



SURESH
GYAN VIHAR
UNIVERSITY
Accredited by NAAC with 'A+' Grade

Bachelor of Computer Application

(B.C.A.)

Principles of Programming & Algorithm

Semester-I

Author- Poonam Ponde

SURESH GYAN VIHAR UNIVERSITY
Centre for Distance and Online Education
Mahal, Jagatpura, Jaipur-302025

EDITORIAL BOARD (CDOE, SGVU)

Dr (Prof.) T.K. Jain
Director, CDOE, SGVU

Dr. Dev Brat Gupta
*Associate Professor (SILS) & Academic
Head, CDOE, SGVU*

Ms. Hemlalata Dharendra
Assistant Professor, CDOE, SGVU

Ms. Kapila Bishnoi
Assistant Professor, CDOE, SGVU

Dr. Manish Dwivedi
*Associate Professor & Dy, Director,
CDOE, SGVU*

Mr. Manvendra Narayan Mishra
*Assistant Professor (Deptt. of Mathematics)
SGVU*

Ms. Shreya Mathur
Assistant Professor, CDOE, SGVU

Mr. Ashphaq Ahmad
Assistant Professor, CDOE, SGVU

Published by:

S. B. Prakashan Pvt. Ltd.

WZ-6, Lajwanti Garden, New Delhi: 110046

Tel.: (011) 28520627 | Ph.: 9205476295

Email: info@sbprakashan.com | Web.: www.sbprakashan.com

© SGVU

All rights reserved.

No part of this book may be reproduced or copied in any form or by any means (graphic, electronic or mechanical, including photocopying, recording, taping, or information retrieval system) or reproduced on any disc, tape, perforated media or other information storage device, etc., without the written permission of the publishers.

Every effort has been made to avoid errors or omissions in the publication. In spite of this, some errors might have crept in. Any mistake, error or discrepancy noted may be brought to our notice and it shall be taken care of in the next edition. It is notified that neither the publishers nor the author or seller will be responsible for any damage or loss of any kind, in any manner, therefrom.

For binding mistakes, misprints or for missing pages, etc., the publishers' liability is limited to replacement within one month of purchase by similar edition. All expenses in this connection are to be borne by the purchaser.

Designed & Graphic by : S. B. Prakashan Pvt. Ltd.

Printed at :

Semester 1

Principles of Programming & Algorithms

Learning Objectives

- Elucidate the basic architecture and functionalities of a computer
- Apply programming constructs of C language to solve the real-world problems
- Explore user-defined data structures like arrays, structures and pointers in implementing solutions to problems
- Design and Develop Solutions to problems using structured programming constructs such as functions and procedures

UNIT-I

Introduction to „C“ Language History, Structures of Programming, Function as building blocks. Language Fundamentals Character set, C Tokens, Keywords, Identifiers, Variables, Constant, Data Types, Comments.

UNIT-II

Operators Types of operators, Precedence and Associativity, Expression, Statement and types of statements Build in Operators and function Console based I/O and related built in I/O function: printf(), scanf(), getch(), getchar(), putchar(); Concept of header files, Preprocessor directives: #include, #define. Control structures Decision making structures.

UNIT-III Introduction to problem solving Concept: problem solving, Problem solving techniques (Trail & Error, Brain Storming, Divide & Conquer) Steps in problem solving (Define Problem, Analyze Problem, Explore Solution) Algorithms and Flowcharts (Definitions, Symbols), Characteristics of an algorithm Conditionals in pseudo-code, Loops in pseudo code Time complexity: Big-Oh notation, efficiency Simple Examples: Algorithms and flowcharts (Real Life Examples)

UNIT-IV

Simple Arithmetic Problems Addition / Multiplication of integers, Determining if a number is +ve / -ve / even / odd, Maximum of 2 numbers, 3 numbers, Sum of first n numbers, given n numbers, Integer division, Digit reversing, Table generation for n, a n C b , Factorial, sine series, cosine series, r , Pascal Triangle, Prime number, Factors of a number, Other problems such as Perfect number, GCD numbers etc (Write algorithms and draw flowchart), Swapping

UNIT-V

Functions Basic types of function, Declaration and definition, Function call, Types of function, Parameter passing, Call by value, Call by reference, Scope of variable, Storage classes, Recursion.

References

- Computer fundamentals and programming in c, “Reema Thareja”, Oxford University, Second edition, 2017.
- E. Balaguruswamy, Programming in ANSI C, 7th Edition, Tata McGraw-Hill.
- Brian W. Kernighan and Dennis M. Ritchie, The ‘C’ Programming Language, Prentice Hall of India.
- elearning.vtu.ac.in/econtent/courses/video/BS/15PCD23.html
- <https://nptel.ac.in/courses/106/105/106105171/> MOOC courses can be adopted for more clarity in understanding the topics and verities of problem solving methods.

CONTENTS

Total Pages

| | | |
|----------|---|-----------|
| 1 | Introduction To C Language | 12 |
| 1.1 | Introduction to 'C' Language | 1-1 |
| 1.2 | Application Areas | 1-5 |
| 1.3 | Features of 'C' | 1-5 |
| 1.4 | Program Development Cycle | 1-6 |
| 1.5 | Structure of a 'C' Program | 1-8 |
| 2 | Language Fundamentals | 12 |
| 2.1 | 'C' Character Set | 2-1 |
| 2.2 | C Tokens | 2-1 |
| 2.3 | Identifiers and Keywords | 2-2 |
| 2.4 | Constants | 2-3 |
| 2.5 | Variables | 2-6 |
| 2.6 | Data Declarations and Definitions | 2-9 |
| 3 | Operators | 16 |
| 3.1 | Operators and Expressions | 3-1 |
| 3.2 | Statements | 3-11 |
| 4 | Built-In Operators and Function | 22 |
| 4.1 | Introduction | 4-1 |
| 4.2 | Character Input and Output | 4-1 |
| 4.3 | String Input and Output [gets() & puts()] | 4-3 |
| 4.4 | General Output / formatted Output (printf) | 4-4 |
| 4.5 | Formatted Input (scanf) | 4-7 |
| 4.6 | Concept of Header Files | 4-8 |
| 4.7 | What is a Preprocessor? | 4-9 |
| 4.8 | Preprocessor Directives | 4-9 |
| 5 | Control Structures | 32 |
| 5.1 | Introduction | 5-1 |
| 5.2 | Selection / Decision making statements | 5-1 |
| 5.3 | Iterative Statements (LOOP Control structure) | 5-11 |
| 5.4 | The for Loop | 5-18 |
| 5.5 | Jump Statements | 5-24 |
| 6 | Introduction to Problem Solving | 20 |
| 6.1 | Introduction | 6-1 |
| 6.2 | Problem Solving Techniques | 6-2 |
| 6.3 | Steps in Problem Solving | 6-5 |
| 6.4 | Algorithms and Flowcharts | 6-7 |
| 6.5 | Characteristics of an Algorithm | 6-9 |
| 6.6 | Conditionals in Pseudocode | 6-10 |
| 6.7 | Loops in Pseudocode | 6-10 |
| 6.8 | Time Complexity | 6-13 |
| 6.9 | Simple Examples: Algorithms.... | 6-17 |
| 7 | Simple Arithmetic Problems | 16 |
| 7.1 | Program for Addition of Two Integers | 7-1 |

| | | |
|----------|--|-----------|
| 7.2 | Program for Multiplication of Two Integers..... | 7-2 |
| 7.3 | Program for Division of Two Integers..... | 7-2 |
| 7.4 | Program for determining Number is +ve or -ve..... | 7-3 |
| 7.5 | Program for determining Number is Odd or Even..... | 7-3 |
| 7.6 | Program for Finding Maximum of Two Numbers..... | 7-4 |
| 7.7 | Program for Finding Maximum of Three Numbers..... | 7-5 |
| 7.8 | Program of Sum of first N Numbers..... | 7-6 |
| 7.9 | Program for Reversing Integer Number..... | 7-6 |
| 7.10 | Program for Table Generation of N Number..... | 7-7 |
| 7.11 | Program for Factorial..... | 7-8 |
| 7.12 | Program for Finding Sine of a Number..... | 7-8 |
| 7.13 | Program for Finding Cosine of a Number..... | 7-9 |
| 7.14 | Program for Combinations..... | 7-9 |
| 7.15 | Program for Permutation..... | 7-10 |
| 7.16 | Program for Pascal Triangle..... | 7-11 |
| 7.17 | Program for Finding Prime Number..... | 7-12 |
| 7.18 | Program to Find Factors of A Number..... | 7-12 |
| 7.19 | Program for Greatest Common Divisor between Two Nos..... | 7-13 |
| 7.20 | Program for Swapping of Two Integers..... | 7-14 |
| 8 | Functions | 22 |
| 8.1 | Introduction..... | 8-1 |
| 8.2 | What is a Function?..... | 8-1 |
| 8.3 | Functions and Structured Programming..... | 8-2 |
| 8.4 | How a function Works?..... | 8-2 |
| 8.5 | Library and User defined Functions..... | 8-3 |
| 8.6 | Function Declaration and Definition..... | 8-5 |
| 8.7 | Writing a Function..... | 8-6 |
| 8.8 | Calling a Function..... | 8-8 |
| 8.9 | Passing Arguments to a Function..... | 8-10 |
| 8.10 | Functions with Variable Arguments..... | 8-12 |
| 8.11 | Command Line Arguments..... | 8-13 |
| 8.12 | Recursion..... | 8-15 |
| 8.13 | Function Returning a Pointer..... | 8-19 |
| 9 | Storage classes | 12 |
| 9.1 | Meaning of Terms..... | 9-1 |
| 9.2 | Scope..... | 9-2 |
| 9.3 | Storage Classes..... | 9-4 |

* * *

INTRODUCTION TO C LANGUAGE

1

1.1

Introduction to 'C' Language

1.1.1 History

The development of C language was a result of the evolution of several languages, which can be called 'the ancestors of C'. These were Algol 60, CPL, BCPL and B.

In the 1960s many computer languages, each for a specific purpose, were developed, e.g COBOL and FORTRAN. The need was felt for a general purpose language that could suit a variety of applications. An international committee set for this purpose, designed Algol 60, which eventually led to the development of C.

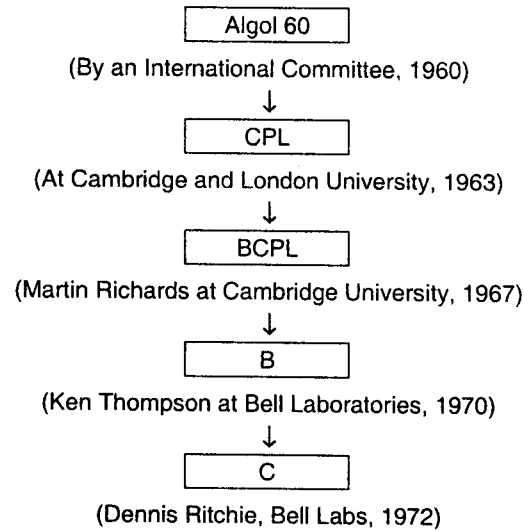
- i. Algol 60 was a modular and structured language but it did not succeed because it was found to be too abstract and too general.
- ii. The **Combined Programming Language (CPL)** developed at Cambridge University and University of London in 1963 was a successor of Algol 60. However it was hard to learn and difficult to implement.
- iii. The **Basic Combined Programming Language (BCPL)** was very close to CPL and developed by Martin Richards at Cambridge University in 1967. BCPL was too less powerful and too specific and hence it failed.
- iv. The father of C language was the B language developed by Ken Thompson of Bell Laboratories in 1970. It was designed for an early implementation of UNIX. However, it was machine dependent and a 'type-less language'. For this reason, Dennis Ritchie began work on a new language as a successor to B.
- v. The 'C' programming language by Dennis Ritchie came into existence in 1972 at Bell Laboratories. The early development and use of C was closely linked with UNIX for which it was developed. For many years, the only reference

available on C was the published informal description in Kernighan and Ritchie's book.

In 1983, the American National Standards Institute (ANSI) established a committee to provide a formal comprehensive definition of 'C'. This ANSI standard known as "ANSI C" was completed in 1988.

Development of 'C'

Summary



1.1.2 Computer Languages

Computer languages have evolved over the years from the earliest machine language to the recent natural languages.

Low Level Languages

These languages were the earliest languages developed. Under this category, we have Machine and Assembly languages.

Features of Low Level Languages

1. These languages are greatly hardware dependent i.e. the code had to be written for specific hardware.
2. Programs written on one machine will not run on another (Non-portable).
3. Programmers are required to have knowledge about the hardware as well.

• **Machine Language**

Since the computer is made up of electronic circuits, they can only understand binary logic (0's and 1's). Hence in order to communicate with the computer, the user

has to give instructions in term of 0's and 1's. This was called **machine language** and it was one of the earliest computer languages (1940's).

Advantage

1. Since the computer circuits can directly interpret 0 and 1, execution of programs is very fast.

Disadvantages

1. Writing programs in binary is very difficult.
2. It is very easy to make errors during writing or data entry.
3. Debugging is very difficult.
4. There is no distinction between the instruction and operands or data.
5. It is difficult to understand the program logic by looking at the program.

• Symbolic / Assembly Language

These were developed in the 1950's to remove the disadvantage of Machine Language. In these languages, small English like words, called **mnemonics** were used for instructions (*For example*: ADD, SUB, etc) and hexadecimal codes were used for data.

Example: 8085, 8086 languages.

Advantages

1. Writing of programs became easier.
2. Errors are minimized
3. Identification of errors is easy.
4. There is a distinction between instructions and data.
5. Programs can be easily understood.

Disadvantages

1. Because a computer does not understand symbolic language, it has to be translated to machine language.
2. A special software called **Assembler** is needed to translate assembly code to machine code.
3. Execution becomes slower.

High Level Languages

High-Level languages were developed to

1. Improve programming efficiency.
2. Shift focus from the computer to problem solving.

3. Develop portable applications.

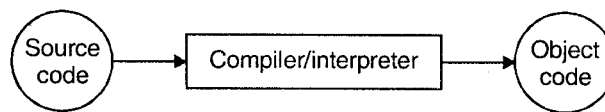
Features of High-Level languages

1. Use of English-like words for instructions.
2. Support to multiple data-types like characters, integers, real-numbers etc.
3. Hardware independent instruction set. (Portability)
4. Programs have to be converted from high-level languages to machine – languages.
5. Conversion is done by special Software (Compiler or Interpreter).

Example: Pascal, FORTRAN, COBOL, BASIC, etc.

Compilers and Interpreters

Programs written in a high level language have to be converted into machine code in order to be executed. The software which does this translation is called a Compiler or Interpreter. Some high level languages use a compiler whereas some use an interpreter.



Difference between Compiler and Interpreter

| No. | Compiler | Interpreter |
|-----|--|---|
| 1. | A compiler takes the entire program and generates the object code for the program. | An interpreter takes a single instruction of the program, converts it to object code and executes it. |
| 2. | An intermediate object code file is created | No intermediate file is created. |
| 3. | Once the object code is created, the program need not be compiled everytime before execution | Every time a program is executed, conversion from high level to machine code has to be performed. |
| 4. | A compiled program executes faster especially if the program contains loops | An interpreter is slower than a compiler. |
| 5. | The compiler is not involved in the execution of the program. | An interpreter also executes the instruction. |
| 6. | There is more memory requirement since object files are created. | Memory requirements are less. |
| 7. | A list of errors is generated after the entire program is checked. | Errors are displayed for every instruction interpreted. ∴ Debugging is easier. |
| 8. | PASCAL, C use compilers | BASIC has an interpreter. |

1.1.3 Where C stands

The 'C' programming language is a very powerful and flexible language.

It provides the programmer a facility to write low-level programs as well as high-level programs.

Thus, it is designed to have both-good programming efficiency and good machine efficiency.

For these reasons, C is called a **Middle Level Language**. It permits machine independent programs to be written as well as permits close interaction with the hardware.

1.2 APPLICATION AREAS

'C' is a general purpose programming language and not designed for specific application areas like COBOL (business applications) or FORTRAN (scientific and engineering applications).

'C' is well suited for business as well as scientific applications because it has various features (rich set of operators, control structures, bit manipulation, etc.) required for these applications.

However it is better suited and widely used for system software like operating systems, compilers, interpreters, etc.

1.3 FEATURES OF 'C'

In the current scenario there are several languages to choose from. Most are well suited for a variety of tasks. However, there are several reasons why 'C' is a popular programming language.

1. **Flexibility:** 'C' is a general-purpose language. It can be used for diverse applications. The language itself places no constraints on the programmer.
2. **Powerful:** It provides a variety of data types, control-flow instructions for structured programs and other built-in features.
3. **Small Size:** 'C' language provides no input/output facilities or file access. These mechanisms are provided by functions. This helps in keeping the language small. 'C' has only 32 keywords, which can be described in a small space and learned quickly.
4. **Modular Design:** The 'C' code has to be written in functions, which can be linked with or called in other programs or applications. C also allows user defined functions to be stored in library files and linked to other programs.
5. **Portability:** A 'C' program written for one computer system can be compiled and run on another with little or no modification. The use of compiler directives

to the preprocessor makes it possible to write a single program that can be used on different types of computers.

6. **High level structured language features:** This allows the programmer to concentrate on the logic flow of the code rather than worry about the hardware instructions.
7. **Low-level features:** 'C' has a close relationship with the assembly language making it easier to write assembly language code in a 'C' program.
8. **Bit Engineering:** 'C' provides bit manipulation operators, which are a great advantage over other languages.
9. **Use of Pointers:** This provides for machine independent address arithmetic.
10. **Efficiency:** A program written in 'C' has development efficiency as well as machine efficiency (i.e. faster to execute).

The 'C' language, however, does have its *limitations*

1. It is not suitable for programming of numerical algorithms since it does not provide suitable data structures.
2. 'C' does not perform bound checking on arrays. This results in unpredictable errors, which are difficult to locate.
3. The order of evaluation of function arguments is not specified by the language.
Example: In the function call, `f(i, ++i);` it is not defined whether the evaluation is left to right or right to left.
4. The order in which operators are evaluated is not specified in some cases.
Example: In `a[i] = b[i + ++i];`, the value of 'i' could be incremented after the assignment or it could be incremented after `b[i]` is fetched but before assignment. The order of evaluation of operands of an operator is also not specified. *Example:* `Sum = (++a, --a);` Here it is left to the compiler as to which it evaluates first.
5. 'C' is not a strongly typed language, which means that the compiler does not strictly check and indicate errors for those statements that attempt a mismatch of data types. This can cause unintentional errors, which are difficult to trace.

1.4

PROGRAM DEVELOPMENT CYCLE

The program development cycle is completed in four steps.

1. Creating the 'C' source code.
2. Compiling the source code.
3. Linking the compiled code.
4. Running the executable file.

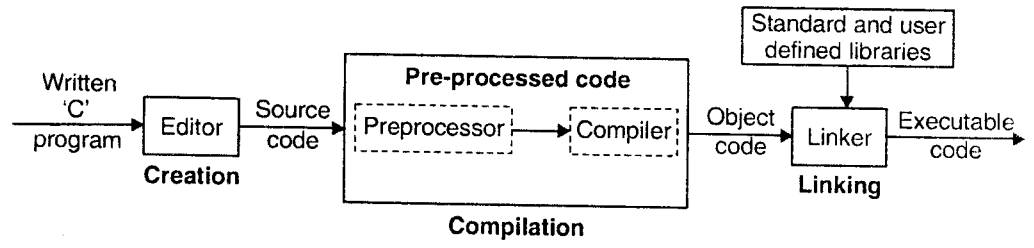


Figure 1.1

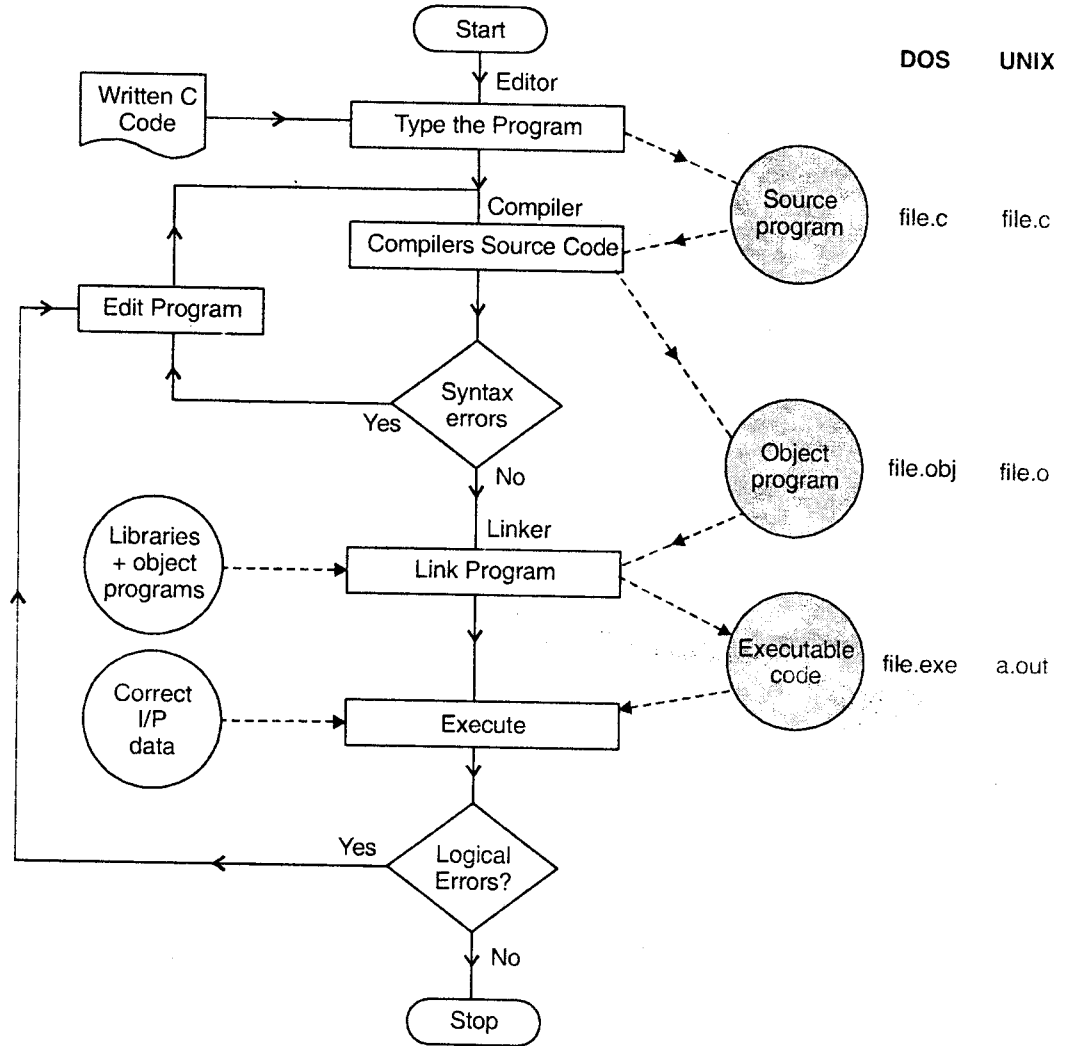


Figure 1.2

Creating the source code

Any editor or word processor can be used to create the source code. The file containing the source code has to be a 'text' file with an extension .C most compilers come with a built in editor. On UNIX, the editors like vi, emacs, etc. can be used.

Compiling the source code

The pre-processing is the first step in the compilation. The source code is given to the pre-processor (Pre-processor is a system program that modifies a C program prior to its compilation) which checks for special instructions (preprocessor directives) in C program (line beginning with # provides an instruction to the preprocessor) and performs other tasks to give the pre-processed code. The compiler then converts this code to binary code (object code). On UNIX systems, the object code has an extension .O and on others it is .obj.

Several compilers have been developed for C. Some of the commonly used ones are: Microsoft C, Borland C, Turbo C, GNU C. Programs can also be compiled on UNIX by the CC compiler.

Linking the object code to create an executable code

The object code of the program has to be linked with the object code of precompiled routines from libraries. The linker creates a file with .exe extension.

Executing the program

Once the executable file is created, you can run it by typing its name at the DOS command prompt or through the option provided by the compiler software. If the desired results are not achieved, changes may have to be made to the source code. When the source code is changed, it has to be recompiled and linked to create the correct executable code.

1.5 STRUCTURE OF A 'C' PROGRAM

The basic building blocks of every C program are **Functions**.

A function is nothing but a module or a subprogram, which performs some task. It may accept some information and may return a single output.

The function main

Every C program consists of one or more functions one of which is the function called **main**. Program execution begins from this function and ends when the instructions in the main function have been executed.

The basic structure of a 'C' program is as shown below:

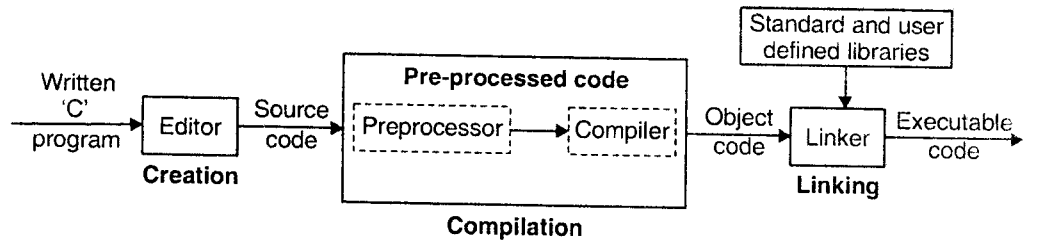


Figure 1. 1

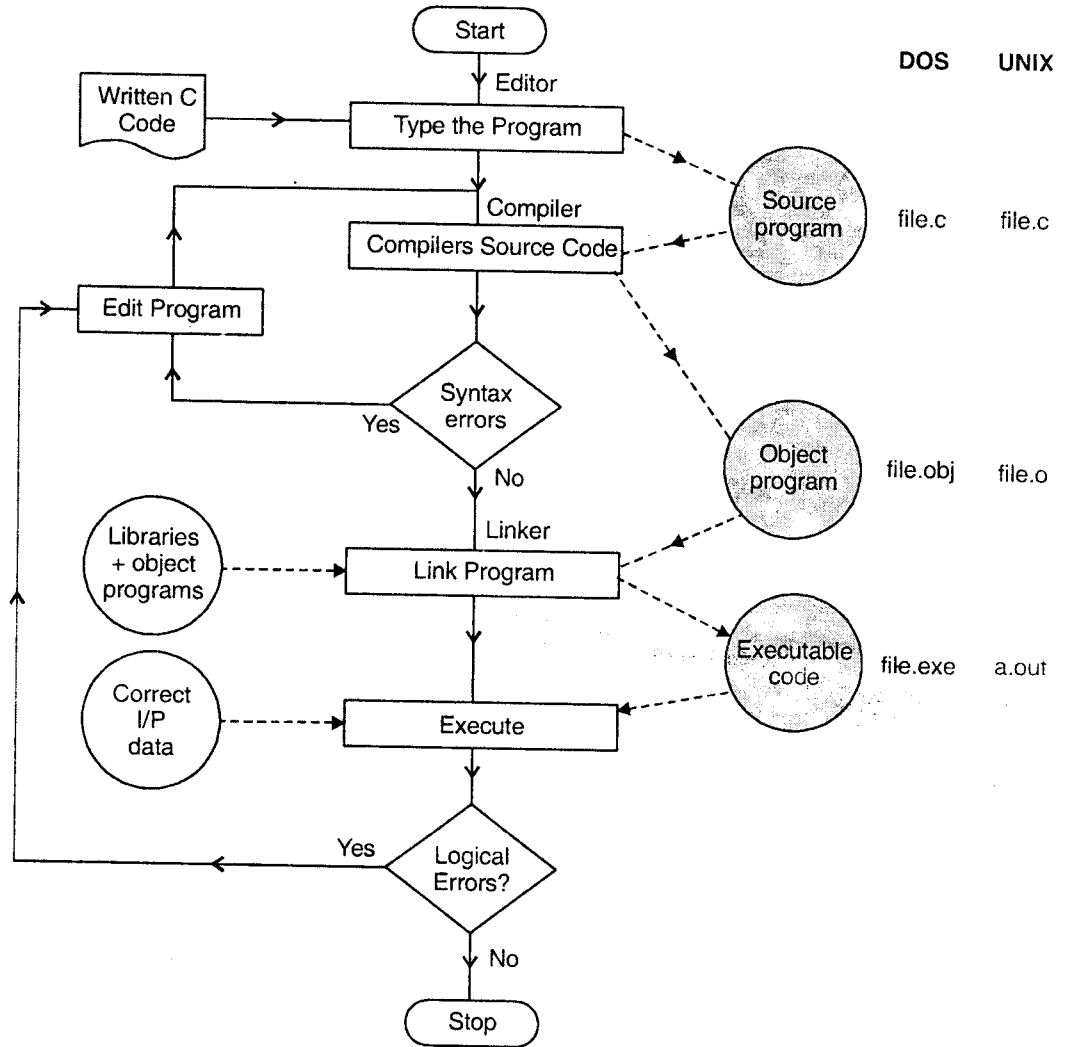


Figure 1.2

Creating the source code

Any editor or word processor can be used to create the source code. The file containing the source code has to be a 'text' file with an extension .C most compilers come with a built in editor. On UNIX, the editors like vi, emacs, etc. can be used.

Compiling the source code

The pre-processing is the first step in the compilation. The source code is given to the pre-processor (Pre-processor is a system program that modifies a C program prior to its compilation) which checks for special instructions (preprocessor directives) in C program (line beginning with # provides an instruction to the preprocessor) and performs other tasks to give the pre-processed code. The compiler then converts this code to binary code (object code). On UNIX systems, the object code has an extension .O and on others it is .obj.

Several compilers have been developed for C. Some of the commonly used ones are: Microsoft C, Borland C, Turbo C, GNU C. Programs can also be compiled on UNIX by the CC compiler.

Linking the object code to create an executable code

The object code of the program has to be linked with the object code of precompiled routines from libraries. The linker creates a file with .exe extension.

Executing the program

Once the executable file is created, you can run it by typing its name at the DOS command prompt or through the option provided by the compiler software. If the desired results are not achieved, changes may have to be made to the source code. When the source code is changed, it has to be recompiled and linked to create the correct executable code.

1.5 STRUCTURE OF A 'C' PROGRAM

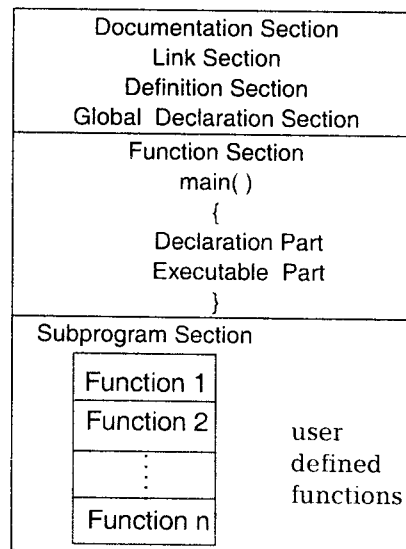
The basic building blocks of every C program are **Functions**.

A function is nothing but a module or a subprogram, which performs some task. It may accept some information and may return a single output.

The function main

Every C program consists of one or more functions one of which is the function called **main**. Program execution begins from this function and ends when the instructions in the main function have been executed.

The basic structure of a 'C' program is as shown below:



- The documentation section consists of comment lines (enclosed in /* and */), which are used to convey program information and other details.
- Note:* Comments can be put anywhere within the program.
- The link section gives instructions to the compiler to link library files and other user files.
 - The definition section defines all symbolic constants.
 - Some variable need to be used in all functions. Such variables are declared in the global declaration section.
 - Every C program must have one main() function. It consists of local declaration (information used only within main) and "C" statements. All statements end with a semicolon.
 - The sub-program section contains all user-defined functions that are called in the main function. The subprogram section may also appear before main() although it is normally placed immediately after main().

1.5.1 Sample 'C' Program

To display the following message C on the screen

Hello!

Welcome to C



Program

```
1.  /* My First C Program */
2.
3.  #include<stdio.h>
4.  main( )
5.  {
6.      printf("Hello!\nWelcome to C");
7.  }
```

Output

```
Hello!
Welcome to C
```



Explanation

1. Line 1 is a 'C' comment. A comment is used to give additional information about the program. It has to be enclosed in /* and */. Comments are ignored by compiler.

Comments can be written anywhere in the program and are used for documentation. They cannot be written inside one another (nesting).

Example: /* First comment /* Second Comment */ /* is invalid.

2. Line 3 is the link section and it tells the compiler to include information about the specified file, i.e. Standard Input – output functions. The #include directive gives the program access to a library. A library is a collection of useful functions and symbols that may be accessed by a program. The ANSI (American National Standards Institute) standard for C requires that certain standard libraries be provided for every ANSI C implementation. A C system may expand the number of operations available by supplying additional libraries; an individual programmer can also create libraries of functions. Each library has a standard header file whose name ends with the symbols.

The #include directive causes the preprocessor to insert definitions from a standard header file into a program before compilation.

The directive #include <stdio.h> /* printf, scanf definitions */ notifies the preprocessor that some names used in the program (such as printf scanf) are found in the standard header file <stdio.h>.

3. Line 4 is the beginning of the main() function. It is the only compulsory and the most important function of any C program.
4. Line 5 and 7 are the opening and closing braces of main. These braces contain the instructions to be executed (statements).
5. Line 6 is the only statement in the function. It is a call to another function called printf, which is an output function. Its job is to display the provided information on the screen. The definition of this function is in the standard input output library stdio.h. Hence we have included that file in the program.
6. The sequence of characters enclosed in " " is called a string which is displayed on the screen as it is.
7. \n is a special character (although it is composed of two characters) called the **newline** character. This character advances the output to the next line.

printf does not supply a new line automatically. Hence multiple printf () statements are used. So, the following printf statements

```
printf("Welcome");  
printf("to");  
printf("C");
```

Will give the following output

```
Welcome to  
C
```

We can introduce the new-line character in the string at the appropriate position. The printf statements will now look like.

```
printf(" Welcome to \n C");  
This is analogous to writing  
printf("Welcome to \n");  
printf("C");
```

Exercise

1. How can a comment be written in a 'C' program?
2. Can the user defined functions be written above main()?
3. What is the purpose of the link section?
4. What will be printed by the following segments of 'C' code?
 - i. `printf (" Hello \n\n\n everyone");`
 - ii. `printf ("\n");`
5. 'C' is middle level language. Comment.
6. What are the advantages of 'C'?



2.1 'C' CHARACTER SET

The C character set consists of upper and lowercase alphabets, digits, special characters and white spaces. The alphabets and digits are together called the alphanumeric characters.

1. **Alphabets**

A B C Z
a b c z

2. **Digits**

0 1 2 3 4 5 6 7 8 9

3. **Special characters**

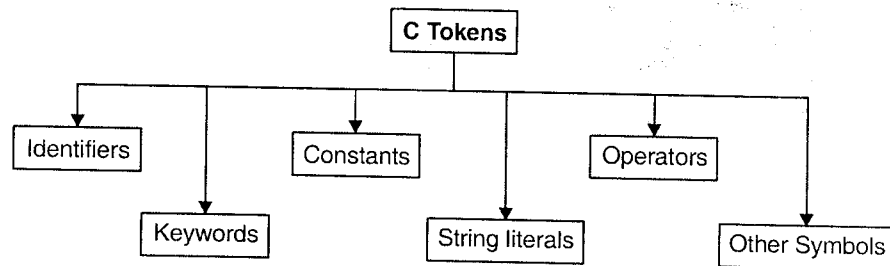
, . ; : # ' " ! | ~ < > { } () - _ \$ % & ^ * + [] / \

4. **White space characters**

blank space, new-line (\n), carriage return (\r), form feed (\f), horizontal tab (\t), vertical tab (\v).

2.2 C TOKENS

The smallest individual units in a C program are called tokens as shown below.



We shall be studying each of these in the sections to come.

2.3**IDENTIFIERS AND KEYWORDS**

Every C word is classified either as an identifier or a keyword.

Identifier

An identifier is a user-defined name given to a program element-variable, function, and symbolic constants.

There are certain rules, which should be followed while naming an identifier. They are:

- i. Identifier names must be a sequence of alphabets and digits and must begin with an alphabet or an underscore (_)
- ii. No special symbols, except an underscore(_) are allowed. An underscore is treated as a letter.
- iii. Reserved words (keywords) should not be used as an identifier.
- iv. C is case sensitive i.e C treats uppercase and lowercase letters differently. It is a general practice to use lower (or mixed) case for variables and function names and uppercase for symbolic constants.
- v. For any internal identifier name (an identifier declared in the same file) at least the first 31 characters are significant in any ANSI C compiler.

Examples of valid identifiers

Rate_of_interest add_matrix Sum PI
Month_of_Year a123

Keywords

Keywords are reserved words and are predefined by the language. They cannot be used by the programmer in any way other than that specified by the syntax. ANSI C language has only 32 keywords. They are:

ANSI C Standard Keywords

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

The following are additional keywords in Turbo C.

| | | | |
|-----|-------|-----------|--------|
| asm | _es | Far | near |
| _cs | _ss | Huge | pascal |
| _ds | cdecl | interrupt | |

2.4 CONSTANTS

Constants refer to fixed values that do not change during program execution. They can be classified as:

- i. Integer constants
- ii. Floating Point Constants
- iii. Characters constants
- iv. String literals
- v. Enumeration constants.

2.4.1 Integer Constants

An integer constant refers to whole numbers. It can be specified in three ways:

- a. Ordinary **Decimal** number (base 10)
- b. **Octal** numbers (base 8)
- c. **Hexadecimal** numbers (base 16)

An integer constant has to follow the following rules.

- i. It contains a sequence of digits from 0 to 9. (Octal contains digits from 0 to 7; Hexadecimal constant contains digits from 0 to 9 and letters A–F)
- ii. An octal constant is preceded with '0' and hexadecimal constant with 0X or 0x.
- iii. No commas, spaces or other symbols are allowed in between.
- iv. The integer can be either positive or negative. It may or may not be prefixed by a – sign.
- v. A size or sign qualifier can be appended at the end of the constant.
 - U or u for unsigned.
 - S or s for short
 - L or l for long.

Examples:

| | |
|--------|---------------------------------------|
| 123 | 56789U (unsigned integer) |
| -31000 | 7689909L (long integer) |
| 0170 | 0X34ADL (long hexadecimal) |
| 0x 2A | 6578890994UL (unsigned long integer) |
| -100 s | 120US (unsigned short) |

Note: The ANSI C standard supports a + sign before the positive integer corresponding to the – for a negative integer although it is rarely used.

2.4.2 Floating Point Constants

These are real numbers having a decimal point or an exponential or both. The rules governing the floating point representation are :

- i. They have a decimal point and digits from 0 to 9.
- ii. No embedded spaces, commas and other symbols are allowed.
- iii. They may or may not be prefixed by a – sign.
- iv. It is possible to omit digits before or after the decimal point.

Examples: 0.246 975.64 –.54 +5.

Exponential notation

This is used to represent real numbers whose magnitude is very large or very small.

The format is:

| |
|--|
| mantissa e exponent Or mantissa E exponent |
|--|

- i. The **mantissa** can be a floating point number or an integer.
- ii. It can be positive or negative.
- iii. The **exponent** has to be an integer with optional plus or minus sign.

Example: The number 231.78 can be written as 0.23178e3 representing 0.23178×10^3 .

75000000000 can be written as 75e9 or 0.75e11. 0.0000045 can be written as $0.45e-5$.

2.4.3 Character Constant

A character constant is any single character from the C character set enclosed within single quotes. *Example:* 'a' '#' '2'

The value of the character constant is the numeric value of the character.

Example: the character constant '0' has ASCII value 48, which is unrelated to numeric digit 0.

Escape Sequences

C supports some special character constants used in output functions. They are also called backslash character constants because they contain a backslash and a character.

Although they look like two characters, they represent only one.

Complete set of escape sequence is:

| Character | Meaning |
|-----------|--|
| \a | alert (bell) |
| \b | backspace |
| \f | form feed |
| \n | newline |
| \r | carriage return |
| \t | horizontal tab |
| \v | vertical tab |
| \0 | null character |
| \\ | backslash |
| \? | question mark |
| \' | single quote |
| \" | double quote |
| \O | octal number |
| \xN | hexadecimal constant (where N is hexadecimal constant) |
| \N | octal constant (where N is an octal constant) |

2.4.4 String Literals

A string constant or string literal is a sequence of zero or more characters enclosed in double quotes.

Example: "Welcome to C"

"First Line \n Second Line"

The double quotes are not a part of the string but only act as **delimiters**. If the backslash or double quote is required to be a part of the string, they must be preceded by a backslash (\).

Example:

```
printf("He said \"Hello \" "); will display
```

```
He said "Hello"
```

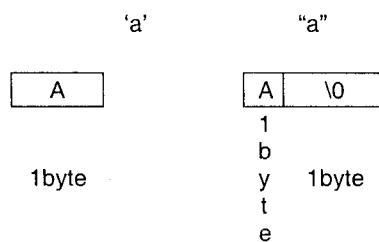
```
printf ("\\ is a backslash" ); displays
```

```
 \ is a backslash
```

Technically, the internal representation of a string has a null character ('\0') at the end. Therefore the physical storage required is one more than the number of characters in the string.

Difference between 'a' and "a".

'a' is a character constant and stored as the numeric value of a. "a" is a string literal and consists of the characters, a and '\0'.



2.4.5 Enumeration Constant

An enumeration is a list of constant values-- each can be represented by an integer.

It is a user defined data type with values ranging over a finite set of identifiers called enumeration constants.

Example: enum color {red, blue, green};

Red, blue and green are constants, which represent the integer values of 0,1 and 2 respectively.

Values can be explicitly specified for the identifiers.

```
enum color {red = 10, blue, green = 30 };
```

Here, blue is assigned number 11. If no value is specified for green it will assume the value 12.

Enumerations provide a convenient way to associate constant values with names. It also makes the program easy to read and understand.

2.5 VARIABLES

A **variable** name is an identifier or symbolic name assigned to the memory location where data is stored. In other words, it is the data name that refers to the stored value. A variable can have only one value assigned to it at any given time during program execution. Its value may change during the execution of the program. Rules regarding naming variables:

- i. Since the variable name is an identifier, the same rules apply.
- ii. Meaningful names should be given so as to reflect value it is representing.

| | |
|--------------|-------------|
| student name | rank 1 |
| basic_sal | amount |
| roll_num | No_of_years |

2.5.1 Data types in C

Programs work by processing data. A programming language must give you a way of storing the data. Associated with the data is its type.

When a variable is used, you have to specify what type of data it can contain.

The C programming language supports the following data types:

```
int    float    double  char    void
```

They are called basic or fundamentals data types. In addition, C also supports the enumerated data type specified by the keyword enum.

Fundamental data types

| Data types | Description | Size (in bytes) | Range |
|------------|--|-----------------|-----------------------|
| char | A single character | 1 | -128 to 127 |
| int | an integer number | 2 | -32768 to 32767 |
| float | A single precision floating point number (6 precision digits) | 4 | 3.4 e-38 to 3.4 e +38 |
| double | A double precision floating point number (10 precision digits) | 8 | 1.7e-308 to 1.7e +308 |
| void | empty data type | 0 | valueless |

The size allocated for an integer depends upon the compiler. The size of a data-type can be obtained by using the `sizeof()` operator which gives the size of the specified data type in bytes.

Usage:

```
sizeof(data_type)
```

Example:

```
printf ("%d", sizeof(char));
```

Qualifiers

A qualifier, when applied to a data type alters its size or sign.

The size qualifiers are

- short
- long

The sign qualifiers are

- signed
- unsigned

Normally, short and long cannot be applied to char and float and signed and unsigned cannot be applied to float, double and long double.

ANSI C has the following rules:

short int < = int < = long int

float < = double < = long double

The data types, sizes and their ranges are as shown in the following table.

All possible Data types in C (Basic and Qualified)

| Type | Size (in bytes) | Range |
|--------------------|-----------------|------------------|
| char | 1 | -128 to 127 |
| unsigned char | 1 | 0 to 255 |
| signed char | 1 | -128 to 127 |
| unsigned int | 2 | 0 to 65535 |
| signed int | 2 | -32768 to +32767 |
| unsigned short int | 2 | 0 to 65535 |

| | | |
|-------------------|----|---------------------------|
| short signed int | 2 | -32768 to 32767 |
| long int | 4 | -2147483648 to 2147483647 |
| long unsigned int | 4 | 0 to 4294967295 |
| long signed int | 4 | -2147483647 to 2147483648 |
| float | 4 | 3.4e-38 to 3.4 e+ 38 |
| double | 8 | 1.7e - 308 to 1.7 e+ 308 |
| long double | 10 | -1.7e4932 to +1.7e4932 |

Note: The exact size allocated and the ranges for these data types can be obtained from constants defined in header files <limits.h>, <float.h> and <values.h>.

Enumerated Data type

A user defined data type along with its set of identifiers can be created by the following declaration.

```
enum data_type_name {constt1 , constt2, .....};
```

Example:

```
enum daysofweek { Sun, Mon, Tue, Wed, Thu, Fri, Sat};
```

void data type

void is an empty data type defined by the keyword **void**. It is used with functions.

When used as a function return type, it means that the function does not return anything.

Example: void calculate_and_display (int a).

When used in place of the parameter list, it indicates that the function does not accept any information.

Example: int random_number (void).

We shall be dealing more with void data type in the book.

Creating new data-types names

C provides a facility called **typedef** for creating new data type names.

The syntax of typedef is

```
typedef data_type synonym
```

For *example*, the statement,

```
typedef unsigned long ulong;
```

declares ulong as a new data type equivalent to unsigned long. It can be used in exactly the same way as the type unsigned long can be.

Example

```
typedef int length;
```

makes the name 'length' a synonym for int.

- It is important to understand that a typedef statement does not create a new type in any sense; it merely adds a new name for some existing type.
- Use of typedef enhances program readability.

2.6 DATA DECLARATIONS AND DEFINITIONS

Programs operate on data. The data items, which a program manipulates, can be divided into two classes:

1. Constants
2. Variables

While variables take different values at different points in time as the program executes, constants have fixed values. These must be declared before they are used.

1. Declaring variables

All variables used in the program must be declared at the beginning.

A variable can be used to store data of any data type irrespective of what the variable name is. A variable is declared by the following syntax.

```
Storage class Data-type var1,  
var2, ..... , varn
```

where var1 to varn are variable names separated by commas. We shall study about storage classes in later.

Example:

```
int marks, age;  
float amount;
```

Declaration does two things:

- i. It informs the compiler the name of the variable.
- ii. It specifies what type of data the variable will hold.

There are three basic places where variables will be declared

- i. Inside functions – local variables
- ii. In the definition of function parameters – formal parameters.
- iii. Outside all functions – global variables.

Local variables: These variables are also called automatic variables (keyword 'auto' may be used to declare them). They can be used only within the block where they are declared. A local variable is created upon entry into the block and destroyed upon exit.

Example : Consider two functions as shown

```
func1( )  
{ int x;  
  x = 20;  
}  
func2( )  
{ int x;  
  x = 100;  
}
```

Here, x has been declared twice but the variable x in func1() is not related to the variable x in func2(). Both are independent and exist only within their respective functions.

Formal parameters

If a function is to accept data, it must use arguments and declare them to accept values. They behave like any other local variable inside the function.

Example

```
sum(int a, int b)
{
    ≡≡≡ } function body
}
```

Here, sum is a function which accepts two integer values in variables a and b. It could also be written as follows:

```
sum(a,b)
int a;
int b;
{
    function body
}
```

We shall be studying formal parameters in detail in the Chapter 'Functions'.

Global Variables

Unlike local variables, global variables exist and can be used anywhere in a program. They may be accessed by any expression regardless of what function the expression is in.

They are created by declaring them outside any function.

Example

```
int count; /* count is global */
main( )
{ count = 200;
  func1( );
}
func1( )
{ count = 300 ;
}
```

Initializing Variables

Assigning values to variables during declaration is called initialization.

Example

```
int i = 5 ;
```

This statement not only declares the variable i but also assign the value 5 to this variable.

Multiple variables can also be initialized.

Example

```
int sum = 0, i = 10 ;
```

2. Defining Constants

A constant can be declared in C by two methods

- Using const qualifier

- Using the `#define` preprocessor directive.

`const` is a qualifier that can be applied to a data item of any data type. The contents of this data item cannot be changed during program execution—only assigned at the time of declaration (initialized).

Syntax

```
cons data_type constant name = value;
```

Example

```
const float pi = 3.142 ;
const char quit = 'q';
```

Another method of defining constants is by using a pre-processor directive — `#define`.

(Pre-Processor directives are covered later in this book)

The `#define` directive works as follows

```
#define CONSTNAME literal
```

This creates a constant named `CONSTNAME`, which represents the constant value of the literal. By convention, the constant name is written in uppercase.

Example

```
#define PI 3.142
#define TRUE 1
```

Any occurrence of `PI` in the program is replaced by the literal 3.142.

Exercise

A. Programming exercises

1. Which of the following are invalid identifier names? Why ?

| | |
|-------------------|---------------------|
| rate_of_interest | Basic salary |
| PI | "name" |
| 2nd_month | Float |
| Compound_interest | Address_of_employee |
| 124.56 | x + y |

2. Which of the following are invalid constants of the specified category ? Why?

a) Integer

| | | |
|--------|-------|---------|
| 25,000 | 40565 | OxAB |
| -75.0 | '123' | -327000 |

b) Character

| | | | |
|-----|------|------|------|
| "a" | '25' | 'x' | '\b' |
| '1' | abcd | '#a' | '\' |

c) String literals

```
"He said, "Hello""
" abc # $ -----\n"
" 123.25"
" ((left corner \ r Back to the left \n\n"
```

d) float

```
16.3 e-18  -17.e.3  914.533
2.5 × 16.6  +1.7e-3  25.4e+4
```

3. Write equivalent C expressions for the following equations.

$$\frac{a+b}{c+d} - \frac{a-b}{c-d} \quad , \quad \left[\frac{3x^2y}{x+y} - \frac{x}{(x+y)(x-y)} \right]$$

$$S = ut + \frac{1}{2}at^2. \quad , \quad f = \frac{9c}{5} + 32$$

B. Review Questions

1. Explain the four basic data types in C.
2. Explain the types of constants in C.
3. What are variables? State the rules for naming a variable.
4. What is an escape sequence?
5. What are the two methods for declaring constants?



3.1 OPERATORS AND EXPRESSIONS

An **operator** is a symbol that represents an operation. It instructs the compiler to perform some action on one or more operands.

Example

The Symbol + represents addition.

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

Examples of expressions

a + b

P I * r * r

(x + y) - z.

An operator can be **unary**, **binary** or **ternary** depending on whether it operates on one, two, or three operands respectively.

Operators can be classified according to the nature of operation they perform. The different categories are:

- Arithmetic operators
- Relational operators
- Logical operators
- Assignment operator
- Increment and Decrement operators
- Conditional Operator
- Bitwise operators
- Other operators,

Operator Precedence Hierarchy and Associativity

If an expression contains more than one operator, the important question is what is the order of evaluation? Some rules are needed to specify the order in which operations are performed. These rules are called Operator Precedence or Hierarchy rules.

Precedence states the relative importance or priority of operators with respect to other operators.

Another possibility is that an expression may contain more than one operator having the same priority. Here, the associativity specifies the order of evaluation of operators having the same precedence or at the same hierarchy level.

3.1.1 Arithmetic Operators

These perform arithmetic operations. C provides five arithmetic operators.

| Operator | Meaning | Remark |
|----------|-----------------|---------------------------------------|
| + | Addition | Can also be used as unary plus. |
| - | Subtraction | Also used as unary minus |
| * | Multiplication | |
| / | Division | |
| % | Modulo Division | Can be used only on integer data type |

Note: C has no operator for exponentiation. (The function `pow(x,y)` in `math.h` can be used to calculate x^y).

- The unary minus operator has the effect of multiplying the operand by -1 .
- The unary plus, which was added later, gives the value of the operand.
- Arithmetic operations performed on integers (integer arithmetic) yields an integer values.

Example

$$16 + 5 = 21$$

$$16 - 5 = 11$$

$$16 * 5 = 80$$

$$16 / 5 = 3$$

$$5 / 2 = 2$$

$$16 \% 5 = 1$$

$$-16 \% 5 = -1 \text{ (remainder after division and the sign is of the first operand)}$$

- Arithmetic operations performed on float operands (float arithmetic) yield a float result, which is rounded off to the number of significant digits permissible.

$$\begin{aligned} \text{Example } 5.0 + 2.0 &= 7.0 \\ 5.0 / 2.0 &= 2.5 \\ -2.0 / 3.0 &= -0.666667 \end{aligned}$$

- when the operands are of different data types (mixed mode arithmetic), the result is promoted to the 'higher' data type. (char < int < float). Thus if one operand is an integer and the other float the result will be of float type.

$$\text{Example } 5.0 / 2 = 2.5$$

Hierarchy of Arithmetic Operators

| Operators | Associativity |
|-----------|---------------|
| * / % | L → R |
| + - | L → R |

Example

Consider the integer expression

$$5/2 + 4 - 6 * 2 + 25 / 5 - 3 / 4$$

The order of evaluation is as shown:

$$\underline{5/2} + 4 - 6 * 2 + 25 / 5 - 3 / 4$$

$$2 + 4 - \underline{6 * 2} + 25 / 5 - 3 / 4$$

$$2 + 4 - 12 + \underline{25 / 5} - 3 / 4$$

$$2 + 4 - 12 + 5 - \underline{3 / 4}$$

$$\underline{2 + 4} - 12 + 5 - 0$$

$$\underline{6 - 12} + 5 - 0$$

$$- \underline{6 + 5} - 0$$

$$- \underline{1} - 0$$

$$- 1$$

Note: In order to override the operator precedence rules, parentheses can be used since parentheses have higher priority over operators.

Example: In the expression $(4+5) * 6$,

The addition will be done first even though * has higher precedence since the addition operation is parenthesized.

3.1.2 Relational Operators

Relational operators are used to compare expressions. An expression containing a relational operator evaluates to either True (1) or False (0).

Any non-zero value is considered 'True' in C and 0 is false. Thus, even negative values are True!

The six relational operators are

| Operator | Meaning |
|----------|---------------------------|
| < | Less than |
| <= | Less than or equal to |
| > | Greater than |
| >= | Greater than or equal to |
| == | Equal to (equality) |
| != | Not equal to (inequality) |

These operators are mainly used in decision-making statements to decide the course of action in a program. These operators are lower in precedence than arithmetic operators. Among themselves, the precedence is

| Operators | Associativity |
|-----------|---------------|
| < <= > >= | L → R |
| == != | L → R |

Examples

| | |
|----------------|---|
| 25 < 30 | True |
| 2.5 <= 2.5 | True |
| 'a' == 97 | True |
| 'b' < 'a' | False |
| (a+b) != (x+y) | True if the sum of values of a and b is not equal to the sum of values of x and y |

3.1.3 Logical Operators

Sometimes, we need to test more than one condition at a time and make a decision depending upon the result.

The logical operators are used to combine two or more expressions (usually relational). The entire expression is called logical expression which evaluates to True(1) or False (0). The three logical operators in C are:

| Operator | Meaning | Remarks |
|----------|-------------|--------------------|
| & & | Logical AND | } Binary operators |
| | Logical OR | |
| ! | Logical NOT | Unary operators |

Evaluation of a logical expression stops as soon as a true or false result is known.

The results of logical AND (&&) and OR (||) operators for different combinations of the two operands is given in the following truth table.

| Op1 | Op2 | Op1 && Op2 | Op1 Op2 |
|-------|-------|------------|------------|
| False | False | 0 | 0 |
| False | True | 0 | 1 |
| True | False | 0 | 1 |
| True | True | 1 | 1 |

Examples

(marks >= 60) && (marks < 70)
age > 60 || salary > 10000

The logical NOT (!) operator takes a single expression and reverses the value of the expression i.e. if the expression is True, the ! operator evaluates to false and vice-versa.

Example

!(5 < 10) evaluate to 0 since 5 < 10 is True.

Precedence and Associativity of logical operators.

| Operators | Associativity |
|-----------|---------------|
| ! | R → L |
| && | L → R |
| | L → R |

Note: ! has higher priority than arithmetic and relational operators, but && and || have lower priority than both.

3.1.4 Increment and Decrement operators

C provides two useful unary operators not generally found in other languages;

They are,

++ Increment
-- Decrement

++ increments the value of the operand by 1 and -- decrements the value of the operand by 1. Both these operators can be used in the **prefix** form (i.e. before the operand) or the **postfix** form (after the operand). The operand can only be a single variable.

When used in the prefix form, the increment or decrement is done before the value of the operand is used. If used in the postfix form, the operand increments or decrements after its value have been used.

Note: When used independently, the prefix and postfix forms make no difference but they behave differently when used in expressions on the right hand of an assignment statements.

Example

If n is 5, then the statements ++n; and n++ ; both increment the values of n by 1 and are equivalent to n= n+1 ;

However, in the statement,

y = n++ ; n increments after its value has been assigned to y i.e y is given the value 5 and then n becomes 6. Whereas y = ++ n first increments n to 6 and 6 is then assigned to y.

The same logic applies to the decrement operator.

Example

Consider the expression x++ && ++y || z++ .

If values of x, y and z are 0,1 and 0 respectively, the expression evaluates to 0 and values of x,y and z becomes 1,1 and 1 respectively.

The && operation is performed before ||. For the && the initial value of x i.e. 0 is used. ++y will not be evaluated since the result of the && operation is known to be 0. For the || operation, one operand is 0 and so the other operand is evaluated. The old value of z (i.e 0) is used since it is post- increment. ∴ 0||0 yields 0 and then z increments to 1.

The increment and decrement operators are along with the logical NOT in precedence, i.e the highest level we have seen so far. They have a R → L associativity.

3.1.5 Bitwise Operators

C has a distinction of providing six operators for manipulation of data at bit level. They are applied only to integral operands i.e. char, short int and long whether signed or unsigned.

| Operator | Operation |
|----------|--------------------------|
| & | Bitwise AND |
| | Bitwise OR |
| ^ | Bitwise XOR |
| << | Left shift |
| >> | Right shift |
| ~ | One's complement (Unary) |

Except for ~ the others are binary operators and operate on corresponding bits of the two operands.

The bitwise XOR (exclusive OR) operator sets a one in each bit position where its operands have different bits and zero where they are the same.

Example

Assume that a and b are integers with values 13 and 7 respectively. Assuming that an integer occupies 2 bytes,

```

a in binary   = 0000 0000 0000 1101
b in binary   = 0000 0000 0000 0111
a & b        = 0000 0000 0000 0101
a | b        = 0000 0000 0000 1111
a ^ b        = 0000 0000 0000 1010

```

Shift operators

The bit pattern of the data can be shifted by a specified number of positions to the left or right using the left shift (<<) and right shift (>>) operators respectively. The shift operators perform shift of their left operand.

When the data is shifted left, the trailing empty spaces are filled with zeros.

Similarly, the leading empty spaces are zero filled when data bits are shifted right.

Example:

```

a = 0000 0000 0000 1101
a << 3 = 0000 0000 0000 1000
a >> 3 = 0000 0000 0000 0001

```

zero filled spaces

(The rightmost three bits drop off)

The general syntax is:

| |
|-------------------------------------|
| operand1 shift_operator operand2 |
|-------------------------------------|

Note: Shifting by one position to left is effectively multiplying the operand by two.

Shifting right by one position divides the operand by two.

One's complement operator

The ~ operator yields the one's complement of an integer, that is, it inverts each bit of the operand (1 to 0 and vice versa)

Example

```

If a = 0000 0000 0000 1101
~ a = 1111 1111 1111 0010

```

Precedence

~ is along with other unary operators like ++, -- and ! in hierarchy with R → L associativity. The shift operators have higher precedence as compared to Bitwise AND, OR and XOR.

3.1.6 Assignment operator

The assignment operator = is used to assign the value of an expression to a variable. The syntax is

```
variable = expression
```

An assignment expression followed by a ; becomes an assignment statement.

Example

```
sum = a + 10;
```

The expression a + 10 is evaluated and its value is assigned to variable sum.

```
c = a << 3 ;
x = a*3 + b/5 ;
```

Shorthand assignment operators

These are obtained by combining certain operators with the = operator. They have the format

```
variable operator= expression;
```

C supports the following shorthand assignment operators

```
+ =  - =  / =  % =  << =  >> = & =  | =  ^ =
```

Examples:

```
x += y ; implies x = x + y ;
m /= 3 ; implies m = m/3 ;
a += b + 1 ; implies a = a + ( b + 1 )
```

Precedence

Assignment operators have the lowest priority so far with associativity R → L.

Example: Consider the statement

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

Here, the value of c is assigned first to b which is then assigned to a.

i = j += k ; is also a valid assignment statement which is the same as i = j = j + k ;

3.1.7 Conditional operators

This is the only ternary operator in C. The operator pair ?: is used to construct conditional expression of the form.

```
expression1? expression2 : expression 3
```

←———— Conditional expression —————→

expression 1 is evaluated first. If it is True (nonzero), then expression2 is evaluated and becomes the resulting value of the conditional expression.

If expression is 0 (False), the value of the entire expression is that of expression3.

Example: Let $a = 10$ and $b = 15$,
 $\text{larger} = (a > b) ? a : b ;$

Here larger will be assigned 15 i.e. the value of b.

This is the same as

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

3.1.8 Other operators

Comma Operator

The comma ',' operator is used to separate a set of expressions. A pair of expressions separated by a comma is evaluated left to right and the type and value of the result are the type and value of the right operand.

Example: Consider $i = (j = 3, j + 2) ;$

Here, the right hand side contains two expressions $j = 3$ and $j + 2$ which are evaluated $L \rightarrow R$. Thus 3 is first assigned to j and the value $3 + 2$ is assigned to i .

It could also be used to interchange the values of two variables in a single statement as shown.

```
temp = a, a = b, b = temp ;
```

The comma operator has the lowest precedence and associates from $L \rightarrow R$.

size of Operator

This unary operator gives the size (in bytes) of the data-type or variable. The usage is

sizeof(data type)

OR

sizeof(object)

Example : `sizeof(char)` gives the result as 1.

Example:

```
printf("%d %d", sizeof(int), sizeof(float);
```

typedef operator

C provides a unary operator for explicit type conversion called cast operator. Its usage is

(type_name)expression

The expression is converted to the specified data type locally only for the purpose of evaluation of the expression.

Example: The ratio of number of males to the number of females in a town can be calculated as :

```
ratio = no_of_males / no_of_females ;
```

Since `no_of_males` and `no_of_females` will be declared integers, the division of the two yields an integer. So even if `ratio` is declared as a float, the fractional part is truncated due to integer arithmetic on the right. This can be solved by locally converting one of the operands to a float so that the result of division is a float `ratio = (float)no_of_males / no_of_females;`

Address (&) and Indirection (.) operators

C provides two unary operators for manipulating data using pointers.

The & operator when used with a variable yields its address.

The operator denotes indirection and returns the value of the object located at the address that follow it.

We shall study more about these in later chapters.

Both these operators have a high precedence along with other unary operators.

The • and -> operators

The • (dot) and -> (arrow) operators are used to refer to individual elements of structures and unions (covered in later chapters). Structures and unions are compound data types that can be referenced under a single name.

[] and ()

Parentheses () are used to increase the precedence of operators inside them. Square brackets perform array indexing i.e. given an array, the expression within [] provides an index or subscript to the array.

3.1.9 Precedence and Associativity of operators

The operators are listed in order of decreasing precedence. The Operators grouped together in one level have the same precedence.

| Level | Operator | Description | Associativity |
|-------|----------|---------------------------------------|---------------|
| 1 | () | Function call | L → R |
| | [] | Array element reference | L → R |
| | -> | Pointer to structure member reference | L → R |
| | • | Structure member reference | L → R |
| 2 | - | Unary Minus | R → L |
| | + | Unary plus | R → L |
| | ++ | Increment | R → L |
| | -- | Decrement | R → L |
| | ! | Logical negation | R → L |
| | ~ | One's complement | R → L |
| | * | Pointer reference (indirection) | R → L |
| | & | Address | R → L |
| | sizeof | Size of an object | R → L |
| | (type) | Type cast | R → L |

| | | | |
|----|--|--------------------------|-------|
| 3 | * | Multiplication | L → R |
| | / | Division | L → R |
| | % | Modulo division | L → R |
| 4 | + | Addition | L → R |
| | - | Subtraction | L → R |
| 5 | << | Left shift | L → R |
| | >> | Right shift | L → R |
| 6 | < | Less than | L → R |
| | <= | Less than or equal to | L → R |
| | > | Greater than | L → R |
| | >= | Greater than or equal to | L → R |
| 7 | == | Equality | L → R |
| | != | Inequality | L → R |
| 8 | & | Bitwise AND | L → R |
| 9 | ^ | Bitwise XOR | L → R |
| 10 | | Bitwise OR | L → R |
| 11 | && | Logical AND | L → R |
| 12 | | Logical OR | L → R |
| 13 | ?: | Conditional | L → R |
| 14 | = *= /= %= += -= &= ^= = <<= >>= | Assignment | R → L |
| 15 | , | Comma | L → R |

3.2 STATEMENTS

A C program consists of statements. A statement is composed of expressions and operators and it is a complete instruction instructing the compiler to carry out some task.

Like mentioned earlier C statements must always end with a semicolon (except for preprocessor directives which are discussed later).

Example: `x = a + b ;`
`y = (i + 3) * (j - 5);`

These are *examples* of assignment statements.

Types of Statements

C statements can belong to one of the following categories:

1. **Null statement**
A semicolon on a line is a null statement. It does not perform any action.
2. **Expression statement**
Most expression statements are assignments or function calls.
3. **Compound statement**
Several statements grouped together in { } forms a compound statement or a block. The body of a function is also a compound statement. There can be statements of other types within the braces.
4. **Selection statement**
These statements involve condition checking and choose one of several flows of control. Statements of this type are if statement, if else and switch statement which will be discussed in later chapters.
5. **Iteration statement**
These statements specify looping, where a statement or a block has to be repeatedly executed a specific number of times or as long as the test expression is satisfied.
The while, do – while and for statements belong to this category.
6. **Jump statement**
Jump statements transfer control unconditionally. These are goto, continue, break and return statements.
7. **Labeled statements**
Some statements may carry label prefixes. The label identifier does not need to be declared and can only be used with the goto statement. Other forms of the labeled statements are within the switch statement (case and default).

SOLVED PROGRAMS

1.



/*Find simple interest */

```
#include<stdio.h>
main()
{
    float principal, rate, time, interest;
    clrscr();
    printf("Enter the principal:");
    scanf("%f", &principal);
```

```
printf("\nEnter the rate of interest:");
scanf("%f", &rate);
printf("\nEnter the time in years:");
scanf("%f", &time);
/* echo the data */
printf("\nPrincipal = %2f\n", principal);
printf("\nRate = %2f\n", rate);
printf("\nTime = %2f\n", time);
interest = principal * rate * time/100.0;
printf("\n\nSimple interest is : %2f\n", interest);
getch(); /* freeze the monitor*/
}
```



Output

```
Enter the principal : 1000
Enter the rate of interest : 5
Enter the time in years : 4
Principal = 1000.00
Rate = 5.00
Time = 4.00
Simple interest is 200.00
```

2.



```
/* Compute surface area and volume of a cube */
#include<stdio.h>
main()
{
    floatside, surface_area, volume;
    clrscr();
    printf("Enter the side of cube");
    scanf("%f", &side);
    surface_area = 6*side*side;
    volume=side*side*side;
    printf("\nSurface area of cube is %2f sq. units\n"),
    surface_area);
    printf("\nVolume of cube is %2f cubic units\n", volume);
    getch(); /* freeze the monitor*/
}
```



Output

```
Enter the side of cube : 3
Surface area of cube is 54.00 sq. units
Volume of cube is 27.00 cubic units
```

3.



```
/* Calculate the sum of average of five numbers */
#include<stdio.h>
main()
{
    float a, b, c, d, e, sum, aveg;
    clrscr();
    printf("Enter the five numbers\n");
    scanf("%f%f%f%f%f", &a, &b, &c, &d, &e);
    /* echo the data */
    printf("\nEntered numbers are");
    printf("%8.2f%8.2f%8.2f%8.2f%8.2f\n", a, b, c, d, e);
    sum = a+b+c+d+e;
    aveg = sum / 5.0;
    printf("\nSum = %2f\n", sum);
    printf("\nAverage=%2f\n", aveg);
    getch(); /* freeze the monitor*/
}
```

**Output**

```
Enter the five numbers
10 25 38 59 13
Entered numbers are 10.00 25.00 38.00 59.00 13.00
Sum = 145.00
Average = 29.00
```

4.



```
/* Leap year checking */
#include<stdio.h>
main()
{
    int year;
    clrscr();
    printf("Enter the year:");
    scanf("%d", &year);
    if(((year%4 ==0) && (year%100!=0)) || (year%400==0))
        printf("\n%d is a leap year\n", year);
    else
        printf("\n%d is not a leap year\n", year);
    getch(); /* freeze the monitor*/
}
```



Output

```

Enter the year : 2004
2004 is a leap year
Enter the year: 2005
2005 is not a leap year

```

Exercises**A. Programming exercises**

1. Evaluate the following C expressions.
 - i. $25/4 + 3 - 7\%3 + 2$
 - ii. $6.5 + (\text{float})5/2 - 3\%8 - 6.5$
 - iii. $(-13\%2)\%(8*2) - 7$
 - iv. $(18-3*3)\%(99-2*10)/(2.5-1.5)$
 - v. $2*((18/5) + (6*(1.5+1)\%(10-2-1)))$
2. Given that initially $i = 0$, $j = 2$ and $k = 3$, find x and the new values of i , j and k for each of the following expressions.
 - i. $x = i++ || ++j \&\& k++;$
 - ii. $x = ((i < j) || j++) \&\& k++;$
 - iii. $k * = (i + j) \% k;$
 - iv. $x = (j == 2) ? k : i$
 - v. $x = (i > j) ? ++i : (k > j) ? j : i$
 - vi. $x = i++ ? j-- : k--;$
 - vii. $k \% = j = (i = 4) \% (j = 3)$
 - viii. $x = j > k ? k > i ? 12 : k > j ? 13 : 14 : 15$
 - ix. $k+ = i++ + ++j * 3$
3. What will be the output of the following?


```

main( )
{ printf( ) ;}
main( )
{ const int i = 10;
i = 20;
}

```

```

main( )
{ printf(" H\re\rll\O");
main( )
{ printf("Hello\n");
main( ) ;
}

```

B. Review questions

1. State the different categories of operators. Explain the arithmetic operators.
2. Explain the use of sizeof () and type cast operator.
3. Explain precedence and associativity of operators.
4. What are the different types of C statements?
5. What is the difference between a statement and a block?
6. Are negative numbers considered true or false by C?
7. Discuss logical operators of C.
8. Explain bitwise operators of C.
9. Discuss various forms of increment and decrement operators.



BUILT-IN OPERATORS AND FUNCTION

4

4.1 INTRODUCTION

All computers programs essentially read, process and display data. Unlike other high level languages, C does not provide built-in input/output statements. All input/output operations have to be carried out by using functions. Many functions for the above purpose have been provided in the C standard input output library (stdio.h). Included in this file are declarations for the I/O functions and definitions of constants (like EOF, NULL, etc.). These functions interact with the standard input (usually the keyboard) and the standard output (usually the screen.) Functions that perform input-output with files are discussed in a later chapter.

4.2 CHARACTER INPUT AND OUTPUT (**getchar** and **putchar**)

The standard library provides several functions for reading and writing one character at a time of which **getchar** and **putchar** are the simplest.

The function **getchar** reads and returns an input character from the standard input device.

Usage

```
variable_name = getchar( );
```

The variable is of char or integer type. **getchar** assigns the character value of the input character to the variable.

Example

```
char c;  
c=getchar( );
```

The function **putchar** writes a single character on the standard output device.

Usage

Usage

Putchar(variable_name);

OR

Putchar(character);

Example:

1. `char c=getchar();`
`putchar(c);`
2. `char ans = 'y';`
`putchar(ans);`
3. `putchar('\ n');` /* positions the cursor to the beginning of
the next line. */

Character test and conversion functions

The header file ctype.h contains declarations of several functions, which are used to test or convert a character.

| Function | Description |
|------------|---|
| isalnum(c) | Returns true if c is an alphanumeric character. |
| isalpha(c) | Returns true if c is an alphabet |
| isdigit(c) | Returns true if c is a digit |
| islower(c) | Returns true if c is a lowercase alphabet. |
| isupper(c) | Returns true if c is an uppercase character. |
| ispunct(c) | Returns true if c is a punctuation mark |
| isspace(c) | Returns true if c is white space characters |
| toupper(c) | Converts c to uppercase if it is a lowercase letter otherwise keeps c unchanged |
| tolower(c) | Returns c converted to lowercase if it is uppercase and unchanged otherwise. |

Example

```
char ch = 'a';
putchar (toupper (ch));
```

will display A on screen.

Character test functions are used with control structures like if, while, etc, which we shall study in the next chapter. However the following program illustrates how they can be used.



```
/* Illustrates character input-output, test and conversion functions*/
```

```
#include<stdio.h>
#include<ctype.h>
main()
{
char ch;
printf("Enter a character:");
ch = getchar( );
if (isalpha(ch))
{ printf (" It is an alphabet");
  if (isupper(ch))
  { printf("\n It is in uppercase \n ");
    putchar(tolower(ch)); /* convert to lowercase */
  }
else
  { printf("\n It is in lower case \n ");
    putchar(toupper (ch));
  }
}
else
  printf("\n Not an alphabet");
}
```

**Output a**

```
Enter a character :*
Not an alphabet
```

Output b

```
Enter a character :b
It is an alphabet
It is in lowercase
B
```

Note: **getch()** and **getche()** can also be used to read a single character as **getchar()**. They are defined in **<conio.h>**. The difference between the two is that **getche()** accepts an input character and echoes (i.e displays) it on screen also whereas **getch()** does not echo it on screen. **getch()** can be used to accept passwords.

4.3 STRING INPUT AND OUTPUT [gets() & puts()]

Two functions **gets()** and **puts()** in the standard input output library are used for string input and output respectively.

gets() accepts a string from stdin (Standard input device). **gets()** continues reading the string, character by character until the 'Enter' key is pressed. The

newline is replaced by a NULL character (\0) at the end of the string. Spaces and tabs are allowed within the string.

Usage: `gets(name_of_string);`

`puts()` outputs a string to the standard output device. It also appends a new-line character at the end.

Usage:

```
puts(name_of_string); or
puts(string literal);
```



/ Illustrates string input-output */*

```
#include<stdio.h>
main()
{ char str[80];
  printf("Type a string less than 80 characters : ");
  gets(str);
  printf("You typed :");
  puts(str);
}
```



Output

```
Type a string less than 80 characters: C is easy!
You typed: C is easy!
```

4.4 GENERAL OUTPUT / FORMATTED OUTPUT (printf)

The `putchar()` and `puts()` functions can be used only with character and string respectively.

A versatile output function is **printf** which can handle any built-in data type and you can specify the format in which the data must be displayed i.e. `printf` displays formatted output to the standard output device. It returns the number of characters actually printed.

Syntax

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

Control string consists of

- Ordinary characters that are printed on screen as they appear.
- Format specifiers or conversion specifiers, which define the output format of each argument.
- Escape sequences like `\n`, `\b`, `\r`, etc.

Format specifier

- There must be exactly the same number of arguments as there are format specifier in the same order.
- Each format specifier begins with a % and ends with a conversion character.
- Between the % and the character, there may optionally be,
 - i. A minus sign for left justification of the argument.
 - ii. A number that specifies minimum field width. If * is given, it implies take next argument as field width.
 - iii. A period, which separates field width from the precision.
 - iv. A number, specifying precision i.e the number of characters to be printed from a string, or the number of digits after the decimal point of a float value, or the minimum numbers of digits for an integer. * means take next argument size.
 - v. h if integer is to be printed as short, l for long and L for long double.

printf conversion character and meaning

| Character | Argument type | Printed As |
|-----------|---------------|--|
| %c | int or char | Single character |
| %i,%d | int | Signed decimal integer |
| %x,%X | int | Unsigned Hexadecimal number using a.....f or A....F. |
| %O | int | Unsigned octal number |
| %f | float, double | Floating point numbers 6 decimal places by default |
| %e,%E | float, double | Floating point numbers in exponential format |
| %g,%G | float, double | Uses %e or %f whichever is shorter. |
| %p | void * | Pointer |
| %% | no argument | Prints a % |
| %u | unsigned int | Unsigned decimal number |

printf conversion character for qualified data types

| Format specifier | Argument type | Output |
|------------------|----------------|------------------------|
| %ld, li | Long | Decimal long integer |
| %Lu | Unsigned long | Unsigned long integer |
| %hd, hi | Short | Decimal short integer |
| %hu | Unsigned short | Decimal unsigned short |
| %le, lf, lg | Double | Signed double |
| %le,lf, lg | Long double | Signed long double |
| %lo | Long | Octal long integer |
| %lx | Long | Hexadecimal long |

Examples

1. `printf("This is a string");`
2. `printf(" ");`
3. `printf("\n");`
4. `printf("The value of x is %d" , x);`
5. `printf("radius %f, area = %f" , rad, area);`
6. `printf(" Hi %d %c %s" , 2, 'U', "Welcome ! ");`

outputs Hi 2 U Welcome !

7. The following statements illustrate the output of number 1234 in different formats.

```
printf ( " %d" , 1234);
printf ( " %2 d" , 1234);
printf ( " %6d " , 1234);
printf ( " % -6d" , 1234);
printf ( " % 06 d" , 1234);
```

| | | | | | |
|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | | |
| 1 | 2 | 3 | 4 | | |
| | | 1 | 2 | 3 | 4 |
| 1 | 2 | 3 | 4 | | |
| 0 | 0 | 1 | 2 | 3 | 4 |

8. Displaying a float value in various formats

```
printf ( " %f " , 12.3456);
printf ( " %8.2f " , 12.3456);
printf ( " %10.2e" , 12.3456);
printf ( " % -10.2e" , 12.3456);
printf ( " %E" , 12.3456);
```

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | . | 3 | 5 | 4 | 6 | 0 | 0 | | |
| | | | 1 | 2 | . | 3 | 5 | | | |
| | | | 1 | . | 2 | e | + | 0 | 1 | |
| 1 | . | 2 | e | + | 0 | 1 | | | | |
| 1 | . | 2 | 3 | 4 | 5 | 6 | E | + | 0 | 1 |

9. Displaying a string " Learn, Write " with different formats.

```
%s
%10s
%.10s
%15 s
% - 15s
%15.10s
% -15.10s
%*.s,15,2
```

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| L | e | a | r | n | , | | W | r | i | t | e | | | |
| L | e | a | r | n | , | | W | r | i | t | e | | | |
| L | e | a | r | n | , | | W | r | i | . | | | | |
| | | | L | e | a | r | n | , | | W | r | i | t | e |
| L | e | a | r | n | , | | W | r | i | t | e | | | |
| | | | | L | e | a | r | n | , | | W | r | i | |
| L | e | a | r | n | , | | W | r | i | | | | | |
| | | | | | | | | | | | | | L | e |

10. `printf (" %d" , printf ("Hello"));` will produce the following output : Hello5

4.5 FORMATTED INPUT (scanf)

The general purpose input function is scanf. It reads characters from the standard input, interprets them according to the format specifics and stores them in the corresponding arguments. It returns a number equal to the number of fields that were successfully assigned values.

Syntax:

```
int scanf ("control string " ,&var1, &var2, .....&varn);
```

The argument, each of which is an address, specifies the location where the corresponding converted input should be stored.

The control string may contain

1. White space characters.
2. Conversion specifications which consists of a % sign, an optional suppression character * , an optional number specifying a maximum field width, an optional h (for short int), l (for long int or double), L (for Long double) and a conversion character.
3. A non-white character which causes scanf to discard the matching character.

scanf conversion characters

| Character | Data read as |
|-----------|---|
| %d | Decimal integer |
| %c | Single character |
| %i | Integer (may be in octal with leading 0 or hexadecimal with leading 0x or ox) |
| %o | Octal integer |
| %u | Unsigned decimal integer |
| %s | Character string |
| %e,f,g | Floating point number |
| %x | Hexadecimal number |
| %[...] | Search sets, which are a sequence of characters. Scanf stops reading a string as soon as a character not in the set is encountered. If the first character in the set is a ^, scanf reads all characters till the first matching character from the set is read from the input. Search sets are case sensitive. |

Examples

1. scanf ("%f", &radius);
2. scanf ("%d %f", &roll_num, &marks);
3. scanf ("%d%s", &age, fname);
(an & is not given with fname since fname will be defined as a string and the name of the string denotes its address)
4. scanf ("%u",&n);
The value of n can be given upto 65535.
5. scanf ("% [abcdef]", address);

- This will read the input characters as long as the input characters are in the search set, abcdef.
6. `scanf ("%[abc]" , address);`
If the input given is Mumbai, only Mum will be stored in address since b is in the search set.
 7. `scanf ("%d%[/-] %d%[/-] %d",&date , &separator, &month , &separator , &year);`
If the date is entered as : 31-12/2000 , the values assigned are

| | |
|-----------|------|
| date | 31 |
| separator | - |
| month | 12 |
| separator | / |
| year | 2000 |
 8. `scanf ("%d * [/-] %d % * [/-] %d", &date, & month &year)`
Here, the suppression character * is used which will skip a / or - (i.e not assign them to any argument)
 9. `printf(" %d", scanf("%d%s", &a, str));`
If the values given are 10 and Hello, the output is 2.

4.6 CONCEPT OF HEADER FILES

C language offers simpler way to simplify the use of library functions to the greatest extent possible.

This is done by placing the required library function declarations in special source files, called header files. Most C compilers include several header files, each of which contains declarations that are functionally related.

`<stdio.h>` is a header file containing declarations for input/output routines; `<math.h>` contains declarations for certain mathematical functions and so on.

The header files also contain other information related to the use of the library functions, such as symbolic constant definitions.

The required header files must be merged with the source program during the compilation process.

This is accomplished by placing one or more `#include` statements at the beginning of the source program.

The other header files are:

`<ctype.h>` character testing and conversion functions.

`<stdlib.h>` utility functions such as string conversion routines , memory allocation routines, random number generator, etc

`<string.h>` String manipulations functions.

`<time.h>` time manipulation functions.

4.7**WHAT IS A PREPROCESSOR?**

A Preprocessor is a program that processes or analyzes the source code file before it is given to the compiler.

It performs the following tasks.

- i. Replaces trigraph sequences (not covered in this book) by their equivalents. Trigraph sequences are used to handle non ASCII characters sets.
- ii. Joins any lines that end with a backslash character into a single line.
- iii. Divides the program into a stream of tokens.
- iv. Remove comments, replacing them by a single space.
- v. Processes preprocessor directives and expands macros.
- vi. Replaces escape sequences by their equivalent internal representation.
- vii. Concatenates adjacent constant character strings.

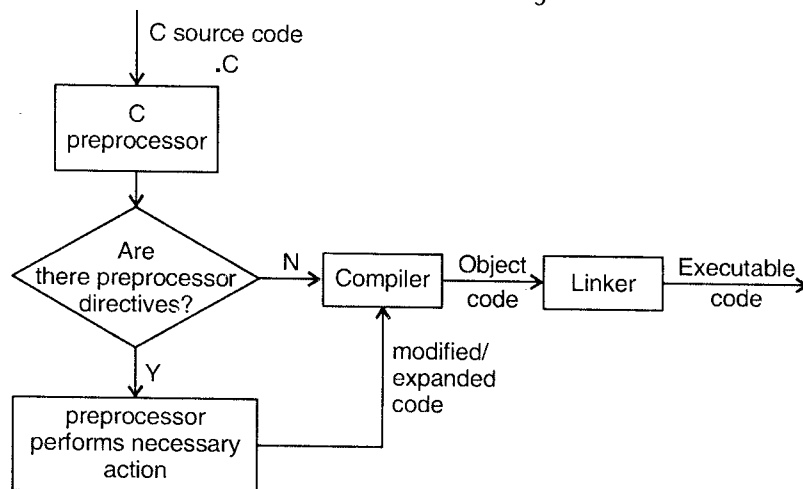


Figure 4.1

4.8**PREPROCESSOR DIRECTIVES**

Preprocessor directives are special instructions for the preprocessor.

- i. They begin with a # which must be the first non-space character on the line.
- ii. They do not end with a semicolon.
- iii. Each preprocessing directive must be on its own line.

Preprocessor directives come under three categories

1. Macro substitution directive.
2. File inclusion directive
3. Conditional compilation directive

4.7.1 Macro substitution directive

A macro is a small subprogram which contains executable code and is similar to a function. Wherever a macro name occurs in a program the preprocessor substitutes the code of the macro at that position (unlike a function). The execution is faster since time is not wasted in function call and return.

a. Simple substitution macro

```
#define macro- id value
```

`#define` is a preprocessor directive that defines an identifier and a value that is substituted for the identifier each time it is encountered in the source file.

We have already used this directive to define symbolic constants.

The identifier is usually written in uppercase to distinguish it from other variables.

- A second `#define` for the same identifier is erroneous unless the second value is exactly identical to the first.
- Use of macros enhances readability of the program.

Examples

- `#define PI 3.142`
- `#define TRUE 1`
- `#define AND &&`
- `#define LESSTHAN <`
- `#define GREET printf("Hello");`
- `#define MESSAGE "Welcome to C"`
- `#define INRANGE (a >= 60 && a < 70)`

Every occurrence of the macro-id in the program will be replaced by its corresponding value.

Example

```
int a = 50;
if (INRANGE)
printf("First class");
```

b. Argumented Macros

An argumented macro is also called a function macro. The macroname can have arguments. Each time the macroname is encountered, the arguments associated with it are replaced by the actual arguments found in the program.

Advantage

1. Their arguments are not type sensitive. Therefore we can pass any numeric variable type to an argumented macro that expects a numeric argument.
2. Argumented macros execute much faster as compared to their corresponding functions.

Example

```
i.
#define HALFOF(x) ((x)/2)
result=HALFOF(10);
```

The occurrence of HALFOF is replaced by

```
Result = ((10 /2));
```

The reason for enclosing x in () is that the parameter could also be an expression in which case, the expression has to be first evaluated. If it is not enclosed in (), it may yield wrong results.

Example

```
result = HALFOF(10+2);
```

This will be evaluated as

```
result = ((10+2) /2);
```

Thus giving the correct result. If no brackets are used, it would evaluate to

```
result = (10+2/2);
```

thereby giving the wrong result.

ii.

```
#define LARGER(x,y) ((x)>(y)?(x):(y))
```

iii. All the parameters of the macro must be used in the substitution value, i.e.

```
#define ADD(x,y,z) ((x)+(y))
```

is invalid because Z is not used. The correct macro is

```
#define ADD(x,y,z) ((x)+(y)+(z))
```

iv.

```
#define SQUARE(X) ((x)*(x))
```

v.

```
#define STREQ(s1,s2) (strcmp((s1),(s2))==0)
```

```
if(STREQ(str1,str2)
```

```
-  
-
```

c. Nested Macros

A macro name can be contained within another macro. This is called nesting of macros.

Example

i.

```
#define CUBE(x) (SQUARE(x)*(x))
```

```
#define MAX(a,b,c) LARGER(LARGER(a,b),c)
```

Macros versus functions

i. Macros are small and do not usually extend beyond one line. They are used when the code is relatively short.

ii. Since the macro is replaced by its code, if a macro occurs many times, the final program contains the expanded code of all the macros; thereby increasing program size.

In contrast, a function code appears only once. A function has space advantage over a macro.

- iii. When a function is called, a certain amount of processing is required to pass control to the function code and return control back to the calling program. This takes a finite amount of time.

This does not occur for a macro because the macrocode is put into the program. Therefore, a macro has a speed advantage over a function.

4.7.2 File inclusion directive

The file inclusion directive is the one that begins with `#include`. We have already used this directive a number of times.

This directive instructs the compiler to include the specified file i.e. it replaces the entire contents of the file at that position.

Syntax

| |
|---|
| <code>#include<filename></code> OR <code>#include"filename"</code> |
|---|

- In the first format, the file is searched in standard directories only.
- In the second, the file is first searched in the current directory. If it is not found there, the search continues in the standard directories.
- Any external file-containing user defined functions; macro definitions etc. can be included.
- An included file can include other files.

Example

```
/* group.h */
#include <stdio.h>
#include <math.h>
#include "myfile.c"
#define PI 3.142
/* mainprog.c */
#include "group.h"
main( )
{
    -
    -
}
```

4.7.3 Compiler Control Directives / Conditional Compilation

Several directives allow compilation of selective portions of the program's code if certain conditions are met. These are,

```
#if
#else
```

```
#elif
#endif
```

They work similar to the if else statement in C. The different formats in which they can be used are as follows

i.

```
# if expression
  statement_block
# endif
```

ii.

```
#if expression
  statement_block1
#else
  statement_block2
#endif
```

iii.

```
#if expression
  statement_block1
#elif expression
  statement_block2
#elif expression
  statement_block3
#else
  default statement_block;
#endif
```

If the constant expression is true, the statement block is compiled otherwise it is skipped and goes to the #else part (if it exists)

Examples

i.

```
#define MAX 10
main()
{ #if MAX>99
  /* Code for larger array */
  #else
  /* Code for smaller array */
  #endif
}
```

ii.

```
#if BACKGROUND==5
  #define FOREGROUND 1
  #elif BACKGROUND==8
  #define FOREGROUND 0
#endif
```

Another method for conditional compilation is the use of `#ifdef`, `#ifndef`

`#ifdef` means if defined and `#ifndef` means if not defined.

In case of a large C program, many macros are defined in various files so it is difficult to remember if a particular macro has been defined or not. In such a case we can check for its definition using the above two macros.

- Redefining an existing macro is erroneous
- Un-defining a non-existent macro is also erroneous.

So the definition of a macro has to be first checked for.

The syntax is

```
#ifdef macro-id
    statement_block
#endif
```

```
#ifndef macro-id
    statement_block
#endif
```

Example

```
#include "declare.h"
#ifndef FLAG
    #define FLAG 1
#endif
```

Un-defining a macro

A macro can be undefined using the `# undef` directive.

```
Example #ifdef FLAG
          #undef FLAG
          #define FLAG 0
          #endif
```

`#ifdef` and `#endif` can be used to compile and run debugging code in the program.

```
Example #define DEBUG 1
main()
{
    _
    #ifdef DEBUG
        /* debugging code put here */
    #endif
    _
}
```

Another important use of conditional compilation directive is when a program has to be run on different machines. In such a case, the common part of program can be run and the machine dependent program part can be conditionally compiled as shown below.

```
main( )
{ #ifdef IBM-Pc
  { code for IBM-Pc}
```

```


#else
{ code for HP machine}
#endif }

```


SOLVED PROGRAMS

1. Let us write a program to accept temperature in °C and convert it to °Fahrenheit using formula temp- in- °F = $\frac{9}{5} \cdot \text{temp-in- } ^\circ\text{C} + 32$.

```

 /* This program converts temperature in degree Centigrade to Fahrenheit */
#include<stdio.h>
main()
{
float centigrade, fahr,; /* declarations*/
printf ("Enter the temperature in Centigrade :");
scanf ("%f", &Centigrade); /* accept input */;
fahr = (9.0/5) * centigrade + 32;
printf ("\n temperature in Centigrade = %f", centigrade);
printf (" \n temperature in Fahrenheit %f" , fahr);
}

```



Output

```


Enter the temperature in centigrade: 37
Temperature in centigrade = 37
Temperature in Fahrenheit = 98.599998

```


2. Write a program to calculate the distance between two points, using formula.

$$d = \sqrt{(y_2 - y_1)^2 + (x_2 - x_1)^2}$$

```

 /*To calculate the distance between two points whose coordinates are (x1,y1)and (x2,y2) */
#include<stdio.h>
#include<math.h>
main()
{
int x1, x2, y1, y2 ;
float d;
printf("Enter the coordinates of the first point:");
scanf("%d%d", &x1,&y1);
printf("Enter the coordinates of the second point:");
scanf("%d%d", &x2,&y2);
d = sqrt((y2-y1)*(y2-y1)+ (x2-x1)*(x2-x1));
printf ("The distance is %lf",d);
}

```



Output

Enter the coordinates of the first point: 10 0
 Enter the coordinates of the second point : 0 10
 The distance is 14.142136

3. Program to convert time in seconds to equivalent hours, minutes, and seconds.



/* This program converts seconds to hours, minutes and seconds*/

```
#include<stdio.h>
/* Define constants */
#define SECONDS_PER_MIN 60
#define MINS_PER_HOUR 60
main()
{unsigned int seconds, minutes, hours, seconds_left, mins_left;
printf("Enter the number of seconds : " );
scanf("%u", &seconds);
hours = seconds / (SECONDS_PER_MIN * MINS_PER_HOUR);
minutes = seconds / SECONDS_PER_MIN;
mins_left = minutes % SECONDS_PER_MIN;
seconds_left = seconds % SECONDS_PER_MIN;
printf("%u Seconds are equivalent to : " seconds);
printf("%u hrs %u mins %u seconds",hours, mins_left,
seconds_left);
}
```

**Output a**

Enter the number of seconds : 60
 60 seconds are equivalent to : 0 hrs 1 mins 0 seconds

Output b

Enter the number of seconds : 20000
 20000 seconds are equivalent to : 5 hrs 33 mins 20 seconds

Exercises

A. Select the Appropriate Answer

1.

```
main()
{ int i;
printf("%d", i );
}
```

- | | |
|----------|------------|
| a. error | b. garbage |
| c. 0 | d. 32767 |

2.

```
main()
{
    int i = 10;
    float j = 20;
    printf("%d", sizeof(i+j));
}
```

- | | |
|------|-------|
| a. 2 | b. 4 |
| c. 1 | d. 30 |

3.

```
main()
{
    int i,
    i = 0x10 + 010+10;
    printf("%d", i);
}
```

- | | |
|-------|------------|
| a. 0 | b. error |
| c. 34 | d. garbage |

4.

```
main()
{
    char ch = 'ABC';
    printf("%c", ch);
}
```

- | | |
|----------|--------|
| a. error | b. ABC |
| c. A | d. ch |

5.

```
main()
{
    printf("\nH\nne\nll\ro");
}
```

- | | |
|----------------------|-----------------|
| a. H e ll o | b. H e 0l |
| c. Hello | d. O |

6.

```
main()
{
    int i = 10 , a ;
    a = i++ / ++i ;
    printf("%d ..%d" ,a , i );
}
```

- | | |
|------------|-----------------------|
| a. 1....12 | b. 10....10 |
| c. error | d. compiler dependent |

7.

```
main()
{ int i = 10 , j = 20;
  i ^=j ; j^=i; i^=j;
  printf("%d, %d" ,i,j );
}
a. 10....10      b. 10....20
c. 20....10      d. 20....20
```

8.

```
enum colors{BLACK, BLUE, GREEN };
main()
{ printf("%d..%d..%d", BLUE , GREEN , BLACK);
}
a. error          b. Blue, Green Black
c. 0...,1...,2   d. 1...,2...,0
```

9. " The stock's value decreased by 10 %"

Which of the following exactly reproduces the above message?

- printf(" The stock's value decreased by 10 %");
- printf("\ The stock's value decreased by %d \% \\ \"\n", 10);
- printf("\ The stock's value decreased by % d %%.\ \"\n", 10);
- None of the above.

B. Predict the outputs

1.

```
main()
{ int a = 300 * 300 , b;
  b = a/2;
  printf( " %d %d", a,b);
}
```

2.

```
main()
{ char ch = 'A';
  int i = 2 ;
  float f = ++ch+i ; }
printf('%f%d%c", f , ch, ch );
```

3.

```
main()
{ int x = 12 , y;
  y = x--;
  y - =--x;
  printf ( "%d%d" ,x,y );
}
```

```
4.
main( )
{ int a = 5 , b = 10 ;
  printf(" %d\n", a++ + b++ + ++a + ++b);
  a = 5 ; b = 10;
  printf(" % d \n " , ++a * ++b);
  a = 5 ; b = 10 ;
  printf(" %d\n", a = ++a * ++ b );
}

5.
main( )
{ int Float = 2 , pi = 3.14;
  printf("%f%f", Float , pi);
}

6.
main()
{ int i,
  i = 32000 + 1536 + 10 * 0;
  printf("%d", i );
}

7.
main()
{ int x,y,z;
  x = y = z = -1;
  z = ++x && ++y || ++z;
  printf("x = %d, y = %d, z = %d", x,y,z);
}

8.
main()
{ char c = 'z',ch ;
  c = c +'a'-'A ' ;
  ch = c -'a'+ 'A';
  printf("%c",ch );
}

9.
main()
{ int i = 10,5 ;
  printf("%d",i);
}

10.
main( )
{ const int x;
  x = 130;
  printf("%d",x);
}
```

11.

```
#define GREAT "xyz"
main()
{ printf(GREAT);
}
```

12.

```
#define GREET HELLO
main()
{ printf(GREET);
}
```

13.

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

14.

```
#define MAIN main()
#define BEGIN
{
  #define END
}
#define GREET printf("Hello")
MAIN
BEGIN
  GREET;
END
```

15.

```
#define SQUARE(x) (x*x)
main()
{ int i = 20, j=10,k;
  k = SQUARE(i-j)
  printf("%d",k);
}
```

16.

```
#define SQUARE(x) (x)*(x)
main( )
{ int i = 20,j=10,k;
  k = SQUARE(i-j);
  printf("%d",k);
}
```

17.

```
#define FLAG
#ifdef FLAG
  int i = 10;
#endif
main( )
{ int i = 5;
  printf("%d",i);}
}
```

18.

```

/* File abc.h */
printf("Hello");
/* File my.c */
main( )
{ #include "abc.h"
printf("C");
}

```

19.

```

/* File xxx.h */
printf("Hello")
/* File my.c */
main( )
{ #include "xxx.h"
;
printf("C");
}

```

C. Programming Exercise

1. Find the roots of a quadratic equation using the formula,

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad \text{Accept values such that } b^2 > 4ac.$$

2. Accept the basic salary of an employee and calculate and display the following.

Dearness Allowance (DA) = 150% of basic

Income Tax (IT) = 30% of basic.

Provident Fund (PF) = 8.33% of basic.

Net Salary = Basic + DA - (IT + PF)

3. Accept two numbers and interchange their values.
4. Given the three sides of a triangle, calculate its area using $\sqrt{s(s-a)(s-b)(s-c)}$ where a,b and c are the three sides and s is the perimeter.
5. The frequency of an electrical circuit is

$$F = \sqrt{\frac{1}{LC} - \frac{R^2}{4C^2}}$$

Write a program that accepts Inductance (L) Capacitance (C) and Resistance (R) of the circuit and calculate its frequency.

6. Write a program to accept a character from the keyboard and check if it is an alphabet, digit or special symbol. If it is an alphabet, check if it is uppercase or lowercase. If uppercase, convert it to lowercase & vice-versa.

D. Review Questions

1. Explain the functions getchar and putchar with examples.
2. Explain the format specifiers used with the printf functions.
3. Explain search sets in the scanf function with examples.
4. Is there a difference between:
printf ("Hello") ; printf ("World");and
puts ("Hello"); puts ("World");
5. What is the difference between getch() and getche() ?
6. What format specifiers are used with scanf ?
7. Write a note on the C Preprocessor.
8. Explain Macro substitution in brief with examples.
9. When an argumented macro is defined, why should each argument be enclosed in parentheses?
10. Do header files need to have a .h extension?
11. Illustrate the use of #ifdef and #undef with examples.
12. Explain any four preprocessor directives.



5.1 INTRODUCTION

In the previous chapters, we have studied some basic input output functions. We have also seen the different types of C statements. In this chapter, we shall be studying the program control statements, which specify the order in which instructions are executed.

Sometimes, it is necessary to alter the sequence of execution of statements based on certain conditions or we may require some statements to be executed repeatedly until some condition is met. This involves decision-making, and looping. In addition we shall also be studying the jump statements, which allow breaking out of decision and loop control statements.

5.2 SELECTION / DECISION MAKING STATEMENTS

Many programs require testing of some conditions at some point in the program and selecting one of the alternative paths depending upon the result of the condition.

C provides three mechanisms to check for conditions and execute or skip certain parts of the program. The three decision-making statements are:

1. if statement
2. if-else statement
3. switch statement

5.2.1 *if statement*

This is the simplest form of decision-making statements in C. It allows decisions to be made by evaluating an expression. Depending upon the result (True or False), the program execution proceeds in one direction or another. Basically it is a two-way decision statement.

The simplest form is:

```
if(expression)
statement
```

Note: Here, statement could be either a single statement or a block of statements (enclosed in braces) as shown below. Henceforth, we shall use **Statement** to imply both.

```
if(expression)
    statement;
```

.single statement

```
if(expression)
{.....
    statements;
... ..
}
```

more than one statement

The keyword **if** must be followed by a set of parentheses containing a **single expression** to be tested. The statement is executed only if the expression is true (i.e. non-zero). If the condition evaluates to false, the statement is skipped.

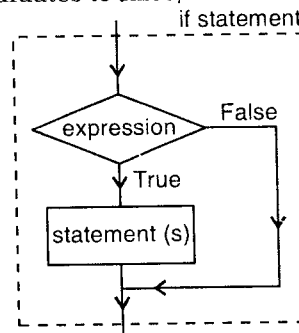


Figure 5.1

Example

i.

```
if(n < 0)
    printf("The number is negative");
```

ii.

```
if(age < 30 && salary >10000)
    printf("You are young and rich !!");
```

iii.

```
if((n % 3 == 0 ) && ( n % 5 == 0))
    printf("The number is divisible by 3 and 5");
```

iv.

```
if(basic_sal > 10000)
{
    it = 30.0 * basic_sal / 100;
    da = 200.0 * basic_sal / 100;
    hra = 800.0 ;
}
```

5.2.2 if ... Else statement

The 'if' statement will execute the statement if the expression is true otherwise it will be skipped.

However, in many cases we require an alternate statement to be executed if the expression evaluates to false. This is possible using an if ...else statement.

The general form is,

```
if(expression)
    statement1
else
    statement2
```

Here, the expression is evaluated. If it is true, **statement1** is executed and if it is false, **statement2** is executed. Thus, either **statement1** or **statement2** will be executed; never both.

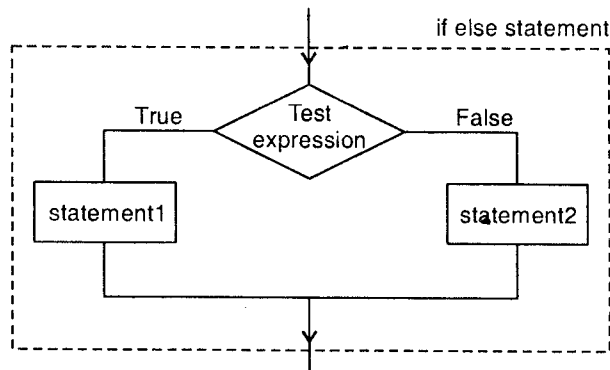


Figure 5.2

Examples

1.

```
if(a > b)
    printf("a is larger");
else
    printf("b is larger");
```

2.

```
if(year % 4 == 0 && year % 100 != 0 || year % 400 == 0 )
    printf("%d is a leap year", year);
else
    printf("%d is not a leap year", year);
```

This can also be written using the conditional operator ?:

```
(year % 4 == 0 && year % 100 != 0 || year % 400 == 0 ) ?
printf ("leap"): printf ("Not Leap");
```



```
3.
if(number % 2 == 0)
    printf("The number is even");
else
    printf("The number is odd");
```

```
4.
if(basic_sal < 10000)
{
    it = 20 * basic_sal / 100;
    da = 150 * basic_sal / 100;
    hra = 500;
}
else
{it = 30 * basic_sal/100;
 da = 200 * basic_sal / 100;
 hra = 800;
}
```

5.2.3 Nested if ...else statements

As seen earlier, the if clause and the else part may contain a compound statement.

Moreover, either or both may contain another if or ifelse statement. This is called as nesting of ifelse statements.

This provides a programmer with a lot of flexibility in programming. Nesting could take one of several forms as illustrated below.

```
i.
if(expression1)
    statement1
else
    if(expression 2)
statement2
```

```
ii.
if(expression1)
    if(expression2)
        statement1
    else
        if(expression3)
            statement2
```

```
iii.
if(expression1)
    if (expression2)
        statement1
```

```
else
    statement2
else
    statement3
iv.
if(expression 1)
    statement 1
else
    if(expression 2)
        statement 2
    else
        statement 3
v.
if(expression1)
    if(expression2)
        statement1
    else
        statement2
else
    if(expression3)
        statement3
    else
        statement4
```

Examples

i.

```
if(a >b)
    if (a > c)
        printf( "a is largest");
    else
        printf( "c is largest")
else
    if(b > c)
        printf("b is largest")
    else
        printf("c is largest");
```

ii.

```
if ((ch >= 'a' && ch <= 'z' ) || (ch > 'A' && ch <= 'Z' ))
    printf( " %c is an alphabet" , ch);
else
    if (ch >= '0'&& ch<='9')
        printf("%c is a digit", ch) ;
    else
        printf("%c is a special symbol", ch) ;
```

Note: It is a good idea to enclose each of the 'if' and 'else' blocks in braces if the logic is complex.

Example: A recruitment agency recruits candidates satisfying the following conditions.

- i. If the candidate is male, between 25 and 30 years of age, height above 160 cm.
- ii. If the candidate is female, between 20 and 25 years of age with height above 155 cm.

The if-else construct for the above can be written as follows:

```
if (sex == 'M')
{
    if (age >= 25 && age <= 30)
    if (height > 160)
        printf("Candidate is recruited");
}
else /* Candidate is Female */
{
    if (age >= 20 && age <= 25)
    if (height > 155)
        printf("Candidate is recruited");
}
```

Note: else always gets associated with the nearest if statement. Hence { } should be used to associate the else with the correct if.

5.2.4 The else - if ladder

If there is an if else statement nested in each else of an if- else construct, it is called an else - if ladder as depicted below.

```
if (expr1)
    statement1;
else
    if (expr2)
        statement2;
    else
        if (expr3)
            statement3;
        else
            statement4;
```

This can also be written as

```
if (expr1)
    statement1;
else if (expr2)
    statement2;
else if (expr3)
    statement3;
else
    statement4;
```

The conditions (expressions) are evaluated from the top downward. As soon as a true expression is found the statement associated with it is executed and the rest of the ladder is bypassed. If none of the expressions are true, the final else is executed. The last else often acts as a default condition i.e. if all other tests fail, the last else statement is executed.

If it is not present, no action takes place if all other conditions are false.

Examples

1. **To check whether a character entered from the keyboard is an alphabet, digit, a special symbol or punctuation mark.**

```
if (isalpha(ch)/*ch is the character variable storing the
character */
printf("11%C is an alphabet",ch);
    else
        if (isdigit(ch))
            printf("%c is a digit", ch);
    else
        if (ispunct(ch))
            printf("%c is a punctuation mark" , ch);
        else
            printf( "%c is a special symbol", ch);
```

2. **To find class of a student from the marks.**

```
if (marks >= 70)
    printf ("Distinction");
else if (marks> = 60 )
    printf("First class");
else if ( marks> = 50)
    printf("second class");
else if ( marks>=40 )
    printf("Pass class");
```

5.2.5 The switch statement

Whenever one of many alternatives is to be selected, nested if – else statements can be used. However, the structure becomes very complicated and the code becomes difficult to read and trace.

For these reasons C has a built-in multiple-branch decision statement called **switch**. This statement tests whether an expression matches one of a number of constant integer values and branches accordingly.

The format is

```
switch (expression)
{
case const-expr1 : statement
case const-expr2 : statement
case const-expr3 : statement
.
.
.
.
default : statement
}
```

As mentioned before **statement** implies a single statement or a compound statement.

- The expression enclosed within parentheses (integer expression) is successively compared against the constant expression (or values) in each case. They are called case labels and must end with a colon (:)
- The statement in each case may contain zero or more statements. If there are multiple statements for a case they need not be enclosed in braces.
- All case expressions must be different.
- The case labeled default is executed if none of the other cases match. The default case is optional and if not included , no action takes place at all if none other match.
- Cases and the default case can occur in any order.
- More than one case value may be associated with a particular statement.



```
/* Use of switch statement */
#include<stdio.h>
main( )
{ int number ;
  printf("Enter a number between 1 and 3 :");
  scanf("%d",&number);
  switch(number)
  {
    case 1 : puts("you entered 1\n");
    case 2 : puts("You entered 2\n");
    case 3 : puts("You entered 3\n");
    default : puts("Out of range\n");
  }
}
```



Output

```
Enter a number between 1 and 3:2
You entered 2
You entered 3
Out of range
```

However, this is not the required output. The output is like this because when a match occurs, not only the statement associated with the matching case is executed but those of all the remaining cases are also executed. Using a break statement can solve this problem.

Use of break statement

The break statement is used to exit a control structure. As soon as a break statement is encountered, program control is transferred to the first statement outside the structure to which the break belongs.

In the above program, if a break statement is included in every case, as soon as a match is found, the statement(s) of the matching case will be executed and the break statement will take control outside the switch statement as illustrated below. The default case need not have a break statement since it will be the last case executed if no others match.

Example**Illustration of switch using break**

```
#include<stdio.h>
main()
{int number;
 printf("Enter any number between 1 and 3 :");
 scanf("%d", &number);
 switch(number)
 {
     case 1 : puts("You entered 1\n");
             break;
     case 2 : puts("You entered 2\n");
             break;
     case 3 : puts("You entered 3\n");
             break;
     default : puts("Out of range.\n");
 }
}
```

**Output a**

```
Enter any number between 1 and 3:2
You entered 2
```

Output b

```
Enter any number between 1 and 3:10
Out of range.
```

Note: To associate more than one case value with a particular statement, you have to simply list the multiple case values before the common statement (s) that are to be executed. This is called-falling through cases.

Examples

1.

```
switch (operator)
{
    ...
    case '*' :
    case 'X' : result = value1 * value2;
              printf ("%f", result);
              break;
    ...
    ...}
```

2.

```
switch (c)
{ case '0' : case '1' use: case '2' : case '3':
  case '4' : case '5' : case '6' : case '7':
  case '8' : case '9': digit ++ ; break ;
  case ' ' : case '\n' : case '\t' : white_space + + ; break;}
```

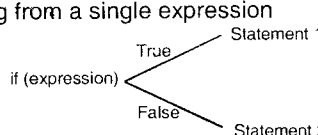
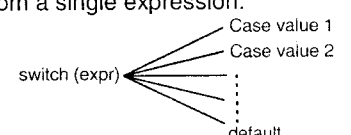
Nested 'switch' statement

It is possible to have a switch statement as a part of a statement in another **switch** statement. Even if the case constants of the inner and outer switch contain common values there is no conflict. *Example*

```
switch (x)
{ case 0 : printf ("Invalid value");
  break;
  case 1 : switch (y)
    { case 0 : printf ("values are 1 and 0");
      break ;
      case 1 : printf ("values are 1 and 1");
        break; }
    break;
  case 3 :
  :
  :
}
```

Comparing if-else and switch statements

Although both these statements can be used for multi-way decision-making, there are some differences between the two, which are crucial for the selection of one of these in a program.

| No. | If-else structure | Switch statement |
|------|---|--|
| i. | The if-else structure allows only two-way branching from a single expression  | Switch allows multi-way branching from a single expression.  |
| ii. | The nested if-else structure is nonelegant and complicated | Switch statement is very elegant and easier to write. |
| iii. | If multiple alternatives exist, the nesting can go to many levels and it becomes difficult to match the else part to its corresponding if. | No such problem occurs using a switch statement |
| iv. | Debugging becomes difficult | Tracing of errors and debugging is easy. |
| v. | The test expression can be a constant expression or an expression involving relational or logical operators. Float and double are also allowed. | Only constant integer expressions and values are allowed. |
| vi. | Multiple statements within if or else have to be enclosed in braces. | The statements belonging to a case need not be enclosed in braces. |

5.2.6 Conditional operators

The ternary operator? : can also be used for decision-making. We have already seen how this operator works. The general form is

```
expr1? expr2 : expr3
```

If expr1 is true, the entire expression takes the value of expr2 else it takes the value of expr3.

Examples

1.

```
char ch;
ch = getchar( );
x = (ch >= 65 && ch <= 90 ) ? 1: 0;
x? puts("Uppercase alphabet"):puts("Other character");
```

This piece of code checks if character ch is an uppercase alphabet.

2. The following statement assigns the largest of three numbers (a,b,c) to x.

```
x = ( a > b ) ? ( a > c ) ? a : c : ( b > c ) ? b : c ;
```

5.3 ITERATIVE STATEMENTS (LOOP CONTROL STRUCTURE)

A segment of program code that is executed repeatedly is called a loop. The repetition is done until some condition for termination of the loop is satisfied.

A loop structure essentially contains

- i. a test condition
- ii. loop statement(s)

The test condition determines the number of times the loop body is executed. It involves evaluating a loop control variable(s), whose value has to change within the loop body so that the loop execution can terminate.

The iteration procedure takes place in four steps.

- Initializing the loop control variable.
- Execution of loop statements
- Changing the value of the control variable
- Testing the condition.

Depending upon when the loop condition is tested, loops can be of two types:

1. Top-tested loop (entry controlled loop)
2. Bottom tested loop (exit controlled loop)

In an entry-controlled loop, the condition is evaluated before the loop body is executed. In the bottom tested or exit controlled loop, the condition is tested after the loop body is executed.

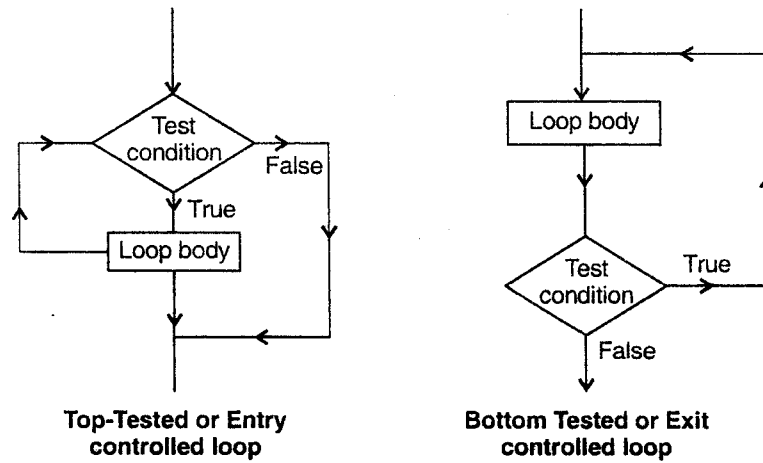


Figure 5.3

The C language provides three loop structures for use in programs.

1. while statement
2. do...while statement
3. for statement

5.3.1 The while statement

The while loop is the simplest loop structure. It is often used when the number of times the loop is to be executed is not known in advance but depends on the test condition.

It is an entry-controlled loop i.e. the condition is tested before the loop body is executed.

The syntax of the loop is:

```
while (expression)
    statement
```

The expression is the test conditions and can be any valid C expression.
The statement can be a single or compound statement.

How it works?

- The expression is evaluated and the statement (loop body) is executed as long as the expression is TRUE (non zero)
- As soon as the expression evaluates to false, the execution of the loop body is stopped and control is transferred to the first statement outside the loop body.
- Since it is an entry-controlled loop, if the expression evaluate to false the first time itself, the loop body will not be executed even once.

Example: Program displaying all even numbers below 50.

```
/* Demonstration of a simple while loop */
#include<stdio.h>
main( )
{ int even_number = 0; /* Initialization */
  while(even_number < 50) /* Loop condition */
  {
    printf("%d \n", even_number); /*Display */
    even_number = even_number+2; /*change value of loop variable*/
  }
}
```

Points to remember

- The loop control variable(s) must be initialized (i.e. given some value before the condition is tested)
- The loop body must contain a statement to alter the value of the control variable.

Examples

1. Calculate the sum of numbers from 1 to n (user specified) i.e. $1+2+3+\dots+n$.



/* Illustrates while loop */

```
#include<stdio.h>
main( )
{int sum = 0, n, loop_var =1; /* Initialization*/
  printf("enter the value of n : ");
  scanf("%d",&n);
  while (loop_var <= n)
  {
    sum = sum + loop_var;
    loop_var++;
  }
  printf("\n The sum of numbers from 1 to %d is %d", n, sum);
}
```



2. To accept characters from the keyboard till the user enters * and count the total number of alphabets entered.



```
#include<stdio.h>
main( )
{
    char ch;
    int counter = 0;
    ch = getchar( ); /* Get the first character */
    while(ch != '*')
    { if (isalpha(ch)) /* check if ch is an alphabet */
      counter++ ;
      ch = getchar( ); /* alter value of loop variable */
    }
    printf("Number of alphabets are %d",counter);}

```

The loop can be written in another way as shown:

```
while((ch= getchar( )) != '*')
{
    if (isalpha(ch))
        counter++;
}

```



Here, `ch = getchar()` is enclosed in `()` because `!=` has higher precedence over `=`. The character has to be read first and then compared. Hence the `()`.

3. Accept numbers, as long as user says 'y' and calculate their sum



/* Program to accept numbers, from the keyboard as long as the user says 'y' and find their sum */

```
#include<stdio.h>
main( )
{ char ans = 'y';
  int sum = 0 , num;
  while (ans == 'y')
  { printf("enter the number :");
    scanf ("%d", &num);
    sum = sum + num ;
    printf("\n do you want to continue (y/n);");
    ans = getchar( );
  }
  printf("\n The sum is %d", sum);
}

```



4. "To find sum of digits of an integer.

/* Program to accept an integer and calculate the sum of its digits */

```
#include<stdio.h>
main( )
{ int number, sum = 0 ;
  printf("Enter the number : ");
  scanf("%d",&number);
  while(number > 0)
  {
    sum = sum+(number%10);/*Add the last digit of number to sum */
    number= number /10 ; /* Get the remaining digits in number */
  }
  printf("\n The sum of digits is %d" sum);
}
```

**Output**

```
Enter the number: 327
The sum of digits is 12
```

5. To reverse a number

/* Program to reverse a number i.e. if user enters 324, the output should be 423 */

```
#include<stdio.h>
main( )
{ int num , rev_num = 0;
  printf("Enter the number to be reversed ");
  scanf("%d", &num);
  while(num>0)
  {
    rev_num = rev_num * 10 + (num % 10);
    num = num /10;
  }
  printf( "In The reversed number is %d" rev_num);
}
```

**Output**

```
Enter the number to be reversed 5678
The reversed number is 8765.
```

Nested 'while' statement

Just like the 'if' statement, while statements can also be nested. Nesting of loops means a loop that is contained within another loop.

```
while (expr1)
{
  while (expr2)
```

```

{
loop body of while (expr2);
}
}

```

Nesting can be done upto any levels. However the inner loop has to be completely enclosed in the outer loop. No overlapping of loops is allowed.

Nesting of loops is required in many programming exercise like multidimensional arrays etc.

Example Program: To display the following structure

```

1
1 2
1 2 3
1 2 3 4

```

i.e. 1 to n rows and numbers from 1 to n in the nth row.



/* Program to display triangle of numbers */

```

#include <stdio.h>
main( )
{ int n , line_number , number;
printf("How many lines: ");
scanf("%d",&n);
line_number = 1 ; /* Initialize line number */
while (line_number <=n) /* line number goes from 1 to n */
{ number = 1 ; /* display begins from 1 */
while (number <= line_number)
{ printf ("%d\t", number);
number++; /* next number */
}
printf ("\n");
line-number++ ; /* next line*/
}
}

```



In the above program, the outer while loop is for the lines from 1 to n. For each line, we have to print numbers from 1 to the line numbers. This is done by the inner loop. i.e. for every value of line-numbers, number takes values from 1 to line_number.

5.3.2 The do-while loop

The second iteration statement provided by C is the **do-while** statement.

The while loop seen earlier is top-tested i.e. it evaluates the condition before executing any of the statements in its body. The do-while loop, on the other hand, is a bottom-tested or exit controlled loop i.e. it evaluates the condition after the

execution of statements in its construct. This means that the statement within the loop are executed at-least once.

The syntax is

```
do
  { statement}
  while
(expression);
```

The statement (single or compound) is executed as long as the expression is true.

Note the ; following the while.

The sequence of events is:

1. The statement(s) in statement are executed.
2. Expression is evaluated. If it is true, execution returns to step 1. If it is false, execution of the loop terminates.

Example

```
do
{ printf("\n 1 - Add a record");
  printf("\n 2 - Delete a record");
  printf("\n 3 - View Records");
  printf("\n 4 - Quit");
  printf("\n Enter your choice: ");
  scanf("%d" &choice);
  switch (choice)
  {
  case 1 : add( );
           break;
  case 2 : delete( );
           break ;
  case 3 : view( );
           break;
  case 4 : printf("Bye");
           }
}while (choice!=4);
```

The above program code shows a do while loop, which displays a menu and accepts a choice.

In this case, we want the menu to be displayed and choice to be accepted at least once and so a do_while loop is preferred.

5.4 THE FOR LOOP

The for loop is very flexible, powerful and most commonly used loop in C. It is useful when the number of repetitions is known in advance.

This is a top-tested loop similar to the while loop but the advantage is that it combines the initialization test condition and loop variable alteration statement in a single statement.

The syntax is:

```
for (expr1 ; expr2 ; expr3)
    statement
```

where expr1 is the initialization expression

expr2 is the test condition

expr3 is the update expression

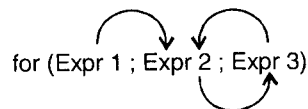
These three expressions have to be separated by semicolon (;).

The above loop is equivalent to

```
expr1;
while (expr2)
{ statement
  expr3;
}
```

Execution of a for loop

- expr1 is evaluated only once i.e. at the beginning. This expression performs initialization of the loop control variable (Multiple initializations can also be done as seen later)
- expr2 is the test expression, which is evaluated before execution of statements in the loop. The statements are executed only if the test expression is true. If it is false, the loop execution terminates. Note that there can be only a **single** test expression.
- expr3 is the update expression, which alters the value of the loop control variable.



Execution of for

Example:

```
for (i= 1 ; i <=100 ; i++)
    printf("%d \n" ,i);
```

i = 1 → initialization

`i <= 100` → test expression

`i++` → update expression

Different forms of the 'for' loop.

- i. `for (i= 0; i < 25 ; i++)`
`statement ;` → single statement
- ii. `for (i = 0; i < 25 ; i++)`
`{ statement ;`
`.....`
`statement;` → compound statement
`}`
- iii. `for (i = 0 ; i < 25 ; i++)`
`;`
`or`
`for (i = 0 ; i < 25 ; i++);` → loop with no body.
- iv. `for (i = 0 , j = 0 ; i < 25 ; i++ , j++)`
`statement ;` → Multiple initialization and
multiple updates separated
by comma
- v. `for (; i < 25 , i ++)` → Initialization expression not used.
- vi. `for (; i < 25 ;)` → Initialization and update
expression not used
- vii. `for (;;)` → All three not used.
`printf("Forever \n");`

Examples:

i.

```
for(i=1,j=50;i<=20||j>=10;i++ j-- )
    printf( "\n %d %d",i,j);
```

ii.

```
for(temp=0;temp<=50;temp=temp+5)
{
    fahr = ( 9*temp) /5 + 32);
    printf("\n centigrade = %f Fahrenheit = %f",temp, fahr);
}
```

iii.

```
/* Accepts values from user till 99 is entered */
int num = 0;
for(;num!=99;)
    scanf("%d",&num);
```


iv.

```
for (i = 0 ; ++i<10;)
    printf("%d \n",i)
```

Example

1. Calculation of factorial of a number. We know that $n! = n \times (n-1) \times (n-2) \dots \times 1$. Thus we have to repeatedly decrement n by 1 till 1 and multiply each value to the previous product.

Note: we can also increment from 1 to n and perform multiplication.



/* Calculation of factorial */

```
#include<stdio.h>
main( )
{ int num, product ;
  printf("Enter the number:");
  scanf(" %d",&num);
  for (product = 1 ; num >= 1 ; num--)
      product = product * num;
  printf("\n the factorial is %d",product);
}
```



Output

```
Enter the number : 5
The factorial is 120
```

Note: The for loop could also have been written as :

```
for (i= 1, product = 1; i <= num ; i++)
    product = product * i;
```

2. To calculate x^y where x is a float and y is an integer.



/* Calculation of x^y */

```
#include<stdio.h>
main()
{
    float x, power = 1, i ;
    int y ;
    printf("Enter the base and power :")
    scanf("%f %d",&x, &y);
    for(i=1;i<=y;i++)
        power *= x ;
    printf("\n %f raised to %d is %f",x,y,power);
}
```




```

printf("%d\t", number++);
printf("\n");
}
}

```



2. To display a rectangle of n rows and m columns filled with the character '*'.

```

*****
*****
*****
*****

```

} 4 rows

└──────────┘
8 columns



```

/* Display a rectangle of n rows and m columns */
#include<stdio.h>
main( )
{ int n_rows, mcols i, j,;
  printf("Enter the number of rows:");
  scanf("%d",&nrows);
  printf("\n Enter the number of columns :");
  scanf("%d",&mcols);
  for (i=1;i<=nrows ;i++)
  { for (j =1;j<=mcols ; j++)
    printf("*");
    printf("\n"); /* Go to the next line after each row */
  }
}

```



3. To display multiplication tables 2 to 9 (n multiples each). The required display is:

$2 \times 1 = 2$ $3 \times 1 = 3$ $9 \times 1 = 9$

$2 \times 2 = 4$ $3 \times 2 = 6$ $9 \times 2 = 18$

If the multiples do not fit on a single screen, display each screen after a pause.
(about 24 multiples will fit on a screen)



```

/* Multiplication Tables */
#include <stdio.h>
main( )
{ int table_of , multiplier, n, , count = 1;
  printf("\n How many multiples ? : ");
  scanf("%d",&n);
  for(multiplier = 1,multiplier<=n; multiplier++,count++)
  {
    for (table_of= 2 ; table_of <= 9 ; table_of++)

```

```

printf("%2d x %2d = %3d \t", table_of, multiplier, table_of *
multiplier);
printf("\n");
if (count %24==0) /* Screen full */
{ printf (" Press any key to continue...");
  getch (); clrscr();
}
}
}

```

This program, for each value of multiplier, table_of varies from 2 to 9 thereby giving each row.

4. To display 'n' lines of the structure from the center of the first line on screen.

```

      *
     **
    ***
   ****
  *****

```

↓ n lines



/* Triangle using the * character */

```

#include<stdio.h>
main( )
{
  int spaces = 39, n, no_of_stars ,line_no, s;
  printf("Enter the number of lines: ");
  scanf("%d \n",&n);
  for (line_no = 1; line_no<=n;line_no++)
  {
    for (s=1;s<=spaces; s++)
      printf(" "); /* display spaces*/
    for (no_of_stars = 1; no_of_stars<=line_no; no_of_stars++)
      printf("*");
    printf("\n");
    spaces--; /* reduce number of spaces by 1*/
  }
}

```



Note: Instead of using a loop to display spaces, we can use a single printf statement as :

```
printf("%*S", spaces, " ");
```

5.5 JUMP STATEMENTS

5.5.1 Break and continue

We have already seen the use of the break statement in the switch-case statement. It also has one more use.

Sometimes, it is required to exit a loop as soon as a certain condition is met i.e to force immediate termination of a loop bypassing the normal loop condition test.

When the break statement is encountered inside a loop, the loop is immediately terminated. Subsequent statements in the loop are skipped and program control resumes at the next statement following the loop.

Format

```
break;
```

Example: The following program checks whether a number is prime or not. To check a prime number, we successively divide it by 2 to number -1. If it is divisible the number is not prime. Thus, as soon as we get a 0 remainder, we have to break out of the loop.



```
#include<stdio.h>
main()
{ int number, i ,prime = 1;
  printf("enter the number: ");
  scanf("%d", &number);
  for(i=2;i <number; i++)
  {
    if(number %i == 0)
    { prime = 0;
      break; } }
  if (prime==0)
    printf("\n The number is not prime");
  else
    printf("\n The number is prime");
}
```

Note: If there are nested loops, the break statement will cause exit only from the innermost loop.

Example

```
count =1;
for (i=1;i<=5;,i++)
{ for (j=1;j<=5;j++)
{
  printf("Enter a number:");
  scanf("%d",&n);
  if (n<0)
    break;
}
count++;
}
```



Here, if the user enters a negative number, the block statement will take control to the statement `count++`, in the outer loop.

Continue statement

The continue statement is somewhat similar to the break statement except that it does not cause the loop to terminate. It bypasses the remaining statements and it forces the next iteration of the loop to take place as usual.

Format:

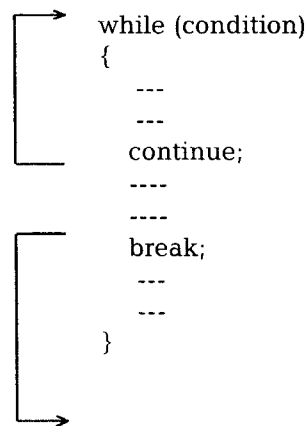
```
continue;
```

Example:

```
do
{ printf("Enter a number :");
  scanf ("%d",&n);
  if (n<0)
    continue;
  sum = sum + n;
} while (n! = 999);
```

This code accepts integers and calculates the sum of only positive numbers. The loop terminates after the user enters 999.

In the case of for loop, first the increment part of the loop is performed, next the condition is tested and finally the loop continues.



Examples

```
1.
int i=5;
while (i)
{ i--
  if(i == 3)
    break;
  printf("%d",i);
}
o/p 4
```

```
2.
int i=5,
while(i)
{ i-- ;
  if(i== 3)
    continue;
  printf("%d",i);
}
o/p 4210
```

5.5.2 goto and labels

The goto statement is an unconditional jump statement. The goto statement (although not used frequently) is used to alter the normal sequence of program execution by unconditionally transferring control to some other part of the program.

Format

```
goto label;
```

The statement where control has to be transferred is identified by the label.

- A label is a valid C identifier.
- A label is followed by a colon.
- The label can be attached to any statement in the same function as the goto.
- The label does not have to be declared like other identifiers.

Example

```
X=1;
loop:
  X++;
  if(X<100)
    goto loop;
```

One good use for the goto statement is to come out of several layers of nesting.

Example

```
for (...)
{ for (...)
  { while (...)
    { ...
      if (error)
        goto out;
      ...
    }
  }
}
out:
... ..
```

Note: Control cannot be transferred from outside to within a loop using the goto statement.

5.5.3 Using exit() function

The exit () function causes immediate termination of the entire program.

The exit () function is called with an argument 0 to indicate that termination is normal. Other arguments are used to indicate some sort of error.

A common use of exit () occurs when some mandatory condition for program execution is not satisfied. Invalid password entered, absence of color graphics card for running computer games, negative or invalid input entered, etc.

Example:

```
main()
{
    int code;
    printf("Enter the security code:");
    scanf("%d",&code);
    if(!valid(code))
        exit(0);
    ...
    ...
    ...
}
```

In this example, a user-defined function valid (code) accepts the code and validates it. If invalid, it returns 0 and 1 if valid. If the code is not valid, the program execution is terminated.

Another use could be in the switch case statement as shown to stop program execution if user enters 4.

```
do
{ ch = getchar( );
  switch (ch)
  { case '1' : add_record( );
    break;
    case '2' : delete_record( );
    break;
    case '3' : view_records( );
    break;
    case '4' : exit(0)
  }
} while (ch!='4');
```


SOLVED PROGRAMS

1. To count the number of words, lines and sentences in the text.
We will define a flag called status, this flag will contain 0 if we are OUT of a word and it will contain 1 if we are within a word.



```
/* Counts number of words, lines and sentences in the text */
#include<stdio.h>
#define IN 1
#define OUT 0
main( )
{
    int wordcount = 0, linecount = 0, sentcount = 0, status = OUT;
    char ch;
    printf("\n Enter the text - ctrl z to terminate \n");
    while ((ch=getchar( )) != EOF)
    {
        switch (ch)
        {
            case ',' : case ';' : case '.' :
                if (status == IN)
                { wordcount++;
                  status = OUT;}
                break;
            case '\n' : linecount++;
                if (status == IN)
                { wordcount++;
                  status = OUT;
                }
                break;
            case '.' : sentcount++;
                if(status == IN)
                { wordcount++;
                  status = OUT;
                }
                break;
            default : status = IN;
        } /* end of switch */
    } /* end of while */
    if (status == IN)
        wordcount ++;
    printf("\n Number of lines = %d",++linecount);
    printf("\n Number of sentences = %d", sentcount);
    printf("\n Number of words = %d", wordcount);
} /* end of main */
```



2. To display the first 'n' prime numbers.



```

/* First n prime numbers, use of nested loops */
#include<stdio.h>
#define PRIME 1
#define NOTPRIME 0
main( )
{ int n, divisor, flag = PRIME, number, count =1;
  printf("\n How many prime numbers ? : ");
  scanf("%d",&n);
  printf("\n The first %d prime numbers are : \n");
  printf("2\t");
  number = 3;
  while (count<=n)
  { /* check if number is prime */
    for (divisor =2; divisor<=n-1;divisor++)
    { if (n% divisor == 0)
      { flag = NOTPRIME;
        break; } }
    if (flag == PRIME) /* if number is prime */
    { count ++ ; printf("%d \t", number) };
    flag = PRIME; /* reset flag */
    number++; /* check if next number is prime */
  } /* end of while */
}/* end of main */

```



Output

```

How many prime numbers? : 5
The first 5 prime numbers are :  2 3 5 7 11

```

Exercises

A. Predict the output of the following.

i.

```

main()
{ int x = 1;
  switch (x)
  { case 0 :x= 1;
    case 1 :x= 3;
    case 2 :x+= 4;
    case 3 :x = 2;
    default:x+= 2;
  }
  printf("%d" x);
}

```

ii.

```
main( )
{ int x=5,y=50,z=(x+y)*10;
  while (x<=5)
    x=y/x;
}
```

How many times will the loop execute?

iii.

```
int l = 4;
switch (l)
{ default : printf("A");
  case 1 : printf("B");
  case 4 : printf("C");
}
```

iv.

```
int i=5;
while (i)
{ i--;
  if (i==3)
    continue;
  printf("\n Hello");
}
```

v.

```
int i=3;
while (i)
{ i=100;
  printf("%d..",i);
  i--;
}
```

vi.

```
main()
{ int i,j,k;
  for (j=1;j<=4;j++)
  if(j*4==12)
    goto there;
  else
    printf("here\n");
  for (i=1,i<=5 i++)
  { k = i*i;
    there : printf("there\n");
  }
}
```

vii.

```

main()
{ int c=97;
  switch(c);
  { case 'a':
    if (c>3)
      case 'b':
        c=10;
        printf("%d",c);
    }}

```

B. Programming exercises

1. Write a program to display all Armstrong numbers below 1000.
(An Armstrong is a number whose sum of cubes of digits is the number itself.
e.g. $153 = 1^3 + 5^3 + 3^3$)
2. Display all perfect numbers below 500.
(A perfect number is a number, such that the sum of its factors is equal to the number itself. $6 = 1 + 2 + 3$)
3. Display the first 'n' terms of the Fibonacci series. (each term = sum of previous two terms).
4. Accept an integer and display its prime factors.
5. Find the sum of first 'n' terms of the following series
 - i. $1+3+5+\dots$
 - ii. $x + x^3 + x^5 + \dots$
 - iii. $\frac{1}{1!} + \frac{2}{2!} + \frac{3}{3!} + \dots$
 - iv. $x - \frac{x^2}{2!} + \frac{x^3}{3!} - \dots$
6. Accept two integers a and b and display $a*b$, a/b and $a\%b$ without using *, / and % operators.
7. Calculate the GCD and LCM of two integers.
8. Accept characters from the keyboard till the user enters EOF. Count the number of uppercase, lowercase alphabets and vowels in the text.
9. Write a C program to read lines of text and count the number of characters, words and lines in the text.
10. Write a C program to read an integer, reverse it and display both.
11. Write a program to display digits of an integer separated by tabs
Example: $1009 \rightarrow 1 \ 0 \ 0 \ 9$
 $2000 \rightarrow 2 \ 0 \ 0 \ 0$
12. Accept data from the keyboard and check if it is valid or invalid.
13. Accept lines of text from the user and find the length of the longest line.

C. Review Questions

1. What are the different forms of the if statement?
2. Explain the switch-case statement with examples.
3. Differentiate between if-else and switch-case.
4. Explain else-if ladder with an example.
5. Explain the syntax of a while loop.
6. How does a do-while loop differ from a while loop?
7. Explain different ways to terminate loop execution.
8. Explain the for loop with examples.
9. Distinguish between break and continue.
10. Write a note on goto and labels.
11. Illustrate the use of the break statement in the switch –case statement.
12. Discuss the working of if-else and switch statement.



INTRODUCTION TO PROBLEM SOLVING

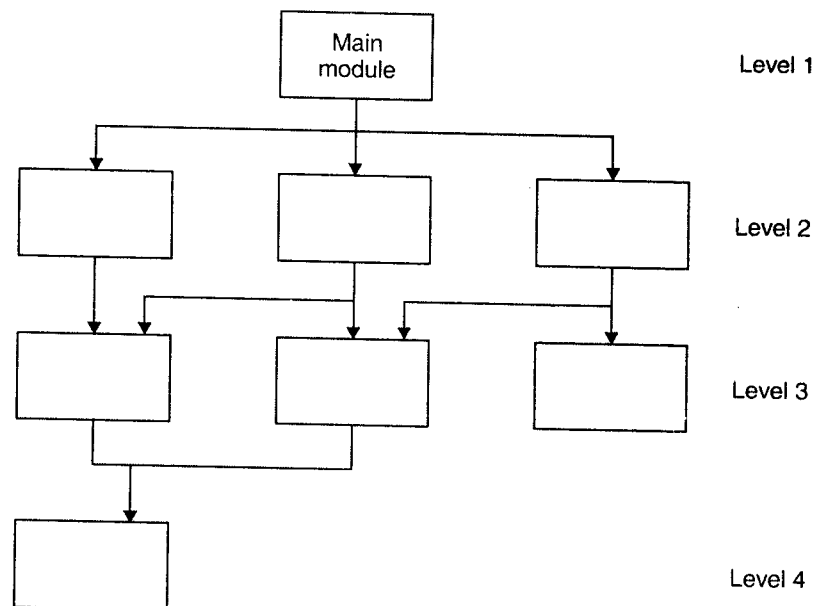
6

6.1 INTRODUCTION

Problem solving is a part of thinking. It is a part of a larger problem process that includes problem finding and problem shaping. George Polya outlined the essence of problem solving:

1. Understand the problem (communication and analysis)
2. Plan a solution (modelling and design)
3. Carry out the plan (code generation)
4. Examine the result for accuracy (testing and quality assurance)

For certain problem the task of defining the problems are much more time consuming and costly than the task of programming them. The modularity on most of the problems can be represented by a hierarchical structure.



The structure has a single main module, with which we associate a level number of 1, which gives the brief general description of the system. The main module refers to a number of subordinate modules which have been numbered as level 2, 3, 4 respectively.

Level 2 gives more detail description of the system, than the main module and so on.

It is possible that modules at upper level refer to the lower one. The concept of hierarchically structuring a problem in this fashion is a fundamental one in the problem solving. It is this form of the organization or structuring which permits us to understand a system at different levels and allow us to make changes at one level. Without having to completely understand more detailed descriptions at higher levels. The important thing that can be done with this is desirability of being able to understand a module at a certain level independently and all remaining modules at that same level.

The task of writing a computer program is made simpler if the problem can be analyzed in terms of sub-problems. In organizing a solution to a problem which is to be solved with the aid of computer, we are confronted with at least four interrelated sub-problems.

The sub-problems are:

1. To understand thoroughly the relationships between the data elements that are relevant to the solution of the problem.
2. To decide on the operations that must be performed on the logically related data elements.
3. To divide the methods of representing the data elements in the memory of the computer such that a) the logical relationships that do exist between data items can best be retained and/or b) the operations on the data elements can be accomplished easily and efficiently.
4. To decide on what problem solving language can best aid in the solution of the problem by allowing the user to express in a natural manner the operations he or she wishes to perform on the data.

6.2 PROBLEM SOLVING TECHNIQUES

There are many approaches to problem solving, depending on the nature of the problem and the people involved in the problem. The more traditional, relational approach is typically used and involves *example*, clarifying the description of the problem, analyzing causes, identifying alternatives, accessing each alternative, choosing one, implementing it, and evaluating whether the problem was solved or not.

Another approach is appreciative inquiry. That approach asserts that "problems" are often the result of our own perspectives on a phenomena, *example* if we look at it

as a "problem," then it will become one and we'll probably get very stuck on the "problem." Appreciative inquiry includes identification of our best times about the situation in the past, wishing and thinking about what worked best then, visioning what we want in the future, and building from our strengths to work toward our vision.

Following are some of the problem solving techniques

A. Trial And Error

In trial and error, one selects a possible answer, applies it to the problem and, if it is not successful, selects (or generates) another possibility that is subsequently tried. The process ends when a possibility yields a solution.

This approach is more successful with simple problems and in games, and is often resorted to when no apparent rule applies. This does not mean that the approach need be careless, for an individual can be methodical in manipulating the variables in an attempt to sort through possibilities that may result in success. Nevertheless, this method is often used by people who have little knowledge in the problem area.

Advantages

1. **Solution-oriented:** Trial and error makes no attempt to discover *why* a solution works, merely that it *is* a solution.
2. **Problem-specific:** Trial and error makes no attempt to generalise a solution to other problems.
3. **Non-optimal:** Trial and error is an attempt to find *a* solution, not *all* solutions, and not the *best* solution.
4. **Needs little knowledge:** Trials and error can proceed where there is little or no knowledge of the subject.

Applications

1. Biological evolution is also a form of trial and error. Random mutations and sexual genetic variations can be viewed as trials and poor reproductive fitness as the error. Thus after a long time 'knowledge' of well-adapted genomes accumulates simply by virtue of them being *able* to reproduce.
2. Bogosort can be viewed as a trial and error approach to sorting a list.
3. In mathematics, the method of trial and error can be used to solve formulae -it is a slower, less precise method than algebra, but is easier to understand.

B. Brain storming

Brainstorming is a group creativity technique designed to generate a large ideas for the solution to a problem. The method was first popularized in the late 1930s by Alex Faickney Osborn. Osborn proposed that groups could double their creative output by using the method of brainstorming.

Although brainstorming has become a popular group technique, researchers have generally failed to find evidence of its effectiveness for enhancing either quantity or quality of ideas generated. Because of such problems as distraction, social loafing, evaluation apprehension, and production blocking, brainstorming groups are little more effective than other types of groups, and they are actually less effective than individuals working independently. For this reason, there have been numerous attempts to improve brainstorming or replace it with more effective variations of the basic technique. Although traditional brainstorming may not increase the productivity of groups, it may still provide benefits, such as enhancing the enjoyment of group work and improving morale. It may also serve as a useful exercise for team building.

There are four basic rules in brainstorming. These are intended to reduce the social inhibitions that occur in groups and therefore stimulate the generation of new ideas. The expected result is a dynamic synergy that will dramatically increase the creativity of the group.

1. Focus on quantity

This rule is a means of enhancing divergent production, aiming to facilitate problem solving through the maxim, *quantity breeds quality*. The assumption is that the greater the number of ideas generated, the greater the chance of producing a radical and effective solution.

2. No criticism

It is often emphasized that in group brainstorming, criticism should be put 'on hold'. Instead of immediately stating what might be wrong with an idea, the participants focus on extending or adding to it, reserving criticism for a later 'critical stage' of the process. By suspending judgment, one creates a supportive atmosphere where participants feel free to generate unusual ideas.

3. Unusual ideas are welcome

To get a good and long list of ideas, unusual ideas are welcomed. They may open new ways of thinking and provide better solutions than regular ideas. They can be generated by looking from another perspective or setting aside assumptions.

4. Combine and improve ideas

Good ideas can be combined to form a single very good idea, as suggested by the slogan "1+1=3". This approach is assumed to lead to better and more complete ideas than merely generating new ideas alone. It is believed to stimulate the building of ideas by a process of association.

C. Divide And Conquer

Divide and Conquer (D&C) is an important technique in problem solving. It recursively breaks down a problem into two or more sub-problems of the same (or related) type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

Advantages

1 Solving difficult problems

Divide and conquer is a powerful tool for solving conceptually difficult problems, such as the classic **Tower of Hanoi** puzzle. Indeed, for many such problems the paradigm offers the only simple solution.

Dividing the problem into sub-problems so that the sub-problems can be combined again is often the major difficulty in designing a new algorithm.

2. Parallelism

Divide and conquer technique is naturally adapted for execution in multi-processor machines, especially shared-memory systems where the communication of data between processors does not need to be planned in advance, because distinct sub-problems can be executed on different processors.

3. **Memory access:** Divide-and-conquer technique naturally tend to make efficient use of memory caches. The reason is that once a sub-problem is small enough, it and all its sub-problems can, in principle, be solved within the cache, without accessing the slower main memory. An algorithm designed to exploit the cache in this way is called cache oblivious, because it does not contain the cache size(s) as an explicit parameter.

6.3 STEPS IN PROBLEM SOLVING

Problem Solving includes three major activities:

1. Define a problem
 2. Analyze problem
 3. Explore solution
1. **Define a problem**

Before beginning work on a house, a builder reviews the blueprints, checks that all permits have been obtained. And surveys the houses foundation. A builder prepares differently for building a skyscraper. Or building a dog house. No matter what the project, the preparation tailored to its needs and done consciously before construction begins.

The first prerequisite you need to fulfill before designing the program model is a clear statement of the problem that the program is suppose to solve. A problem definition defines what the problem is without any reference to the

possible solutions. Its simple statement may be one to two pages, and it should sound like a problem. For example the statement "We can't keep up with orders Gigatron", sounds like a problem and is a good problem definition. Whereas the statement, "we need to optimize our automated data-entry system to keep up with orders for the Gigatron" is a poor problem definition because the term "We need to ..." itself is, in a way explaining what needs to be done. It doesn't sound like a problem; it sounds like a solution. Problem definition comes before requirement analysis, which is more detailed analysis of the problem.

The problem definition should be in user language, and the problem should be described from the user's point of view. It usually should not be stated in technical computer terms.

The penalty for failing to define the problem is that you can waste a lot of time solving the long problem is a double – barreled penalty because you also don't solve the right problem.

Solving a problem without a clear understanding of its components may turn out to be a futile exercise as the solution may not meet the requirement of the user. So a problem statement has to be prepared which explains every minute detail of the problem beyond doubt. This can be best achieved by writing down the problem in clear statements. Better problem definition results in faster, easier and accurate solutions.

2. Analyze Problem

It describe in detail what a problem is supposed to do, and they are the first step toward a solution. The requirements activity is also known as "functional specification". And explicit set of requirements, is important for several reasons. Explicit requirements help to insure that the user rather than the programmer drives the programs functionality. If the requirements are explicit, the user can review them and agree to them. Explicit requirements keep you from guessing what the user wants. Specifying requirements adequately is a key to the programs success.

Essentially, we must look for three main components which are

- i. What is given as input
- ii. What is expected as output, and
- iii. How to arrive at the solution

You are already familiar with the above three items i.e. input (data) , process, and output (information) . Hence while determining program requirements we have discern from the problem statement what exactly constitutes input, what is expected as output and how to processing is to be done. It will not be out of place to mention here that, a given problem or business solution may be solved in a particular way manually , and we may or may not choose to adopt the same processing logic while developing a solution to be computerized.

Lets take an example and understand the above concept of input, process and output.

A Program is required to retrieve motor vehicles registration record from a file upon receipt of request from an operator at a terminal. The operator will supply a vehicle registration number and the program will display the details of its vehicle and its owner. An error message will be displayed if the program is unable to locate the vehicle's record.

Input: Vehicle registration number

Process: Using the registration number search for it, and if found, retrieve the details of the vehicle and its owner's name from the disk.

Output: If retrieval was successful, then allow the details of the vehicle to be displayed on the screen but if unsuccessful, indicate the absence of the vehicle registration on the disk and display a suitable error message.

3. Explore solution

Once the problem is clearly defined an algorithm (another term for processing logic or model) can be developed. This is the most creative part of programming. At this stage, the algorithm may be constructed in the broad terms to help solve the problem. To be useful as a basis for writing a program, the algorithm must:

- Arrive at a correct solution within a finite time.
- Be clear, precise and unambiguous.
- Be in a format which lends itself to an elegant implementation in a programming language.

The important tools in developing a solution and in the preparation of an algorithm are flowcharts and pseudocodes among others. Flowcharts provide a visual and graphical representation of the solution while pseudocodes mean writing the program logic in a simple English – like language. Logic devised using these tools can be written using a programming language. In other words these are the generic tools.

6.4 ALGORITHMS AND FLOWCHARTS

A computer is a machine that manipulates data by using a finite number of unambiguous instructions obediently, uncritically, and without showing any emotions. Take an instance of a major who went to a Post Office with the order "Buy five 50 paise stamps". The servant went with the money to the Post Office and did not turn up for a long time.

The major got worried and went in search of him to the Post Office and found him standing there with the stamps in his hands. When the major angrily asked him what made him stand there, he replied that he was ordered to buy five 50 paise stamps but not ordered to return with them. Computer solving is an intricate process requiring much thought, careful planning, logical precision, persistence and attention.

Definition

Algorithm

In order to carry out a task using computer, a sequence of explicit and unambiguous instructions is known as an algorithm.

An algorithm consist of a set of explicit and unambiguous finite basic steps, when followed for a given set of initial conditions may produce the corresponding output and terminates in a finite time. An algorithm expressed in a programming language is called a **program**.

Flowchart

A flowchart is a pictorial representation of a program. A flowchart is designed to visually represent the flow of execution through a program.

A flowchart captures sequence, selection, and iteration, all the three basic constructs of the program. Flow charts are made up of boxes, each with their own function. The shape of box shows what it is doing.

Arrows between these boxes shows the program flow.

A typical flowchart may have the following kinds of symbols:

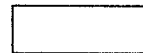
1. **Start** and **end** symbols, represented as lozenges, ovals or rounded rectangles, usually containing the word "Start" or "End", or another phrase signaling the start or end of a flowchart.



2. **Arrows**, showing what's called "flow of control". An arrow coming from one symbol and ending at another symbol represents that control passes to the symbol the arrow points to.



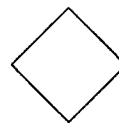
3. **Processing steps**, represented as rectangles. Examples: "Add 1 to X"; "replace identified part"; "save changes" or similar.



4. **Input/Output**, represented as a parallelogram.



5. **Conditional (or decision)**, represented as a diamond (rhombus).



These typically contain a Yes/No question or True/False test. This symbol is unique in that it has two arrows coming out of it, usually from the bottom point and right point, one corresponding to Yes or True, and one corresponding to No or False.

The arrows should always be labeled. More than two arrows can be used, but this is normally a clear indicator that a complex decision is being taken, in which case it may need to be broken-down further, or replaced with the "pre-defined process" symbol.

6.5 CHARACTERISTICS OF AN ALGORITHM

Following are the basic characteristics of an algorithm:

1. **INPUT:** There are no restrictions over the input requirements of the algorithm. Number of inputs of an algorithm may be zero or more than zero.
2. **OUTPUT:** There must be at least one output produced by the algorithm as it accomplishes the given task.
3. **EFFECTIVENESS:** Every instruction used in algorithm should be basic enough or it can be broken into basic instructions so that these instructions can be carried out manually using pen and paper.
4. **DEFINITENESS:** Every instruction of the algorithm should be unambiguous.
5. **FINITENESS:** The algorithm should get terminated in a finite amount of time.

Qualities of a good algorithm

1. They are simple but powerful and general solutions.
2. They can be easily understood by others.
3. They can be easily, modified, if necessary.
4. They are correct for clearly defined solutions.
5. They are economical in the use of computer time, storage, and peripherals.
6. They are well documented.
7. They are machine independent.
8. They are able to be used as a subprogram for other problems.

6.6**CONDITIONALS IN PSEUDOCODE**

Pseudocode (derived from pseudo and code) is a compact and informal high-level description of a computer programming algorithm that uses the structural conventions of some programming language, but typically omits details that are not essential for the understanding of the algorithm, such as subroutines, variable declarations and system-specific code. The purpose of using pseudocode is that it may be easier for humans to read than conventional programming languages, and that it may be a compact and environment-independent description of the key principles of an algorithm.

Flowcharts can be thought of as a graphical alternative to pseudocode. Pseudocode resembles, but should not be confused with, skeleton programs including dummy code, which can be compiled without errors.

As the name suggests, pseudocode generally does not actually obey the syntax rules of any particular language; there is no systematic standard form, although any particular writer will generally borrow the appearance of a particular language. Popular sources include Pascal, BASIC, C, Java, Lisp, and ALGOL. Details not relevant to the algorithm (such as memory management code) are usually omitted. Blocks of code, for example code contained within a loop, may be described in a one-line natural language sentence.

Just like structured programs pseudocode is built on three basic constructs: sequence, selection and looping. And just like Visual Basic program, a program written in pseudocode is divided into functions or procedures. Each function has signature (name, return type and arguments) and a body (a sequence of pseudocode statements).

An alternative method of representing program logic is pseudocode. Instead of using symbols to represent the program logic steps, a pseudocode uses statements which are a bridge between actual programming and ordinary English. In a pseudocode each step is written using a simple English phrase which is also called a construct.

6.7**LOOPS IN PSEUDOCODE**

Some of the conventions which are to be used while writing pseudocodes are as follows :

1. All statements in a loop should be intended.
2. All alphanumeric values should be enclosed in a single or double quotes.
3. The beginning and end of the pseudocode is marked with keywords like 'start' and 'end' respectively.
4. All statements must include certain key words which denote an operation.

The Input Statement

The following verbs can be used to accept or input data from the keyboard or from an exciting from like a file.

Accept or Read

For Example

```
Accept Name  
Read Name
```

The Output Statement

The following verbs can be used to output data

Write or Display

For Example

```
Write Name  
Display Name
```

For Example. A function to return absolute value of an integer might look like this:

```
Absolute_Value(x : Integer) -> Integer  
Begin  
  If (x<0)  
    Return(-x)  
  Else  
    Return(x)  
End -If  
End
```

The function name is `Absolute_Value`, it receives an integer value(x) and returns an integer value as its result. The function body starts with the word `Begin` and ends with the `End` word. Within a body we have a sequence of statements. Each statement within the sequence may be either a simple statement, a selection statement, or an iteration statement.

Simple statements

A simple statement is one of the following:

- *Variable* : Type(declare a new variable of a given type)
- *Variable* := Expression(Assign the value of an expression to a variable)
- Function(Arguments) (Call function, passing in arguments)
- Return(Expression) (return expression as the value of this function)
- Break (break out of the current loop or switch statement)
- These are not only the possibilities for simple statements, but they are the most common.

Selection statements

A selection statement is either an if statement or a case statement

```
If condition1 Then
    Statement1 Body 1
Elseif condition2 Then
    Statement Body 2
... (You can have many else if clause)
Else
    Statement Body N
Endif
```

As shown an if statement tests a condition, or boolean expression (i.e. an expression that evaluates to either True or False) If the condition is true, then the corresponding statement body is executed. But if the condition is false, then the next expression is checked similarly.

This continues in sequence until an expression is found to be true. If all the expressions evaluate to false, then the statement body associated with the else clause is executed. You can have as many elseif clauses as you had like (including none).

The else clause is also optional.

Select Case expression

```
Case value1:
    Statement Body 1
Case Value2 :
    Statement Body 2
...
Default:
    Statement Body N
Endselect
```

A select case statement (also called a case statement or sometimes a switch statement) evaluates an expression and compares it against several values. If the result of the expression is equal to one of the values in the case clauses, then the corresponding statement body is executed.

If the result of the expression does not match any case clause, then the default statement body is executed. After a statement body is executed, the computer executes the next instruction immediately following the Endselect .

Iteration Statements Or Loops

A repetition statement (also called an iteration statement or a loop) is very useful. It causes a block of code to be executed repeatedly. There can be many kinds of loops.

While Do loop

```
While condition do
    Statement /body
EndWhile
```

As long as the condition is true, the statement body is executed. Thus, something in the statement body should modify one of the variables in the condition expression, or else you will be stuck in an infinite loop.

Do Until loop

```
Do  
  Statement Body  
Until condition
```

A do until loop is very similar to the while loop. The main difference is that the while loop tests the condition before executing the statement body, but the do while loop tests the condition after executing the statement body. Thus, the body of a do until loop will always be executed at least once. For instance, suppose the condition is false.

For a while statement, since we test before, we discover that the expression is false and do not execute the body. However, for a do until loop we have already executed the body before we test the condition expression for the first time.

The other difference is that a while loop repeats as long as a condition is true, but a do until loop repeats as long as a condition is false.

6.8 TIME COMPLEXITY

An algorithm is said to be correct if, for every input instance, it halts with the correct output. We say that a correct algorithm solves the given computational problem. An incorrect algorithm might not halt at all some input instance or it might not halt at all on some input instances.

Analyzing an algorithm has come to mean predicting resources that the algorithm requires. Occasionally, resources such as memory, communication bandwidth, or logic gates are of primary concern, but most often it is computational time that we want to measure.

The performance analysis and measurement of an algorithm is based on two criteria:

- **Space Complexity:** It is the amount of memory is needed to run to completion.
- **Time Complexity:** It is the amount of time needed to run to completion.

Time Complexity

The time $T(p)$ taken by a program p is the sum of the compile time and run time. The compile time does not depend on the instance characteristics so we shall concern ourselves with just the time of a program which is denoted by t_p .

$$T_p(n) = C_a\text{ADD}(n) + C_s\text{SUB}(n) + C_m\text{MUL}(n) + C_d\text{DIV}(n) + \dots$$

In computational complexity theory, big O notation is often used to describe how the size of the input data affects an algorithm's usage of computational resources

(usually running time or memory). It is also called Big Oh notation, Landau notation, Bachmann-Landau notation, and asymptotic notation

$f(n) = O(g(n))$ if there exists positive constants c and n_0 such $f(n) < cg(n)$

For all n ; where f and g are non negative functions.

We write $O(1)$ to mean a computing time that is a constant

- $O(n)$ is called linear
- $O(n^2)$ is called quadratic
- $O(n^n)$ is exponential

Big OH notation

Big OH notation is the characterization scheme that allows to measure properties of an algorithm complexity performance and/or memory requirements in a general fashion. The algorithm complexity can be determined ignoring the implementation dependent factors. This is done by eliminating constant factors in the analysis of the algorithm.

Basically, these are the constant factors that differ from computer to computer. Clearly, the complexity function $f(n)$ of an algorithm increases as n increases. It is the rate of increase of $f(n)$ that we want to examine.

Suppose $f(n)$ and $g(n)$ are functions defined on positive integer numbers n , then function $f(n) = O(g(n))$, read as "f of n is big Oh of g of n" or as "f(n) is of the order of g(n)", if there exist positive constants c and n_0 , such that $f(n) = c * g(n)$ for all values of $n = n_0$.

That is, for all sufficiently large $g(n)$. Thus g is upper bound, except for a constant factor c on the value of f for sufficiently large $g(n)$. Thus g is an upper bound, except for a constant factor c on the value of f for all suitably large n i.e., $n \geq n_0$. While providing an upperbound function g for f , we will use only simple functional forms. These typically contains a simple term in n with a multiplicative constant of one.

Categories of Algorithms

Based on Big Oh notation, the algorithms can be categorized as follows :

- Constant time ($O(1)$) algorithms
- Logarithmic time ($O(\log n)$) algorithms
- Linear time ($O(n)$) algorithms
- Polynomial time ($O(n^k)$, for $k > 1$) algorithms
- Exponential time ($O(k^n)$, for $k > 1$) algorithms

Many algorithms are $O(n \log n)$.

Limitations of Big Oh notation

Big Oh has two basic limitations :

- It contains no consideration of programming effort
- It masks potentially important constants

As an example of later limitation, imagine two algorithms, one using $500000n^2$ time, and the other n^3 time. The first algorithm is $O(n^2)$, which implies that it will take less than the other which is $O(n^3)$. However the second algorithm will be faster for $n < 500000$, and this would be faster for many applications.

Basic time analysis of an algorithm

Lets take an example of analysis of time required for the execution of an algorithm Consider the following algorithm to sum the values in vector V that contains N values:

Algorithm SUM_VALUES

Given a vector V containing N elements, this algorithm computes the arithmetic sum (SUM) of these elements. I is a integer variable.

1. [Sum the values in Vector V]
 SUM <- 0
 Repeat for I = 1,2,...N
 SUM <- SUM +V[I]
2. [Finished]
 Exit

Rather than calculating the exact time, we want an estimate of it. Usually this is most easily done by isolating a particular operation, sometimes called an active operation, that is central to the algorithm and that is executed essentially as often as any other. In the above example, a good operation to isolate is the addition that occurs when another vector value is added to the partial some. The other operations in the algorithm, the assignments, the manulation of the index I, and the accessing of a value in the vector, occur no more often then the addition of vector values. These other operations we collectively called book keeping operations and are not generally counted.

It is very important that none of the bookkeeping operations are executed significantly more often than the active operations. After the active operations are isolated , the no of times that it is executed is counted.

The number of additions of values in the above example is N. As long as the active operation occurs at as often as others , then the execution time will increase in proportion to the number of times the active operation is executed. The above algorithm has execution time proportional to N . or expressed another the time required is linearly proportional to the size of the input.

Example : Matrix multiplication of two $N \times N$ matrices A and B to form $N \times N$ matrix C.

Algorithm MATRIX_MULTIPLICATION

Given two dimensional square matrices A and B, each containing N rows and columns, this algorithm computes the matrix product and places the result in matrix C. I, J, K are integer variables.

1. [Multiply matrices A and B and store the result in matrix C]
 - Repeat for I = 1, 2, ...N
 - Repeat for J = 1, 2, ...N
 - SUM ← 0
 - Repeat for K = 1, 2, ...N
 - SUM ← SUM + A[I, K] * B[K, J]
 - C[I, J] ← SUM
2. [Finished]
 - Exit

The actual size of the input for this algorithm is $2N^2$, but it is convenient to use N as our measure of the size of the input in order to simplify the calculations. For the active operations, we can select either the multiplication of A[I, K] AND B[K, J], or the addition of above product to sum. This follows since both are central operations and essentially occurs as any other. It is to see that either of these operations is executed N^3 times, so that the time for the algorithm is proportional to N^3 .

Note that there are actually more assignments than multiplications or additions. There are n assignments to I, N^2 assignments to J and C, N^3 assignments to K, and $N^2 + N^3$ to SUM. This yields a total of $N + 3N^2 + 2N^3$ assignments. Certainly assignments could have been selected as our active operation although it is questionable whether it is as central to the problem as either multiplication or addition.

If it were used as a active operation, we would conclude that the time was proportional to $N + 3N^2 + 2N^3$. Fortunately, we are normally only interested in the order of magnitude of the time required. The order only considers the term that grows fastest, $2N^3$, ignore the constant 2, associated with it. Thus we obtain the order of magnitude for the time required is N^3 , independent of which operation is chosen as active, but that just happens in the problem. In other cases, may be assignments would have to be active.

Thus it is easy to verify the following:

- $100n^3$ is $O(n^3)$
- $6n^2 + 2n + 4$ is $O(n^2)$
- $1 + 2 + 3 + \dots + n = n * (n+1)/2 = n^2 + O(n) = O(n^2)$
- 1024 is $O(1)$

- $n + \log n$ is $O(n)$
- $3n$ is $O(n^2)$ and also $O(n)$

6.9 SIMPLE EXAMPLES: ALGORITHMS AND FLOWCHARTS (REAL LIFE EXAMPLES)

Example

Accept two numbers, add them and display the result.

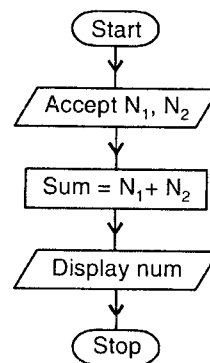
The steps for the above problem statement are

```
START
ACCEPT N1
ACCEPT N2
SUM = N1 + N2
DISPLAY SUM
END
```

The above is the pseudocode for our problem. It may be noted here that whether we input N_1 first or N_2 first is immaterial here.

However the statement $SUM = N_1 + N_2$ cannot come before the two numbers have been input.

Flowchart

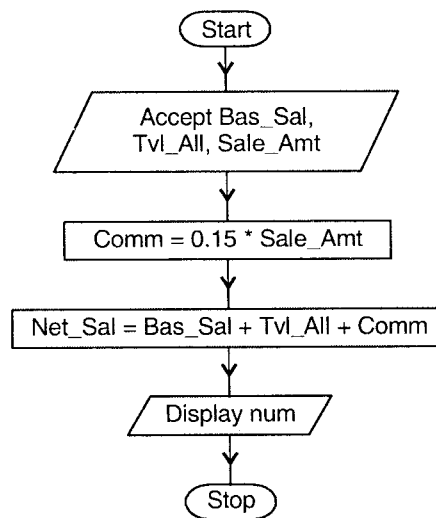


ACCEPT N_1, N_2 denotes that two numbers are accepted from the user and stored in variable N_1 and N_2 . $SUM = N_1 + N_2$ denotes that a process is taking place, which is adding the numbers N_1 and N_2 and the resultant output is stored in the variable SUM . DISPLAY SUM denotes that the resultant SUM is displayed on the screen.

Example: Mohan's monthly salary consists of basic salary, traveling allowances and 15% commission on sales made. At the end of the month we need to calculate his salary which is done in the following pseudocode.

```
START
ACCEPT BAS_SAL, TVL_ALL, SALE_AMT
COMM = SALE_AMT * 0.15
NET_SAL = BAS_SAL + TVL_ALL + COMM
DISPLAY NET_SAL
END
```

It may be noted here that we are accepting 3 variables. BAS_SAL, TVL_ALL, SALE_AMT with one accept statement. This is valid. Alternatively, three ACCEPT statements could have been written one for each of the three variables. Here BAS_SAL, TVL_ALL, NET_SAL are variables which hold the value for basic salary, travelling allowance, sales amount and net salary, respectively.



Example. Maximun of three numbers :

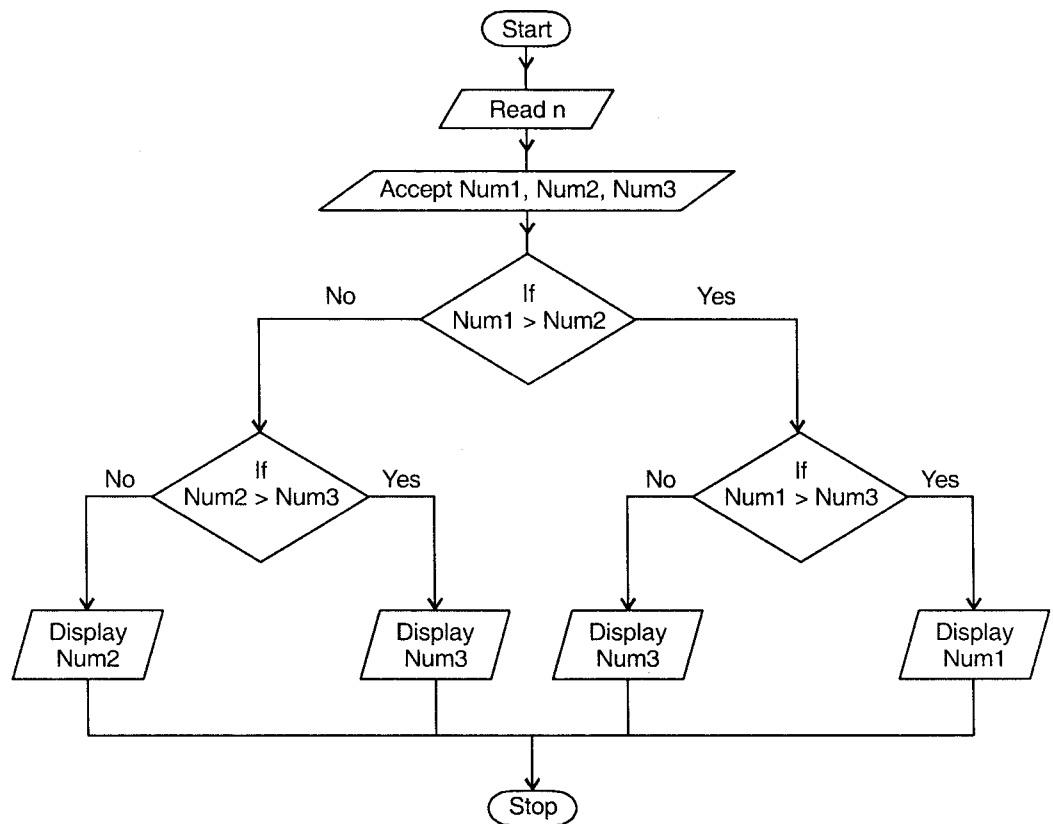
```
START
ACCEPT NUM1, NUM2, NUM3
IF NUM1 IS > NUM2 THEN
    IF NUM1 IS > NUM3 THEN
        DISPLAY "NUM1 IS MAXIMUM"
    ELSE
```

```
    DISPLAY "NUM3 IS MAXIMUM"  
ELSEIF NUM2 IS > NUM3  
    DISPLAY "NUM2 IS MAXIMUM"  
ELSE  
    DISPLAY "NUM3 IS MAXIMUM"  
ENDIF  
END
```

We are accepting three variables here NUM1, NUM2, NUM3. And the algorithm is working to find out the maximum of these three numbers. In the following flowchart it has been shown that how to implement selection statements like if..else in flowchart.

At first we are comparing first two numbers NUM1 and NUM2. If NUM1 is greater than NUM2 we need to compare.

Flowchart:



Exercise

1. What is problem solving?
2. Which are different techniques used for problem solving? Explain in detail
3. Discuss the advantages of Divide and Conquer method.
4. Which are the different steps in problem solving?
5. Give the definition of algorithm and flowchart.
6. Which are the characteristics of the algorithm.
7. What is a time complexity? (Explain along with Big Oh Notation)
8. What is Pseudocode? Explain with example
9. Write an algorithm, flowchart and time complexity for the following:
 - i. Factorial of a given number
 - ii. Addition of two metrics
 - iii. Sorting the given data
 - iv. Check whether the given number is prime or not.



SIMPLE ARITHMETIC PROBLEMS

7

7.1 PROGRAM FOR ADDITION OF TWO INTEGERS



```
#include<stdio.h>
#include<conio.h>
void main()
{
    /*Declaration of variable */
    int Number1,Number2,Sum;
    clrscr();
    printf("\n Enter First Number :"); /* Input First Number */
    scanf("%d",&Number1);
    printf("\n Enter Second Number :"); /* Input Second Number */
    scanf("%d",&Number2);
    Sum=Number1 + Number2;           /* Addition of Two Number */
    printf("\n Addition is : %d",Sum); /* Output of Sum */
    getch();
}
```



Output

```
Enter first number : 23
Enter second number : 17
Addition: 50
```

7.2 PROGRAM FOR MULTIPLICATION OF TWO INTEGERS



```
#include<stdio.h>
#include<conio.h>
void main()
{
    /*Declaration of variable */
    int Number1,Number2,Sum;
    //clrscr();
    printf("\n Enter First Number :"); /* Input First Number */
    scanf("%d",&Number1);
    printf("\n Enter Second Number :"); /* Input Second Number */
    scanf("%d",&Number2);
    Sum=Number1 * Number2;          /* Multiplication of Two Number */
    printf("\n Multiplication is : %d",Sum); /* Output of Sum */
    getch();
}
```



Output

```
Enter first number: 3
Enter second number : 12
Multiplication is: 36
```

7.3 PROGRAM FOR DIVISION OF TWO INTEGERS



```
#include<stdio.h>
#include<conio.h>
void main()
{
    /*Declaration of variable */
    int Number1,Number2,Div;
    clrscr();
    printf("\n Enter First Number :"); /* Input First Number */
    scanf("%d",&Number1);
    printf("\n Enter Second Number :"); /* Input Second Number */
    scanf("%d",&Number2);
    Div=Number1 / Number2;          /* Division of Two Number */
    printf("\n Division is : %d",Div); /* Output of Division */
    getch();
}
```



Output

Enter First Number: 55
Enter Second Number: 5
Division is: 11

7.4 PROGRAM FOR DETERMINING NUMBER IS +VE OR -VE

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int Number;
    clrscr();
    printf("\n Enter Number :");
    scanf("%d",&Number);
    /* Check Number is Negative or Positive */
    if(Number<0)
    {
        printf("\n Number is Negative");
    }
    else
        printf("\n Number is Positive");
    getch();
}
```

**Output**

Enter Number: - 4
Number is Negative

7.5 PROGRAM FOR DETERMINING NUMBER IS ODD OR EVEN

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int Number;
    clrscr();
    /* Get Number */
    printf("\n Enter Number :");
```

```
scanf("%d",&Number);
/* Check Number is Even or Odd */
if((Number%2)==0)
{
    printf("\n Number is Even");
}
else
{
    printf("\n Number is Odd");
}
getch()
}
```



Output

```
Enter Number: 3
Number is Odd
```

7.6 PROGRAM FOR FINDING MAXIMUM OF TWO NUMBERS



```
#include<stdio.h>
#include<conio.h>
void main()
{
    int Number1,Number2;
    //clrscr();
    /*Get Two Number*/
    printf("\n Enter First Number :");
    scanf("%d",&Number1);
    printf("\n Enter Second Number :");
    scanf("%d",&Number2);
    /*Check Maximum Number*/
    if(Number1>=Number2)
    {
        printf("\n First input number is maximum");
    }
    else
    {
        printf("\n Second input number is Maximum");
    }
    getch()
}
```



Output

Enter First Number: 69
Enter Second Number: 4
First input number is maximum

7.7**PROGRAM FOR FINDING MAXIMUM OF THREE NUMBERS**

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int Number1,Number2,Number3;
    //clrscr();
    /*Get Three Numbers */
    printf("\n Enter First Number :");
    scanf("%d",&Number1);
    printf("\n Enter Second Number :");
    scanf("%d",&Number2);
    printf("\n Enter Third Number :");
    scanf("%d",&Number3);
    /*Check Three Numbers For Maximum*/
    if((Number1>=Number2) && (Number1>=Number3))
    {
        printf("\n First input number is Maximum");
    }
    else
    {
        if((Number2>=Number3) && (Number2>=Number3))
        {
            printf("\n Second input number is Maximum");
        }
        else
            printf("\n Third input number is Maximum");
    }
    getch();
}
```

**Output**

Enter First Number: 43
Enter Second Number: 87
Enter Third Number: 12
Second input number is Maximum

7.8 PROGRAM OF SUM OF FIRST N NUMBERS



```
#include<stdio.h>
#include<conio.h>
void main()
{
    int Number,i=0,Sum=0;
    clrscr();
    /* Get Number */
    printf("\n Enter Number :");
    scanf("%d",&Number);
    /* Sum of N Numbers */
    for(i=1;i<=Number;i++)
        Sum+=i;
    printf("\n Sum of N Numbers : %d",Sum);
    getch();
}
```



Output

```
Enter Number: 8
Sum of N Numbers: 36
```

7.9 PROGRAM FOR REVERSING INTEGER NUMBER



```
#include<stdio.h>
#include<conio.h>
void main()
{
    int Number1,Number2;
    clrscr();
    /* Get Number */
    printf("\n Enter Number :");
    scanf("%d",&Number1);
    printf("\n Reverse of Integer :");
    /* Reverse Number */
    while(Number1>0)
    {
        Number2=Number1%10;
        Number1=Number1/10;
        printf("%d",Number2); }
    getch();
}
```



Output

Enter Number: 12345
Reverse of Integer: 54321

7.10 PROGRAM FOR TABLE GENERATION OF N NUMBER

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int Number,i=0,j=0,k;
    clrscr();
    /* Get Number */
    printf("\n Enter Number :");
    scanf("%d",&Number);
    if(Number<=5)
        k=2;
    else
        k=1;
    /* Print Table Here */
    for(i=1;i<=Number*k;i++)
    {
        for(j=1;j<=Number;j++)
        {
            printf("%d\t",j*i);
        }
        printf("\n");
    }
    getch();
}
```

**Output**

Enter Number: 5

| | | | | |
|----|----|----|----|----|
| 1 | 2 | 3 | 4 | 5 |
| 2 | 4 | 6 | 8 | 10 |
| 3 | 6 | 9 | 12 | 15 |
| 4 | 8 | 12 | 16 | 20 |
| 5 | 10 | 15 | 20 | 25 |
| 6 | 12 | 18 | 24 | 30 |
| 7 | 14 | 21 | 28 | 35 |
| 8 | 16 | 24 | 32 | 40 |
| 9 | 18 | 27 | 36 | 45 |
| 10 | 20 | 30 | 40 | 50 |

7.11**PROGRAM FOR FACTORIAL**

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int Number,Factorial=1,i=0;
    clrscr();
    printf("\n Enter Number :");
    scanf("%d",&Number); /* Input Number */
    if(Number<=0)
        Factorial=1; /* Assign 1 to If Number Is Less than 0 */
    else
    {
        for(i=1;i<=Number;i++)
        {
            Factorial*=i; /* Calculate Factorial */
        }
    }
    printf("\n Factorial is : %d",Factorial); /* Print Factorial */
    getch();
}
```

**Output**

```
Enter Number: 5
Factorial is: 120
```

7.12**PROGRAM FOR FINDING SINE OF A NUMBER**

```
#include<stdio.h>
#include<conio.h>
void main()
{
    float Number;
    clrscr();
    printf("\n Enter Number :");
    scanf("%f",&Number); /* Input Number */
    printf("Sine of a Number is : %f",sin(Number));/* Print Sine of
Number */
    getch();
}
```



Output

Enter Number: 2
Cosine of a Number is: 0.3489

7.13 PROGRAM FOR FINDING COSINE OF A NUMBER

```
#include<stdio.h>
#include<conio.h>
void main()
{
    float Number;
    clrscr();
    printf("\n Enter Number :");
    scanf("%f",&Number); /* Input Number */
    printf("Cosine of a Number is : %f",cos(Number)); /* Print
    Cosine of Number */
    getch();
}
```

**Output**

Enter Number: 0
Cosine of a Number is: 1.000000

7.14 PROGRAM FOR COMBINATIONS

```
#include<stdio.h>
#include<conio.h>
float fact(float);
void main()
{
    float n,r,np=0,rf=0,rp=0,nrp=0;
    clrscr();
    printf("\n Enter Distinct Element n: ");
    scanf("%f",&n);
    printf("\nEnter r :");
    scanf("%f",&r);
    np=fact(n); /* Calculate Combinations */
    rf=fact(r);
    rp=fact(n-r);
    nrp=np/(rf*rp);
}
```

```
printf("Number of Combinations is : %f",nrp);/*Print
Combinations */
getch();
}
float fact(float Number)
{
    float Factorial=1,i=0;
    if(Number<=0)
        Factorial=1; /* Assign 1 to If Number Is Less than 0 */
    else
    {
        for(i=1;i<=Number;i++)
        {
            Factorial*=i; /* Calculate Factorial */
        }
    }
    return Factorial;
}
```



Output

```
Enter Distinct Element n: 5
Enter r: 2
Number of Combinations is: 10.000000
```

7.15

PROGRAM FOR PERMUTATION



```
#include<stdio.h>
#include<conio.h>
float fact(float);
void main()
{
    float n,r,np=0,rp=0,nrp=0;
    clrscr();
    printf("\n Enter Distinct Element n: ");
    scanf("%f",&n);
    printf("\n Enter r :");
    scanf("%f",&r);
    np=fact(n); /* Calculate Permutations */
    rp=fact(n-r);
    nrp=np/rp;
    printf("Number Permutation are :%f",nrp);/*Print Permutation */
    getch();
}
float fact(float Number)
{
    float Factorial=1,i=0;
    if(Number<=0)
        Factorial=1; /* Assign 1 to If Number Is Less than 0 */
    else
```

```
{
    for(i=1;i<=Number;i++)
    {
        Factorial*=i; /* Calculate Factorial */
    }
    return Factorial;
}
```



Output

```
Enter Distinct Element n: 5
Enter r: 2
Number Permutation are: 20.000000
```

7.16 PROGRAM FOR PASCAL TRIANGLE



```
#include<stdio.h>
#include<conio.h>
void main()
{
    int i=1,j=1,k=1,l=0;
    clrscr();
    /* Print Pascal Tringle */
    for(i=1;i<=5;i++)
    {
        for(l=5;l>=i;l--)
            printf(" ");
        for(j=1;j<i;j++)
        {
            printf(" %d",j);
        }
        for(k=j;k>=1;k--)
        {
            printf(" %d",k);
        }
        printf("\n\n");
    }
    getch();
}
```



Output

```

          1
        1 2 1
      1 2 3 2 1
    1 2 3 4 3 2 1
  1 2 3 4 5 4 3 2 1
```

7.17 PROGRAM FOR FINDING PRIME NUMBER



```
#include<stdio.h>
#include<conio.h>
void main()
{
    int Number,i,j;
    clrscr();
    /* Get Number */
    printf("\n Enter Number :");
    scanf("%d",&Number);
    /* Check Number is Prime or Not */
    for(i=2;i<Number;i++)
    {
        if((Number%i)==0)
        {
            printf("\n Number is Not Prime : %d", Number);
            getch();
            return;
        }
    }
    printf("\nNumber is Prime : %d",Number);
    getch();
}
```



Output

```
Enter Number: 7
Number is Prime: 7
```

7.18 PROGRAM TO FIND FACTORS OF A NUMBER



```
#include<stdio.h>
#include<conio.h>
void main()
{
    int Number,i,j,k=2;
    clrscr();
    printf("\n Enter Number :");
    scanf("%d",&Number);
    k=Number;
    for(i=2;i<=Number;i++)
```

```
{
    for(j=Number1; j>=2; j--)
    {
        if((i*j)==k)
        {
            printf("%dx%d", i, j);
            break;
        }
    }
    k=j;
}
getch();
}
```



Output

Enter Number: 12
2x6 3x2

7.19

PROGRAM FOR GREATEST COMMON DIVISOR BETWEEN TWO NOS



```
#include<stdio.h>
#include<conio.h>
void main()
{
    int Number1, Number2, i, j;
    clrscr();
    /*Get Two Numbers*/
    printf("\n Enter First Number :");
    scanf("%d", &Number1);
    printf("\n Enter Second Number :");
    scanf("%d", &Number2);
    /*Assign Maximum Number*/
    if(Number1>Number2)
        i=Number1;
    else
        i=Number2;
    for(; i>=1; i--)
    {
        if((Number1%i)==0 && (Number2%i)==0) /* Calculate Factorial*/
            break;
    }
    printf("\n GCD Between Two Numbers :%d", i);
    getch();
}
```



Output

```
Enter First Number: 15
Enter Second Number: 3
GCD Between Two Numbers: 3
```

7.20 PROGRAM FOR SWAPPING OF TWO INTEGERS

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int Number1,Number2,Temp;
    clrscr();
    /*Input Two Numbers*/
    printf("\n Enter First Number :");
    scanf("%d",&Number1);
    printf("\n Enter Second Number :");
    scanf("%d",&Number2);

    /*Swapping of a Number*/
    Temp=Number1;
    Number1=Number2;
    Number2=Temp;
    printf("\n Number 1 is :%d \n Number 2 is :%d",Number1,Number2);
    getch();
}
```

**Output**

```
Enter First Number: 3
Enter Second Number: 6
Number 1 is: 6
Number 2 is: 3
```

Exercises

1. Find the output of the following program:

```
#include<stdio.h>
int a=0; /* This is a global variable */
void foo(void);
int main(void)
{
    int a=2; /* This is a variable local to main */
    int b=3; /* This is a variable local to main */
    printf("1. main_b = %d\n", b);
    printf("main_a = %d\n", a);
    foo();
    printf("2. main_b = %d\n", b);
}
```

```
void foo(void){
    int b=4; /* This is a variable local to foo */
    printf("foo_a = %d\n", a);
    printf("foo_b = %d\n", b);
}
```

2. Study the below program and explain the call to functions and their response

```
#include<stdio.h>
/* Examples of declarations of functions */
void square1(void); /*Example of a function without input
parameters and without return value*/
void square2(int i); /*Example of a function with one input
parameter and without return value */
int square3(void); /*Example of a function without input
parameters and with integer return value */

int square4(int i); /*Example of a function with one input
parameter and with integer return value */

int area(int b, int h); /*Example of a function with two input
parameters and with integer return value */

/* Main program: Using the various functions */
int main (void)
{
    square1(); /* Calling the square1 function */
    square2(7); /* Calling the square2 function using 7 as actual
parameter corresponding to the formal parameter i */
    printf("The value of square3() is %d\n", square3()); /*Using the
square3 function */
    printf("The value of square4(5) is %d\n", square4(5));/*Using
the square4 function with 5 as actual parameter corresponding to
i*/
    printf("The value of area(3,7) is %d\n", area(3,7));/* Using
the area function with 3, 7 as actual parameters corresponding
to b, h respectively */
}
/* Definitions of the functions */
/* Function that reads from standard input an integer and prints
it out together with its sum */
void square1(void)
{
    int x;
    printf("Please enter an integer > ");
    scanf("%d", &x);
    printf("The square of %d is %d\n", x, x*x);
}
/* Function that prints i together with its sum */
void square2(int i)
{
    printf("The square of %d is %d\n", i, i*i);
}
/* Function that reads from standard input an integer and returns
its square */
```

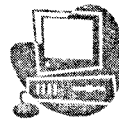


```
int square3(void)
{
    int x;
    printf("Please enter an integer > ");
    scanf("%d", &x);
    return (x*x);
}

/* Function that returns the square of i */
int square4(int i)
{
    return (i*i);
}
/* Function that returns the area of the rectangle with base b
and hight h */
int area(int b, int h)
{
    return (b*h);
}
```

B. Programming exercises

1. Write a program in C to find the roots of equation using quadratic equation formula.
2. Write a program in C to find largest of n numbers.
3. Write a program in C to sort numeric integer n in ascending order.
4. Make use of two dimensional arrays to show addition of two matrices.
5. Use while loop and generate tables 1 to 10.
6. Write a program in C to demonstrate use of nested for.
7. Write a program in C to find proper factors.
8. Write a program in C to accept your name, age and address.



8.1 INTRODUCTION

Functions are the building blocks of C and are central to C programming and to the philosophy of C program design.

`main()` is the function where execution begins. The other functions are executed when they are called directly or indirectly by `main`.

It is mandatory to have a single `main()` function in every program. In the following sections, we shall be studying more about `main` and other functions.

8.2 WHAT IS A FUNCTION?

The program development cycle includes problem analysis, problem definition, design and coding. The code is a set of instructions in a logical sequence, which performs the specified task. 'Real world' applications programs are large and complex. Therefore it is more logical and convenient to break up the task into smaller, compact and more manageable modules, called functions.

Definition

A function is a named, independent or self-contained block of statements that performs a specific, well defined task and may return a value to the calling program.

- A function is named. Each function is identified by a unique name and is invoked (or called) using this name.
- A function is independent. It can perform the task on its own. It can contain its own variables and constants to be used only within the function.
- It performs a specific task: A function is given a discrete job to perform as a part of the overall program. The task has to be well defined.
- It can return a value to the calling program. The function can perform executions and optionally return information to the calling program.

8.3**FUNCTIONS AND STRUCTURED PROGRAMMING**

Functions and structured programming are closely related. In structured programming, independent sections of program code perform program tasks.

Advantages of functions

1. Modular or structured programming can be done by the use of functions.
2. By following the top-down approach, the main function can be kept very small and all the tasks can be designated to various functions.
3. Troubleshooting and debugging becomes easier in structured programs.
4. Individual functions can be easily built and tested.
5. Program development becomes very easy.
6. It is easier to understand the program logic.
7. Multiple functions can be developed and tested simultaneously thereby reducing the program development cycle time.
8. A repetitive task can be put into a function that can be called whenever required. This reduces the size of the program.
9. Frequently used functions can be put together in a customized library.
10. A function can call other functions. It may even call itself. This technique called recursion is very useful in solving complex problems and writing a compact code.

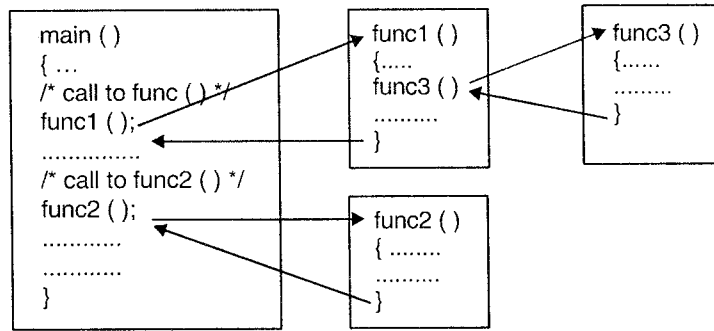
8.4**HOW A FUNCTION WORKS?**

A C program does not execute the statements in a function until the function is invoked or called. When the function is called, control passes to the function and returns back to the calling part after the execution of function is over.

The calling program can send information to the functions in the form of argument.

An argument stores data needed by the function to perform its task. Functions can send back information to the program in the form of a return value.

Function calls and returns can be illustrated by the following example.



main () calls func1 () and func2 (); func1 () calls func3 ()

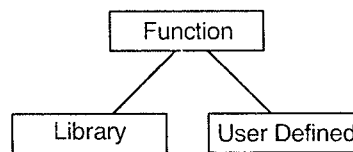
Figure 8.1

Note: A function can be called as many times as needed and can be written and called in any order.

8.5 LIBRARY AND USER DEFINED FUNCTIONS

In a C program, functions are of two types.

1. Pre-defined functions or library functions.
2. User defined functions.



The pre-defined or library functions are pre written, compiled and placed in libraries. They come along with the compiler.

User defined functions are written by the user and the user has the freedom to choose the name, arguments (number and type) and return data type of the function.

One of the greatest features of C is that there is no conceptual difference between the user defined functions and library functions. A user can write functions, collect them and put them into a library, which can be used by anyone.

In this chapter we shall be mainly studying user defined functions.

Standard Library Functions

Some commonly used library functions are given in the table below. We shall be using some of them in the later chapters. To use a library function in a program, its corresponding header file must be included in the program.

1. *stdio.h*

| Function | Prototype | Purpose |
|----------|---|---------------------------------------|
| getchar | int getchar (void) | gets a character from stdin |
| putchar | int putchar (int c) | writes a character to stdout |
| gets | char *gets (char *) | gets a string from stdio |
| puts | int puts (const char *) | outputs a string to stdout |
| printf | int printf(const char* format, [arg, ...]); | writes a character to stdout |
| scanf | int scanf (const char * format,[address, ...]); | scans and formats an input from stdin |
| sprintf | int sprintf (char* buffer, char * format, [argument , ...]); | writes formatted output to a string |
| sscanf | int sscanf (const char * buffer, const char * format , [address, ...]); | scans and formats input from a string |
| fflush | int fflush (file *); | flushes a stream |

2. *math.h*

| Function | Prototype | Purpose |
|----------|---------------------------------|---------------------------------------|
| abs | int abs (int x) | Returns the absolute value of x |
| cos | double cos (double x) | Returns cosine of x (x is in radians) |
| exp | double exp (double x) | Calculates e^x |
| floor | double floor (double x) | Returns the largest integer $\leq x$ |
| log | double log (double x) | Returns natural log of x |
| pow | double pow (double x, double y) | Calculates x^y |
| sin | double sin (double x) | Calculated sine of x |
| sqrt | double sqrt(double x) | Calculates square root of x |

3. *conio.h*

| Function | Prototype | Purpose |
|----------|--------------------|--|
| clrscr | void clrscr (void) | Clears the text mode window |
| cleof | void cleof (void) | Clears to end of line in text window |
| getch | int getch(void) | Gets a character from console. No echoing |
| getche | int getche(void) | Same as getch but echoes to screen. No buffering is done |
| kbhit | int kbhit(void) | Returns an integer corresponding to a keystroke |
| putch | int putch (int ch) | Outputs a character to the text window on screen |

4. **stdlib.h**

| Function | Prototype | Purpose |
|-----------|-----------------------------------|--|
| atof | double atof (const char *s) | Converts a string to float |
| atoi | double atoi (const char *s) | Converts a string to int |
| atol | double atol (const char *s) | Converts a string to long |
| random | int random (int num) | Returns an integer between 0 and (num-1) |
| randomize | void randomize (void) | Initialize the random number generator with a random value |
| system | int system (const char * command) | Used to execute an MS-DOS command |

8.6 FUNCTION DECLARATION AND DEFINITION

Just as variables used within a program have to be declared, so as the functions. The function declaration is called the **function prototype** and it provides the following information to the compiler.

- The name of the function.
- The return data type (optional, default is integer).
- The number and type of arguments that will be passed to the function. (The argument names need not be specified.).

A prototype should always end with a semicolon.

Syntax:

```
return-type function_name(type arg1, type arg2 ...);
```

Examples:

- i. int sum(int a, int b, int c); OR int sum(int, int, int);
- ii. void display(void);
- iii. double square(double number);

Function definition

The function definition is the actual function. The definition contains the code that will be executed. The first line of the definition called the **function header** should be identical to the function prototype with the exception of the semicolon. The argument names have to be specified here. More about this, in the next section.

8.7**WRITING A FUNCTION**

Each function definition has the following form.

```
Return_type function_name(parameter list)
{
  declarations;
  statements;
}
```

The function header

The first line of every function is the function header, which has three components.

a. The function return type

This specifies the data type that the function returns to the calling program. If the function does not return a value, the return data type of void is used.

Examples:

```
int func1 (....) /* Returns an integer value */
float func2 (....) /* Returns a type float */
void func3 (....) /* Returns nothing */
```

b. The function name

The function name can be any valid C identifier. The function name has to be unique and it should be preferably named so as to reflect the purpose of the function

c. The parameter list

Function parameters are the means of communication between the calling and the called functions. They can be classified as:

- Formal parameters (or parameters), which are given in the function header.
- Actual parameters (or arguments) which are specified in the function call.

Each function has to declare the type and name of the parameter. Commas separate multiple parameters. For each argument passed in the function call there has to be corresponding parameter in the parameter list in the function headers with the same data type and the order in which arguments are sent. *Examples as follows:*

```
i. main()
   { int x,y, result;
     result = sum(x,y);} /* function call */
int sum(int a, int b) /* function definition */
{return a + b};
```

In this example, sum is a function accepting two integers and returning an integer. x and y are the actual parameters. a and b are the formal or dummy parameters.

```
ii. float area(float radius)
```

area is a function returning a float and accepts one float argument.

iii. `int max(int a, int b, int c)`

max is a function accepting three integers and returning an integer.

iv. `int random(void)`

This function returns an integer but takes no arguments.

The function body

The function body is enclosed in braces and immediately follows the function header. It consists of,

- a. **Declarations:** You can declare and initialize variables within a function. These are called local variables, which means that they can be used only within that function.

Example:

```
float area(float radius)
{ float result;
  const float pi = 3.142;
  ..... /* function code */
  ..... }
```

- b. **Function statements:** These statements perform the specified task. There is no limitation on the statements that can be included within a function.

However, another function cannot be defined in a user-defined function.

- c. **The return statement:** The keyword **return** is used to terminate the execution of the function and return program control to the calling program.

Syntax:

```
return;
```

Example:

```
if (n < 0)
  return;
```

It is also used to return a value to the calling program. (A function can accept any number of values but can send back only one)

Syntax:

```
return (expression);
OR
return expression;
```

Example:

```
return(0);
return(a+b);
return ++i;
```

A return statement at the end is optional for functions not returning a value. There may be multiple return statements within a function but only the first return statement encountered during control flow will be executed.

Example:

```
int max(int a, int b)
{
    if (a>b)
        return a;
    else
        return b;
}
```



CALLING A FUNCTION

A function can be called by two ways:

1. Any function can be called by simply using its name and arguments alone in a statement as shown. If the function has a return value, it is discarded.

Example:

```
disp_message( );
display_value(x);
```

2. The second method can be used only with functions that return a value. Since they return a value, they can be used anywhere a C expression can be used: in a printf statement, on the right side of an assignment operators, etc. Here are some examples.

```
i.   printf("Square of %d is % d",x, square(x));
ii.  area = calculate_area(radius);
iii. Sum_of_all = sum(a,b) + sum(c,d);
iv.  if (sum(a,b)>100)
    {
        /* statements */
    }
v.   maximum = max(a,b);
vi.  max_of_three = max(C, max(a,b));
```

Types of functions

1. Functions with no arguments and no return values

These functions do not take any information from the calling function nor do they pass back any value. Such functions are commonly used to display messages.

Example

1.

```
#include<stdio.h>
main( )
{ void greet(void);      /* function prototype */
  greet( );             /* function call */
}
void greet(void)        /* function definition */
{ printf("\n Hello and welcome to C");
```

```

2.
#include<stdio.h>
main( )
{ int n;
  void error_msg(void);
  printf("Enter the value of n :");
  scanf ("%d",&n);
  if (n<0)
  { error_msg( );
    exit( );
  }
  .....
  .....
}
void error_msg(void)
{ printf("Error ! Negative value");
}

```

b. Functions with arguments and no return value

Here, the function accepts arguments but does not return any value back to the calling program. It is a one way communication i.e calling program to function.

In such functions, the result of operations on the arguments may be displayed from the function itself.

Example

Demonstrate functions



/* Calculate and display the area of a circle */

```

#include<stdio.h>
main( )
{ float radius;
  void area(float); /* function prototype */
  printf("Enter the radius :")
  scanf("%f" ,&radius);
  area(radius);
}
void area(float r)
{ float result ;
  const float pi = 3.142;
  result = pi*r*r;
  printf("The area is %f", result);
}


```



c. Function accepting arguments and returning a value

Such a function accepts information and also returns back a value to the calling program. Thus, there is a two way communication between the two.

Example: We shall modify the above program such that the function area now returns the calculated value back to main.



```
/* Illustrate function returning a value */
#include<stdio.h>
main( )
{ float radius, a;
  float area(float);
  printf("Enter the radius :");
  scanf("%f", &radius);
  a = area(radius);
  printf("\n The area is %f", a);
}
float area(float r)
{ const float pi = 3.142;
  return(pi*r*r);
}
```




8.9 PASSING ARGUMENTS TO A FUNCTION

There are two mechanisms to pass arguments to a function.

1. Call by value
2. Call by reference

In C all function arguments are "passed by value". This method copies the value of an argument into the formal parameter of the function. Changes made to the formal parameters have no effect on the arguments in the calling function.

The following example illustrates this concept.



```
/* Program to demonstrate call by value */

#include<stdio.h>
main( )
{ int num = 10;
  void modify(int);
  printf("The value of num is %d",num);
  modify(num);
  printf("\n In main the modified value is %d" num);
}

void modify(int num)
{ num = 20;
  printf("\n In the function num is % d", num);
}
```




Output


```
The value of num is 10
In the function num is 20
In main the modified values is 10
```

In the above program, the variable num has a value 10. When the function is called, this value gets copied into the variable num which exists only in the function modify. So, even if its value is changed it does not affect the variable in main. As soon as the function modify ends, the variable num in this function ceases to exist. Back in main, the variable num still retains its original value.

Example:

Here is another program demonstrating call by value. The aim is to interchange the values of two numbers.

```
 /* Illustrate call by value */
#include<stdio.h>
main( )
{ int a=10, b=20;
  void swap(int,int);
  printf("Before interchange a=%d,b=%d", a,b);
  swap(a,b);
  printf("\n After interchange a=%d b=%d" a,b);
}
void swap(int x,int y)
{ int temp;
  temp=x; x=y; y=temp;
  printf("\n In the function x=%d y=%d", x,y);
}
```



Output

```
Before interchange a=10    b=20
In the function    x=20    y=10
After interchange  a=10    b=20
```

In this program, the values of a and b get copied into variables x and y respectively. The function swap interchanges the values of x and y but the values of a and b remain unchanged.

The function can access only the variable value but not the original copy of the variable. Thus, it cannot modify the original variable. An exception to this rule is when an array is passed to a function. Arrays will be covered in details in the next chapter.

Advantage: Passing by value is the default method to protect data from inadvertent modification.

Call by reference

In this method of passing arguments, the called function has access to the original argument, not the local copy. Languages like Pascal and Fortran allow this method.

Although the C language allows passing of arguments only by value the call by reference method can be simulated by the use of addresses and pointers. This allows the function to directly access the original variables and modify their values.

We will rewrite the program to interchange two numbers but in a slightly different way.



```
/* Creating a call by reference to swap two numbers */
#include<stdio.h>
main()
{ int a= 10 , b = 20;
  void swap(int *x , int *y);
  printf("Before swapping a = %d b = %d", a,b);
  swap(&a, &b);
  printf("\n After swapping a = %d b = %d", a,b);
}
void swap(int *x, int *y)
{ int temp;
  temp = *x; *x=*y; *y=temp;
}
```



In the above program, the values of variables a and b have to be interchanged by the function swap. This will only be possible if the function has an access to the original variables. Since it is not possible by the call by value method seen earlier, some other method has to be used.

Since the addresses (memory location) of the variables are unique, if the function is given the address of the variable instead of its value, the function will be directly referring to the original variable. Thus, the addresses of variables a and b are passed using the & (address) operator.

The addresses need to be stored in special variables called as pointer variables, declared as int *x and int *y. The *operator is used to access the variable whose address is stored in its operand. Thus, *x refers to variable a and *y refers to variable b. Thus, by altering *x and *y, we are altering the values of a and b respectively.

8.10 FUNCTIONS WITH VARIABLE ARGUMENTS

It is possible to declare functions with variable numbers of arguments. Such functions are called "Variable" functions. Some standard library functions can accept a variable list of arguments (such as printf).

A function is also defined as variable using an ellipsis ('...') in the argument list. The function is called by passing fixed arguments followed by the additional variable arguments.

Example

```
int func1(intx, ...)
{
}
```

Here, func1 is a function with one fixed argument and the ellipsis indicates variable arguments.

Accessing variable arguments

Since variable arguments have no names, they must be accessed sequentially using special macros from "stdarg.h". These macros are:

- i. va_list
- ii. va_start
- iii. va_end

Example

```
int addnos(int count,...)
{ va_list ab;
  int i, sum;
  va_start(ab, count);      /* Initialize the argument list */
  sum = 0 ;
  for (i = 0; i < count i++)
    sum = + va_arg(ab,int); /* Get next argument*/
  va_end(ab);               /* clean up*/
  return sum;
}
main( )
{ printf("%d\n", addnos(3,5,5,6));
  /* This prints 16 */
  printf("% d\n",addnos(5,10,20,30,40,50)); /* This prints 150*/
```

8.11

COMMAND LINE ARGUMENTS

So far we have been using main with an empty pair of parentheses. In environments that support C, there is a way to pass arguments or parameters to main when it begins executing i.e. at runtime.

These arguments are called command line arguments because they are passed from the command line during run time.

main is called with two arguments

- i. int argc - argument count which is the number of command -line arguments the program was called or invoked with.
- ii. char * argv[] - Argument vector. It is an array of pointers each pointing to a command line argument.

Declaration of main

When main has to accept command line arguments, it has to be declared differently. It is declared as

```
main(int argc, char *argv[ ])
{
  _____
  _____
  _____
}
```

- The subscripts for argv[] are 0 to argc-1.
- argv[0] is the name of the program.
- It is not necessary to use the words argc and argv and any others will also do. However, they are used conventionally, so it is better to stick to them.
- The arguments have to be separated by white spaces. If a space is to be given as a part of an argument, the argument along with the spaces can be specified in double quotes.

Example:

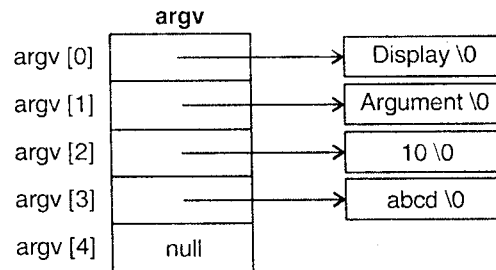
A simple program is the program Display which echoes its command line arguments on the screen. If the command is given as

Display argument1 10 abcd

The output should be

argument1 10 abcd

For this *example* argc = 4 and the arguments will be stored as:



The program will be:

Examples

1. /* Displays command line arguments */

```
#include<stdio.h>
main(int argc, char *argv[])
{
    int i;
    for (i = 1; i< argc ; i++)
        printf("%s%s",argv[i], " ");
}
```

2. /* This displays all the command line arguments in the reverse order*/

```
#include <stdio.h>
main(int argc, char *argv [ ])
{
    while ( -- argc >= 0)
        printf("%s %s", argv[argc], " ");
}
```

Advantages of command line arguments

- i. Arguments can be supplied during runtime. Therefore the program can accept different arguments at different times.
- ii. There is no need to change the source code to work with different inputs to the program.

Example: If a program is to be written without using command line arguments for copying the contents of one file to another, both filenames will have to be specified in the program.

By using command line arguments, the program can be run with different file names every time since the code in the program will refer to them using `argv[]`

- iii. There's no need to recompile the program since the source code is not changed.

We shall be studying more about command line arguments.

8.12 RECURSION

Recursion is a process by which a function calls itself either directly or indirectly. It is called circular definition. Direct recursion is when a statement in the body of the function calls itself. Indirect recursion occurs when the function calls another function, which in turn makes a call to the first one. They are commonly used in applications in which the solution to a problem can be expressed in terms of successively applying the same solution to subset of the problem. Two important conditions should be satisfied by any recursive function.

- Each time the function is called recursively it must be closer to the solution.
- There must be some terminating condition, which will stop recursion.

There are many examples of recursion. One of the most common example is the calculation of the factorial of a numbers. The factorial can be stated as :

1. The factorial of 0 is 1 and the factorial of any positive integer is the product of all integers from 1 to n.
2. The factorial of 0 is 1 and the factorial of any positive integer n is the product of n and the factorial of number n-1.

The first definition is iterative while the second is recursive and represented as



```

/* Using a recursive function to calculate factorial */
#include<stdio.h>
main( )
{ unsigned int num;
  unsigned int factorial(int n);
  printf("\n Enter the value of the number:");
  scanf("%d", &num);
  printf("\n The factorial of %d is %u" ,num, factorial(num));
}
unsigned int factorial(unsigned int n)
{
  if (n ==0 || n ==1)
    return(1);
  else
    return (n* factorial(n-1))
}

```



Output

```

Enter the value of the number: 3
The factorial of 3 is 6.

```

The function calls are depicted below:

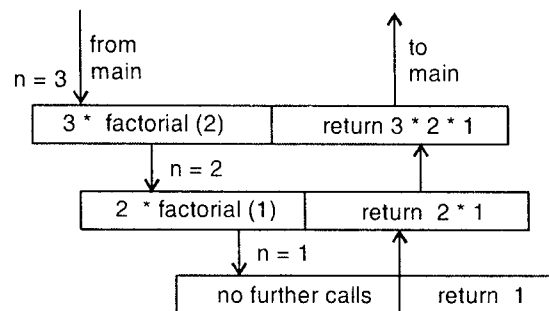


Figure 8.2

$$\begin{aligned}
 \text{i.e. } 3! &= 3 * \text{factorial}(2) \\
 &= 3 * 2 * \text{factorial}(1) \\
 &= 3 * 2 * 1 \\
 &= 6
 \end{aligned}$$

Disadvantage

- Recursive functions may not provide saving in storage since a stack of values is being processed has to be maintained by the system.

It will not be faster than iterative functions because function calls and returns take longer.

Advantage

- However, recursive code is much more compact and often much easier to write and understand than the non-recursive equivalent.

More examples of recursion

1. Computation of Fibonacci series

0, 1, 1, 2, 3, 5, 8,

Each element in this sequence is the sum of the two preceding elements. The series can be defined by the relations.

$$\text{fib}(n) = n \quad \text{if } n == 0 \quad \text{or } n == 1$$
$$\text{fib}(n) = \text{fib}(n-2) + \text{fib}(n-1) \quad \text{if } n >= 2.$$

The following program displays the first 'n' fibonacci numbers using a recursive function to calculate the nth fibonacci number.



```
/* Fibonacci series */
```

```
#include<stdio.h>
main()
{ int num, i;
  unsigned int fib(int);          /* function prototype */
  printf("How many numbers: ");
  scanf("%d",&num );
  printf("\n The first %d, fibonacci numbers are : \n" num);

  /* display the n numbers */
  for (i=0; i<num, i++)
    printf("%u\t", fib(i));
}
unsigned int fib(int n)
{ if(n<=1)
  return (n);
  return (fib(n-2) + fib (n-1));
}
```



Output

How many numbers : 5
 The first 5 fibonacci numbers are :
 0 1 1 2 3

The recursion tree in the calculation of the fifth fibonacci number is:

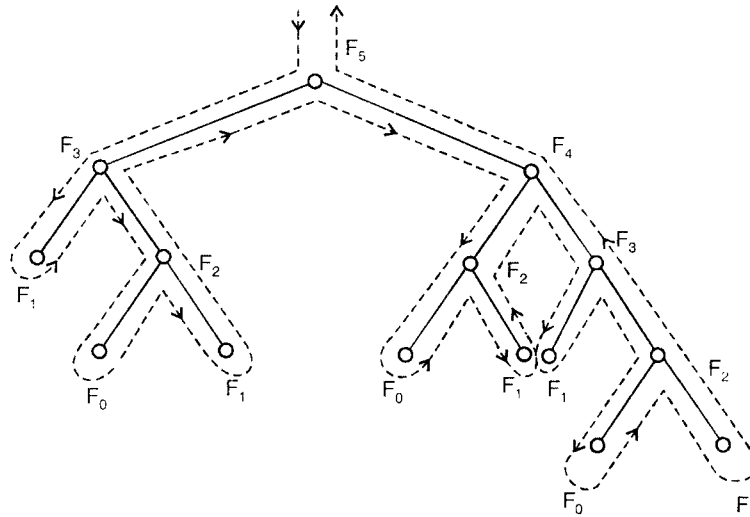


Figure 8.3 : Recursion Tree

- The recursive relation can define calculation of Greatest Common Divisor (GCD) of two positive integers.

$$\text{gcd}(x,y) = x \quad \text{if } y == 0$$

$$\text{gcd}(x,y) = \text{gcd}(y, x \% y) \quad \text{otherwise}$$

The recursive function can be written as:

```
int gcd(int x, int y)
{
    if (y==0)
        return (x)
    else
        return (gcd(y, x%y));
}
```

and it can be used in main as



```

/* Calculation of GCD using above function */
#include<math.h>
#include<stdio.h>
main( )
{ int a, b;
  printf("Enter two numbers:");
  scanf("%d %d", &a &b);
  a = abs(a); /* if a is negative, convert it to positive */
  b = abs(b);
  printf("\n The gcd of %d and %d is %d", a,b,gcd(a,b));
}

```

**Output**

```

Enter two numbers : 25 20
The gcd of 25 and 20 is 5

```

Note : abs is a function which returns the absolute value of its argument.

8.13**FUNCTION RETURNING A POINTER**

A function can return a pointer to the calling function. The function header has to be declared as

```
pointer _ datatype * function_name(parameter list)
```

Example:

i. `int *f1(int);`

f1 is a function accepting an integer and returning pointer to an integer.

ii. `char *f2 (int *, int *);`

f2 is a function returning a pointer to datatype char and accepting the addresses of two integer arguments in two integer pointers.

Examples

1. /* This program accepts the addresses of two integer variables and returns the address of the larger variable to main */



```

#include<stdio.h>
main( )
{
  int *larger(int *, int*); /* prototype */
  int n1, n2,*max;
  printf("Enter the two numbers : ");
  scanf("%d %d", &n1,&n2);
  max = larger(&n1, &n2);
  printf("\n The larger value is %d", *max);
}

```

```
}
int *larger(int *ptrn1,int *ptrn2)
{
    if (*ptrn1 > * ptrn2)
        return (ptrn1);
    else
        return (ptrn2);
}
```

**Output**

Enter the two numbers: 10 20
The larger value is 20.

Exercises

A. Predict the output.

1.

```
main()
{ int i;
  for (i = 1; i<=5; i++)
  { printf("%d",i);
    main( );
  }
}
```

2.

```
main( )
{ int a = 10, b = 15;
  change(a, &b);
  printf("%d%d", a,b);
}
change (int x, int *y)
{   x = 20;
    *y= 30;
}
```

3.

```
main( )
{ int i = abc(100) == 10;
  printf("%d",i);
}
abc(int n)
{ return (n/10);
}
```

4.

```
main( )
{ abc(100,200)
}
abc(int n)
{ printf("%d",n);
}
```

5.

```
main()
{ int i = 5, j = 10;
  abc(i,j);
  printf("i = %d", i);
  printf("\n j = %d", j);
}
abc(int i, int j)
{ i = i+j;
  j = i-j;
  i = i-j;
}
```

B. Programming Exercises

1. Write a function to calculate the roots of a quadratic equation.
2. Write a function that takes two integer parameters and returns the sum of all integers between them.
3. Write a function power which accepts two integers x and y and returns x^y .
4. Write a function ctoi which accepts a character and returns its integer equivalent if it is a digit and returns -1 otherwise.
Example: ctoi(ch) should return integer 5 if ch has value '5'.
5. Write a recursive function to calculate and return the sum of digits of a number.
Example: Sum of digits of 397 = 19.
6. Modify the above function such that the sum of digits is a single digit number.
Example: Sum of digits of 397 = 1
7. Write a recursive program to find the multiplication of two integers.

C. Review Questions

1. Define a function and illustrates how it works.
2. What are the advantages of using functions?
3. What are library and user defined functions?
4. What do you mean by a function prototype?
5. State the different parts of a function? Explain the function header.
6. What are formal and actual parameters?
7. Illustrate with an example function declaration, function definition and function call.
8. What is a local variable? Explain using examples.
9. Explain call by value and call by reference.
10. What is recursion? Explain with examples.
11. What is the meaning of the following declarations?
 - a. `int f(float, char);`
 - b. `void g(int, int , int);`
 - c. `double h(void);`



9.1 MEANING OF TERMS

Every variable in a program has some memory associated with it. Memory for variables is allocated and released at different points in the program.

The **scope** of a variable can be defined as the region or part of the program in which the variable is visible or valid. Visible here also means accessible.

When speaking about scope, the term variable refers to all C data types: Simple variables, arrays, structures, pointers, symbolic constants, etc.

Scope also affects a variables **extent** or **lifetime**.

Extent: This is the period of time during which memory is associated with a variable. In other words, a variable lifetime is how long the variable persists in memory.

Storage class refers to the manner in which memory is allocated by the compiler to variables.

The storage class determines the scope and the lifetime of a variable.

Storage classes are:

- auto
- static
- extern
- register

We have written a number of programs so far and have not used any of these classes as yet.

The reason that the previous programs compile and run is that if no class is mentioned, a default storage class will be assigned depending upon the context in which the variable is used.

9.2 SCOPE

A demonstration of Scope

Examples



/* Illustration variable scope */

```
#include<stdio.h>
main( )
{ int n = 5;
  void display(void);      /* function prototype */
  printf("\n %d",n);
  display( );
}
void display(void)
{
  printf("%d\n",n);
}
```



Output

Compiler error: The variable n is defined within main and is visible only in function main. It cannot be accessed in the function display.

We will now make a small modification to the above program.

2.



/* Illustration variable scope */

```
#include<stdio.h>
int n = 5;
main( )
{ void display (void);    /* function prototype */
  printf("\n %d",n);
  display ( );
}
void display (void)
{
  printf("\n%d",n);
}
```



Output

5
5

We have made a minor modification in the first program by moving the definition of `n` outside `main ()`. By doing so, we have changed its scope.

In Program 1, `n` is a local variable i.e. its scope is limited to the block where it is defined.

In Program 2, `n` is a global (external) variable and its scope is the entire program.

9.2.1 Block Scope and File Scope

The scope of an identifier falls under two categories

1. Block scope (or local scope)
2. File scope

Block Scope: An identifier is said to have local or block scope if it is defined within a function or a block. It can be used only within that function or block. It cannot be used outside. Such identifiers are called **local identifiers**.

File Scope: If an identifier is defined outside a function it can be used in any function in the program i.e. it has a visibility over the entire file. Such identifiers are called **global identifiers**.

Examples

```
/* Local and file scope */
#include<stdio.h>
int n = 20;
main( )
{ int m = 10;
  disp_values( ) }
void disp_values( )
{ printf("%d %d", m,n);
}
```

In this program, variable `n` has **file scope** whereas `m` has **block scope**. `n` can be used in any function in the file whereas `m` can only be used in function `main` because it has been defined in `main`.

Advantages of Block Scope

1. Data integrity is preserved since a function cannot access the data of another.
2. Only the necessary data can be passed to a function thus protecting the remaining data.

Advantages of File Scope

1. If some common data is needed by all functions, passing it as parameters will not be feasible. Making it global will be much easier.
2. Any changes made to the global data by a function can be seen and used by other functions.

Disadvantages of File Scope

1. If too many variables are made global, they will remain in memory till program execution is over. Thus, memory will remain allocated even when they are not being used.
2. Any function can modify global data. Hence data cannot be protected.

9.3 STORAGE CLASSES

The storage class of a variable determines

- i. where it is stored,
- ii. its default initial value,
- iii. scope of the variable,
- iv. lifetime of the variable,

We shall now study the four storage classes

9.3.1 Automatic Storage Class

This is the default storage class of variables that are declared within a function. All the variables that we have studied in previous chapters belong to this class.

In order to explicitly declare a variable which belongs to this class, the keyword `auto` is used.

Example:

```
auto int i;
```

This variable comes into existence only when the function (where it is defined) is called and ceases to exist after the function is exited; hence termed automatic.

Features

1. Storage - Memory
2. Scope - Local to the block where it is defined. (Block scope)
3. Lifetime - It exists as long as control remains in the block where it is defined.
4. Default initial value - Garbage.



```
/*Illustrate automatic variables*/
#include<stdio.h>
main( )
{ auto int i = 10;
  {
    auto int i = 20;
    printf("%d\n",i);
  }
  printf("%d\n",i);
}
```



Output

```
20
10
```

In this program, the two variables *i* are different variables since they are defined in different blocks.

9.3.2 Extern Storage Class

Variables belonging to this class are also called as global variables or external variables. They are declared outside all functions and are accessible to all the functions in that source code file.

The variable *n* is a global variable, *n* is declared outside `main()` which makes it accessible to all the functions in that file.

In some cases however, the program code may extend over two or more separate files. In such a case, special handling is required for external variables.

Use of extern keyword

If the function uses an external variable, it is a good programming practice to declare it again within the function using the `extern` keyword.

Syntax :

```
extern data_type var;
```

Example:

Program 2 in section 9.2 with changes will now be:



```
/* Illustrates external variables */
#include<stdio.h>
int n = 5 ; /* definition */
main()
{
extern int n; /* declaration */
void display(void);
printf("\n %d",n);
display( );}
void display(void)
{
extern int n ; /* declaration */
printf("\n%d",n);
}
```



Note:

- i. The declaration within the function indicates that the function uses an external variable, which is defined elsewhere.
- ii. If both these functions are in the same source code file, the declarations are not required.
- iii. If the variable n is to be used in functions written in separate source code files, the declaration using the extern keyword is required.

Features

1. Storage - Memory
2. Scope - File scope
3. Lifetime - It exists as long as the program which uses the variable is running. It retains its value between functions.
4. Default - Initial value zero

Uses of global variables

- i. Use of global variable simplifies communication i.e. they need not be passed to functions, (thereby making argument lists shorter) and any function can use them whenever required.
- ii. Symbolic constants are often declared globally.

Disadvantages

- i. By using external variables, the principles of modular programming i.e. data isolation is violated.
- ii. Even when not required, external variables persist in memory.
- iii. Variables can be changed in unexpected and inadvertent ways and it is difficult to keep track of the changes made thereby leading to problems.

9.3.3 Static Storage Class

Local variables are automatic by default, which means that every time the function in which they are declared is called, they are created and destroyed when the function ends. They do not retain their value between functions calls.

However, in many cases it is required that a variable retains its value between function calls. This is possible if the variable is declared belonging to the static storage class.

Syntax:

```
static data-type variable;
```

Example:

```
static int x;  
static long factorial;
```

Types of static variables**1. Local static variables**

These variables have block or function scope and they retain their value between calls to the function.

2. Global static variables

They are global to the file in which they are defined. Unlike an ordinary external variable, which is visible to all functions in the file and functions in other files, a static external variable is visible only to functions in its own file.



/* Illustration of local static variable and automatic variable */

```
#include<stdio.h>
main()
{ int n :
  void increment(void);
  for (n=1;n<=5; n++)
    increment( );
}
void increment(void)
{
  int lcount = 0 ;          /* automatic variable */
  static int scount = 0 ;  /* static variable */
  lcount++;
  scount++;
  printf("\n lcount = %d scount = %d", lcount, scount);
}
```



Output

```
lcount = 1  scount = 1
lcount = 1  scount = 2
lcount = 1  scount = 3
lcount = 1  scount = 4
lcount = 1  scount = 5
```

The result shows that every time function increment is called lcount is created and initialized to 0 whereas scount is initialized only once and its value persists between function calls.

Features

1. Storage - Memory
2. Scope - Block or file scope depending upon where it is declared.
3. Lifetime - Persists between function calls if scope is block scope.
4. Default - Initial value zero.

9.3.4 Register Storage Class

The **register** keyword is used to tell the compiler to store the variable in a CPU register rather than in main memory. The register variables have similar features as the automatic storage class except for the storage location.

Advantages of register variables

The CPU has its own limited storage locations, which it uses for actual data operations. These locations are called registers. To manipulate data and perform operations, the CPU moves data back and forth between the memory and registers, which takes a finite amount of time.

Thus, if a particular variable is kept in the register itself, the CPU can access it faster. Hence, variables, which are heavily used, may be declared of this type so that execution is faster.

Syntax:

```
register data_type variable;
```

Example:

```
register int i;  
register char ch;
```

Limitations

- i. There are only a limited number of registers in the CPU. So, a register may not be available for the variable. In such a case, the variable is treated as an ordinary automatic variable.
- ii. Most compilers allow this storage class to be used only with integer data type. (int or char)
- iii. The unary & operator (address of) cannot be used with these variables either explicitly or implicitly.
- iv. It cannot be used with either static or external storage classes.
- v. It cannot be used for structures, arrays or unions.

Features

1. Storage - CPU registers
2. Scope - Block scope
3. Lifetime - Exists as long as control is within the block where it is defined
4. Default initial - value-garbage

9.3.5 Summary

The following table summarizes the storage classes, scope and initializations.

| Storage Class | Variable is declared | Visibility | Remarks |
|---------------|-------------------------|---|---|
| Static | Outside a function | Anywhere within the file | Are initialized only once, Values retained through function calls, default initial value is zero. |
| | Inside a function block | Function/Block Scope | |
| Extern | Outside a function | Anywhere within the file | If they are to be used in multiple files, they have to be declared in each function using the extern keyword. Initialization can be done only once-outside the functions. |
| Register | Inside a function/block | Function/block scope | limited number of registers, restriction on the type of variables, cannot use pointers for register variables, no default value |
| Auto | Inside a function/block | Function / block scope i.e. local to the function/block | Variable is initialized each time the function / block is entered, no default value. Does not exist outside function block where declared. |

Exercises

A. Predict the Outputs

1.

```
main( )
{ int i;
  i = abc( );
  printf("%d..." ;i);
  i = abc( );
  printf("%d",i);
}
static int abc()
{ int i = 1;
  return i++;
}
```

2.

```
extern int i;
main()
{ printf("%d",i);
}
```

3.

```
static int i = 100;
main()
{ static int i = 200;
  abc();
  printf("%d",i);
}
abc( )
{ printf("%d..", i);
}
```

4.

```
/* File aa.c */
int a = 100;
/* File bb.c */
#include "aa.c"
extern int a;
main( )
{ printf ("%d",a);
}
```

B. Review Questions

1. What do the following terms mean?
 - a. Scope
 - b. Extent
 - c. Storage class
2. What do you mean by block scope and file scope? Explain with examples.
3. What is meant by the storage class of a variable? Name the different storage classes in C.
4. What is meant by local variables?
5. Distinguish between local and global variables.

6. What are static variables? What are the two types of static variables?
7. Differentiate between automatic and static storage classes.
8. What is the purpose of the extern keyword?
9. What values does an un-initialized global variable contain?
10. What do you understand by block scope of a variable? How does nested blocks affect its accessibility?
11. What are the advantages and limitations of the register storage class?
12. When is the register storage class most useful?
13. Discuss the different storage classes in C.
14. Write two differences between auto and static variables.



Suggestive Readings

- ADAMS, G.B. III, AGRAWAL, D.P., And SIEGEL, H.J.: “ A Survey And Comparison Of Faulttolerant Multistage Interconnection Networks,” Computer, Vol. 20, Pp. 14–27, June 1987.
- ADAMS, K., And AGESEN, O.: “ A Comparison Of Software And Hardware Technqies For X86 Virtualization,” Proc. 12th Int’l Conf. On Arc H. Support For Prog. Lang. And Operating Systems, ACM, Pp. 2–13, 2006.
- AGESEN, O., MATTSON, J., RUGINA, R., And SHELDON, J.: “Software Techniques For Av Oiding Hardware Virtualization Exits,” Proc. USENIX Ann. Tech. Conf., USENIX, 2012.
- AHMAD, I.: “Gigantic Clusters: Where Are They And What Are They Doing?” IEEE Concurrency, Vol. 8, Pp. 83–85, April-June 2000.
- AHN, B.-S., SOHN, S.-H., KIM, S.-Y., CHA, G.-I., BAEK, Y.-C., JUNG, S.-I., And KIM, M.-J.: “Implementation And Evaluation Of EXT3NS Multimedia File System,” Proc. 12th Ann. Int’l Conf. On Multimedia, ACM, Pp. 588–595, 2004.
- ALBATH, J., THAKUR, M., And MADRIA, S.: “Energy Constraint Clustering Algorithms For Wireless Sensor Networks,” J. Ad Hoc Networks, Vol. 11, Pp. 2512–2525, Nov. 2013.
- AMSDEN, Z., ARAI, D., HECHT, D., HOLLER, A., And SUBRAHMANYAM, P.: “VMI: An Interface For Paravirtualization,” Proc. 2006 Linux Symp., 2006.
- ANDERSON, D.: SATA Storage Technology: Serial ATA, Mindshare, 2007.
- ANDERSON, R.: Security Engineering, 2nd Ed., Hoboken, NJ: John Wiley & Sons, 2008.
- ANDERSON, T.E.: “The Performance Of Spin Lock Alternatives For Shared-Memory Multiprocessors,” IEEE Trans. On Parallel And Distr. Systems, Vol. 1, Pp. 6–16, Jan. 1990.
- ANDERSON, T.E., BERSHAD, B.N., LAZOWSKA, E.D., And LEVY, H.M.: “Scheduler Activations: Effective Kernel Support For The User-Level Management Of Parallelism,” Acmtrans. On Computer Systems, Vol. 10, Pp. 53–79, Feb. 1992.
- ANDREWS, G.R.: Concurrent Programming—Principles And Practice, Redwood City, CA: Benjamin/Cummings, 1991.
- ANDREWS, G.R., And SCHNEIDER, F.B.: “Concepts And Notations For Concurrent Programming,” Computing Surveys, Vol. 15, Pp. 3–43, March 1983.
- APPUSWAMY, R., VAN MOOLENBROEK, D.C., And TANENBAUM, A.S.: “Flexible, Modular File Volume Virtualization In Loris,” Proc. 27th Symp. On Mass Storage Systems And Tech., IEEE, Pp. 1–14, 2011.
- ARNAB, A., And HUTCHISON, A.: “Piracy And Content Protection In The Broadband Age,” Proc. S. African Telecomm. Netw. And Appl. Conf, 2006.
- ARON, M., And DRUSCHEL, P.: “Soft Timers: Efficient Microsecond Software Timer Support For Network Processing,” Proc. 17th Symp. On Operating Systems Principles, ACM, Pp. 223–246, 1999.
- ARPACI-DUSSEAU, R. And ARPACI-DUSSEAU, A.: Operating Systems: Three Easy Pieces, Madison, WI: Arpacci-Dusseau, 2013.

- BRATUS, S., LOCASTO, M.E., PATTERSON, M., SASSAMAN, L., SHUBINA, A.: “From Buffer Overflows To Weird Machines And Theory Of Computation,” ;Login;, USENIX, Pp. 11–21, December 2011.
- BRINCH HANSEN, P.: “The Programming Language Concurrent Pascal,” IEEE Trans. On Software Engineering, Vol. SE-1, Pp. 199–207, June 1975.
- BROOKS, F.P., Jr.: “No Silver Bullet—Essence And Accident In Software Engineering,” Computer, Vol. 20, Pp. 10–19, April 1987.
- BROOKS, F.P., Jr.: The Mythical Man-Month: Essays On Software Engineering, 20th Anniversary Edition, Boston: Addison-Wesley, 1995.
- BRUSCHI, D., MARTIGNONI, L., And MONGA, M.: “Code Normalization For Self-Mutating Malware,” IEEE Security And Privacy, Vol. 5, Pp. 46–54, March/April 2007.
- BUGNION, E., DEVINE, S., GOVIL, K., And ROSENBLUM, M.: “Disco: Running Commodity Operating Systems On Scalable Multiprocessors,” ACM Trans. On Computer Systems, Vol. 15, Pp. 412–447, Nov. 1997.
- BUGNION, E., DEVINE, S., ROSENBLUM, M., SUGERMAN, J., And WANG, E.: “Bringing Virtualization To The X86 Architecture With The Original Vmware Workstation,” ACM Tr Ans. On Computer Systems, Vol. 30, Number 4, Pp.12:1–12:51, Nov. 2012.
- BULPIN, J.R., And PRATT, I.A.: “Hyperthreading-Aware Process Scheduling Heuristics,” Proc. USENIX Ann. Tech. Conf., USENIX, Pp. 399–403, 2005.
- CAI, J., And STRAZDINS, P.E.: “ An Accurate Prefetch Technique For Dynamic Paging Behaviour For Software Distributed Shared Memory,” Proc. 41st Int’l Conf. On Parallel Processing, IEEE., Pp. 209–218, 2012.
- CAI, Y., And CHAN, W.K.: “Magicfuzzer: Scalable Deadlock Detection For Large-Scale Applications,” Proc. 2012 Int’l Conf. On Software Engineering, IEEE, Pp. 606–616, 2012.
- CAMPISI, P.: Security And Privacy In Biometrics, New York: Springer, 2013.
- CARPENTER, M., LISTON, T., And SKOUDIS, E.: “Hiding Virtualization From Attackers And Malware,” IEEE Security And Privacy, Vol. 5, Pp. 62–65, May/June 2007.
- CARR, R.W., And HENNESSY, J.L.: “Wsclock—A Simple And Effective Algorithm For Virtual Memory Management,” Proc. Eighth Symp. On Operating Systems Principles, ACM, Pp. 87–95, 1981.
- CARRIERO, N., And GELERNTER, D.: “The S/Net’s Linda Kernel,” ACM Trans. On Computer Systems, Vol. 4, Pp. 110–129, May 1986.
- CARRIERO, N., And GELERNTER, D.: “Linda In Context,” Commun. Of The ACM, Vol. 32, Pp. 444–458, April 1989.
- CERF, C., And NAV ASKY, V.: The Experts Speak, New York: Random House, 1984.
- CHEN, M.-S., YANG, B.-Y., And CHENG, C.-M.: “Raidq: A Software-Friendly, Multiparity RAID,” Proc. Fifth Workshop On Hot Topics In File And Storage Systems, USENIX, 2013.