.41 appears. Since you want the table design to be modified, click Yes.

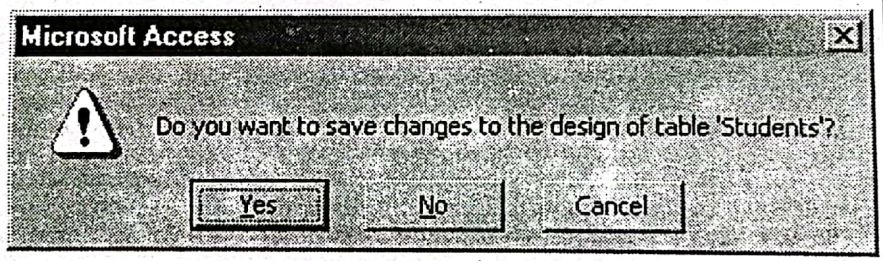


Fig. 16.41

Any time you open the table, the data will now appear in alphabetical order by Name of Student. This is because you modified the table structure by specifying a sort property for the table. In the event that you would ever need to change this property, you need to know where to change it.

CamScanner

Exercise

16.1. What is the database window?
16.2. How many object classes are there in Access?
16.3. All database objects are displayed in which screen element?
16.4. Which database object is used to create and format printed output?
16.5. Microsoft Access is based upon which model?
16.6. How do fields and records differ?
16.7. How does a database differ from a table?
16.8. How does a record differ from a field?
16.9. What is the default data type listed in the Table Design window?
16.10. Where do you change the properties of a field in a table?
16.11. Why is the AutoNumber data type often used for a table's primary key?
16.12. How do you create a table in Access?
16.13. What is displayed on the screen when you open an Access database file?
16.14. Where in the database window are tables displayed?
16.15. How do you add a new record to a table using Datasheet view?
16.16. How does inserting fields in Table Design view affect the Datasheet view?
16.17. Can you make a field a key field after you have added data to a table?
16.18. What button on the toolbar do you use to switch from Datasheet view to Table Design view and back?
16.19. When you change the table structure, what must you do before retuning to Datasheet view?
16.20. What is the difference between finding and replacing data?
16.21. What is the difference between finding and replacing?
16.22. What is an easy method for locating records to update?
16.23. What is the most efficient way to add new records that closely resemble existing records?
16.24. To sort records from highest to lowest based on numeric values in a Number field, which sort button on the toolbar would you use?

CamScanner

Objectives and MCQs

Choose the answer that best completes the statement.

16.1. Which Access object is considered primary?
- (a) table
- (b) query
- (c) form
- (d) report

16.2. Which screen element is used to easily design or open the objects within a database?
- (a) close button
- (b) status bar
- (c) database toolbar
- (d) database window

16.3. A database contains the first name, last name, and phone number for a group of students. The first name entries are what kind of data?
- (a) fields
- (b) records
- (c) queries
- (d) tables

16.4. During which phase of the database design are the output specifications identified?
- (a) defining the purpose
- (b) planning the objects
- (c) creating and relating tables
- (d) creating queries

16.5. How many database files can be open simultaneously?
- (a) one
- (b) two
- (c) three
- (d) four

16.6. Which data type is used to store long text entries?
- (a) text
- (b) memo
- (c) numeric
- (d) OLE object

16.7. The upper pane in the Table Design window is used to enter
- (a) field names
- (b) field data types
- (c) field properties
- (d) a and b

CamScanner

16.8. Which of the following statements is false?
(a) When you open an Access database, the database window is displayed
(b) To create database objects, you must first open the database file containing them.
(c) All Access database objects are contained in one file.
(d) When you open a database file, a new table is automatically created.

16.9. What field type uses a checkbox to store field data?
(a) text
(b) number
(c) date/time
(d) yes/no

16.10. Which of the following data types allows you to specify a field size?
(a) text
(b) number
(c) date/Time
(d) currency

16.11. You are designing a table containing name and address information. How many characters should you reserve for the Name field, which is a text data type?
(a) 255
(b) 50
(c) 10
(d) 5

16.12. Which view do you use to enter data into a table?
(a) Table Design view
(b) Datasheet view
(c) Database Location view
(d) Database Design view

16.13. A warning appears when you
(a) insert a field
(b) delete a field
(c) edit field data
(d) change fields

16.14. The Undo feature cannot reverse
(a) inserting a field
(b) editing a field
(c) deleting a field
(d) rearranging fields

16.15. To insert new fields in a database table,
(a) select the field that follows the new field, and click the Insert Rows button on the toolbar.
(b) select the field that follows the new field, and press Insert.

CamScanner

(c) select a record in the table, and press Enter.
(d) select the field that follows the new field, and choose File, Insert New Record.

16.16. Key fields must contain
(a) filtered data
(b) default values
(c) unique data
(d) presorted information

16.17. To locate a record in a database table quickly, use the
(a) Find feature
(b) Replace feature.
(c) Datasheet
(d) Print Preview feature

16.18. Options available in the Find dialog box enable you to do all the following except
(a) match any part of the field
(b) move up or down from the active data field
(c) replace field contents automatically
(d) search for text with capitalization that exactly matches the text typed

16.19. Updating records includes all the following activities *except*
(a) changing an address
(b) creating a new database
(c) inserting new records
(d) removing obsolete records

16.20. To delete a record from a table,
(a) select a field in the table and press Enter
(b) select a field in a form and press Delete
(c) click the frame button or record selection bar and press Delete
(d) select Datasheet View and press Enter

16.21. Which of the following is not true concerning the sort feature in Access?
(a) Records can be sorted in ascending or descending order according to the values in a specific field.
(b) Information about the sort can be saved with the table design properties.
(c) Dates are sorted from most recent to least recent when using the Sort Descending feature.
(d) The case of text entries does not matter when sorting records.

16.22. To identify the field you want to use for sorting records,
(a) click the Sort Ascending button
(b) select the entire table
(c) position the insertion point somewhere in the desired field
(d) click the Sort Descending button

CamScanner

16.23. To restore a deleted record,
(a) press Ctrl+Page Down
(b) close the database without saving and reopen it
(c) click the Undo button
(d) retype the record

Objectives

16.1. Microsoft Access is a relational database management system.
16.2. A database is a collection of information related to a common purpose.
16.3. Access includes two methods for creating a new database.
16.4. A table is the primary object in Access.
16.5. Table structure includes field names and data types.
16.6. Access supports ten data types.
16.7. Record data can be entered, displayed, and edited in Datasheet view.
16.8. A key field will never contain duplicate entries.
16.9. You can use Table Design view to define a table.
16.10. In Datasheet view, each row represents a record.
16.11. The AutoNumber data type is often used to define a table's primary key.
16.12. In Datasheet view, each column represents a field.
16.13. You can use the Access Find feature to locate records and the Access Replace feature to update information automatically or to change data.
16.14. Updating data contained in records in a table ensures accuracy of information.
16.15. Records you insert automatically appear at the end of the database table. You can reposition new records by performing a sort routine.
16.16. Sorting enables you to rearrange records by listing them alphabetically or numerically in ascending or descending order by selected fields.

CamScanner